1. **🎯 Goal Recap**

We’ll implement a **character stat and skill system** using:

* **Base Stats**: Strength, Intelligence.
* **Skills**: Chopping, Mining, Attack.
* **Proficiency-based skill growth** (use = improve).
* **Stat-based performance boosts** (STR/INT enhance skills).

**🧱 Project Structure**

swift

CopyEdit

/Scripts/

├── player.gd

├── stat\_component.gd

├── skill\_component.gd

├── skill\_data.gd

**📄 1. stat\_component.gd**

**Handles base stats and leveling.**

gdscript

CopyEdit

# stat\_component.gd

extends Node

class\_name StatComponent

## Player base stats

var strength: int = 5

var intelligence: int = 5

var available\_points: int = 0

## Allocate points to stats

func allocate\_point(stat: String) -> void:

match stat:

"strength":

strength += 1

"intelligence":

intelligence += 1

\_:

push\_warning("Unknown stat: %s" % stat)

available\_points = max(available\_points - 1, 0)

## Add stat points on level-up

func gain\_stat\_points(amount: int) -> void:

available\_points += amount

**📄 2. skill\_data.gd**

**Holds data for each skill. Can be reused by different characters.**

gdscript

CopyEdit

# skill\_data.gd

extends Resource

class\_name SkillData

@export var skill\_name: String

@export var base\_power: float = 1.0

@export var proficiency: float = 0.0

@export var proficiency\_max: float = 100.0

# Internal XP tracker for the skill (used to level it up)

var proficiency\_xp: float = 0.0

const XP\_PER\_LEVEL: float = 10.0

func gain\_proficiency\_xp(amount: float) -> void:

proficiency\_xp += amount

while proficiency\_xp >= XP\_PER\_LEVEL and proficiency < proficiency\_max:

proficiency += 1

proficiency\_xp -= XP\_PER\_LEVEL

**📄 3. skill\_component.gd**

**Applies the stats + proficiency to skill calculations.**

gdscript

CopyEdit

# skill\_component.gd

extends Node

class\_name SkillComponent

@export var chopping: SkillData

@export var mining: SkillData

@export var attack: SkillData

var stat\_component: StatComponent

func \_ready():

stat\_component = get\_parent().get\_node("StatComponent")

## Utility: Calculates chopping effectiveness

func get\_chopping\_power() -> float:

return chopping.base\_power + \

(stat\_component.strength \* 0.5) + \

(stat\_component.intelligence \* 0.2) + \

(chopping.proficiency \* 1.0)

## Utility: Calculates mining effectiveness

func get\_mining\_power() -> float:

return mining.base\_power + \

(stat\_component.strength \* 0.4) + \

(stat\_component.intelligence \* 0.3) + \

(mining.proficiency \* 1.0)

## Utility: Calculates attack damage

func get\_attack\_damage() -> float:

return attack.base\_power + \

(stat\_component.strength \* 0.6) + \

(stat\_component.intelligence \* 0.2) + \

(attack.proficiency \* 1.5)

## Gain XP toward a skill

func use\_skill(skill\_name: String, xp\_gain: float) -> void:

match skill\_name:

"chopping":

chopping.gain\_proficiency\_xp(xp\_gain)

"mining":

mining.gain\_proficiency\_xp(xp\_gain)

"attack":

attack.gain\_proficiency\_xp(xp\_gain)

\_:

push\_warning("Unknown skill: %s" % skill\_name)

**📄 4. player.gd**

**Main player script that connects everything.**

gdscript

CopyEdit

# player.gd

extends CharacterBody2D

@onready var stat\_component = $StatComponent

@onready var skill\_component = $SkillComponent

func perform\_chop\_action():

skill\_component.use\_skill("chopping", 2.0)

var power = skill\_component.get\_chopping\_power()

print("Chopping Power: ", power)

func perform\_mining\_action():

skill\_component.use\_skill("mining", 2.0)

var power = skill\_component.get\_mining\_power()

print("Mining Power: ", power)

func perform\_attack():

skill\_component.use\_skill("attack", 1.5)

var damage = skill\_component.get\_attack\_damage()

print("Attack Damage: ", damage)

**📁 Resource Setup in Godot**

1. Create 3 SkillData resources in the editor:
   * res://Skills/chopping.tres
   * res://Skills/mining.tres
   * res://Skills/attack.tres
2. Set their names and base powers.
3. In the SkillComponent script in the editor, **drag in the resources** to the exposed variables.

**📘 In-Editor Documentation**

**StatComponent**

* Holds the player's core attributes.
* Use allocate\_point("strength") to assign a point to Strength.
* Automatically tracks available points from level-ups.

**SkillData**

* Tracks a skill's proficiency and XP.
* Proficiency affects power when used in formulas.

**SkillComponent**

* Combines stats and proficiency to calculate skill power.
* Use get\_<skill>\_power() to get effectiveness.
* Use use\_skill(name, xp) to increase skill proficiency.

**✅ Advantages of This System**

* **Composable:** Easy to extend by adding more skills.
* **Reusable:** Skills are Resources, can be shared or saved.
* **Customizable:** You can add modifiers like tools, gear, buffs later.
* **Scalable:** Add soft caps, diminishing returns, or traits easily.

1. **✅ Step-by-Step: Set Base Powers and Connect Skills**

**1. Set Up Base Powers in the Skill Resources**

1. Open Godot and navigate to the **FileSystem** tab.
2. Click on chopping.tres to open it in the Inspector.
3. In the **Inspector**, set the following:
   * skill\_name = "chopping"
   * base\_power = 2.0 (example value; adjust based on balance)
   * proficiency = 0.0 (starting level)
   * proficiency\_max = 100.0

Do the same for mining.tres and attack.tres, with appropriate base powers. For example:

* mining.tres: base\_power = 2.5
* attack.tres: base\_power = 3.0

These can be adjusted later for balancing.

**2. Connect the SkillData Resources to the SkillComponent**

1. Select your **Player node** in the Scene tree.
2. Find the SkillComponent (child node).
3. In the **Inspector**, you’ll see exposed fields for:
   * chopping
   * mining
   * attack
4. Click the 🔍 next to each field and assign the corresponding .tres file:
   * chopping → chopping.tres
   * mining → mining.tres
   * attack → attack.tres

Now your skill component has all three skills loaded and ready to use!

**▶️ Next Steps: Test Your System**

**3. Trigger Actions from Player Script**

If you're using keyboard inputs for testing:

gdscript

CopyEdit

# Inside player.gd

func \_unhandled\_input(event):

if event.is\_action\_pressed("ui\_accept"): # Replace with actual action

perform\_chop\_action()

if event.is\_action\_pressed("ui\_select"):

perform\_mining\_action()

if event.is\_action\_pressed("ui\_cancel"):

perform\_attack()

In the Godot Input Map (Project > Project Settings > Input Map), add custom actions like:

* chop\_action → assign to spacebar
* mine\_action → assign to M
* attack\_action → assign to A

Update your player.gd script to match those names.

**4. Monitor Output**

When you press the action keys, you should see something like:

yaml

CopyEdit

Chopping Power: 5.2

Mining Power: 5.0

Attack Damage: 6.5

Each time you trigger the action, skill XP increases and proficiency slowly rises.

**🧪 Optional (But Recommended) Next Steps**

Here are some great directions you can go from here:

| **Feature** | **Description** |
| --- | --- |
| **UI Display** | Show STR/INT, skill proficiency, and damage power on screen. |
| **Tool Influence** | Add modifiers for tools (axe, pickaxe, sword). |
| **Saving System** | Store player stats and skills using JSON or FileAccess. |
| **Skill Unlocks** | Unlock new abilities at certain proficiency levels. |
| **Stamina Usage** | Use INT to reduce stamina drain when using skills. |

1. **🎮 How to Use Stats and Skills With the Player**

We'll walk through:

1. **Attaching stat/skill components to the player**
2. **Calling skill functions in gameplay**
3. **Handling input or interactions**
4. (Optional) Making tools or enemies use them too

**✅ 1. Attach Stat and Skill Components to the Player**

Your Player scene should look something like this:

scss

CopyEdit

Player (CharacterBody2D or Node2D)

├── StatComponent (Node)

├── SkillComponent (Node)

**In the Scene Tree:**

* Add two child nodes to your Player node:
  + StatComponent (attach stat\_component.gd)
  + SkillComponent (attach skill\_component.gd)

**In the Script (player.gd):**

gdscript

CopyEdit

extends CharacterBody2D

@onready var stat\_component = $StatComponent

@onready var skill\_component = $SkillComponent

func \_ready():

print("Player strength: ", stat\_component.strength)

print("Mining power: ", skill\_component.get\_mining\_power())

**🛠️ 2. Perform Actions Using Skills**

Now you can use skills like this in your player code:

gdscript

CopyEdit

func perform\_chop\_action():

skill\_component.use\_skill("chopping", 1.5)

var power = skill\_component.get\_chopping\_power()

print("Chopping with power: ", power)

func perform\_attack():

skill\_component.use\_skill("attack", 2.0)

var damage = skill\_component.get\_attack\_damage()

print("Attacking for: ", damage)

**⌨️ 3. Hook Actions to Input**

Update player.gd to handle input:

gdscript

CopyEdit

func \_unhandled\_input(event):

if event.is\_action\_pressed("chop\_action"):

perform\_chop\_action()

elif event.is\_action\_pressed("attack\_action"):

perform\_attack()

**Input Map:**

Go to Project > Project Settings > Input Map and add:

* chop\_action → assign to Space
* attack\_action → assign to A

Now pressing those keys will trigger the skills using your STR/INT and proficiency.

**🧪 4. Integrate With Gameplay (Tree, Enemy, etc.)**

Let’s say you have a Tree scene with a health value. You’d do something like:

**tree.gd**

gdscript

CopyEdit

extends Node2D

var health: float = 10.0

func chop(damage: float):

health -= damage

if health <= 0:

queue\_free()

**In player.gd**

gdscript

CopyEdit

func try\_chop\_tree(tree: Node):

var power = skill\_component.get\_chopping\_power()

tree.chop(power)

skill\_component.use\_skill("chopping", 1.0)

**🧩 Summary of What You Now Have**

| **Element** | **Description** |
| --- | --- |
| StatComponent | Manages Strength/Intelligence and stat points |
| SkillComponent | Calculates skill power using stats and proficiency |
| SkillData | Tracks individual skill levels (chop, mine, attack) |
| Player | Calls those components to affect the game world |

1. **🌳 Step-by-Step: Creating a Chop-able Tree Scene**

**✅ Step 1: Create the Tree Scene**

1. **In Godot**, go to Scene → New Scene
2. Select **Node2D** as the root
3. Rename it to Tree
4. Add the following children:
   * Sprite2D → Assign a tree texture
   * CollisionShape2D → Use RectangleShape2D or CircleShape2D
   * (Optional) Area2D if you want to use proximity detection
5. Save the scene as Tree.tscn in a folder like res://Scenes/Trees/

**🧠 Step 2: Create and Attach the Script**

Create a new script named tree.gd and attach it to the root of Tree.tscn.

**tree.gd**

gdscript

CopyEdit

# res://Scripts/tree.gd

extends Node2D

@export var max\_health: float = 10.0

var current\_health: float = max\_health

func chop(damage: float) -> void:

current\_health -= damage

print("Tree health: ", current\_health)

if current\_health <= 0:

queue\_free() # Remove tree when chopped down

You can tweak max\_health in the Inspector for different tree types (e.g., small = 5, big = 20).

**🎮 Step 3: Add Tree to the Game World**

* Drag and drop Tree.tscn into your main game scene.
* Place it somewhere near the player for testing.

**🧩 Step 4: Let Player Interact With the Tree**

Now you need the player to detect and chop a nearby tree.

**Option A: Use Area2D (recommended for interaction zones)**

**Add to the Player Scene:**

* Add an Area2D node (call it ChopDetector)
* Add a CollisionShape2D (e.g., circle in front of the player)
* Set monitoring = true

**In player.gd:**

gdscript

CopyEdit

@onready var chop\_detector = $ChopDetector

func \_unhandled\_input(event):

if event.is\_action\_pressed("chop\_action"):

var overlapping = chop\_detector.get\_overlapping\_bodies()

for body in overlapping:

if body.has\_method("chop"):

var chop\_power = skill\_component.get\_chopping\_power()

body.chop(chop\_power)

skill\_component.use\_skill("chopping", 1.0)

This lets the player chop any tree within range.

**🧪 Step 5: Test It!**

1. Run the game
2. Walk up to the tree
3. Press your "chop" input key (e.g. Spacebar)
4. Watch health reduce in the console
5. Tree disappears when health <= 0

**🛠 Optional Next Steps**

* Add a **tree fall animation** or particle effect
* Drop **logs or items** when the tree is chopped
* Add **tree regrowth** or respawn timer
* Show a **chop progress bar or HP bar**

1. **📊 Breakdown of What's Happening**

You saw this output:

python-repl

CopyEdit

Mining with power: 4.5

rock health: 5.5

...

Mining with power: 4.5

rock health: 1.0

...

**➕ So we know:**

* **Initial health** of the rock: probably 10.0
* Each time you mine, rock loses **4.5 HP**
* Which means your **mining power = 4.5**

Let’s walk through how that **4.5 mining power** was computed.

**📐 How Mining Power Is Calculated**

In your skill\_component.gd, you probably have something like this:

gdscript

CopyEdit

func get\_mining\_power() -> float:

var base = mining.base\_power

var str\_bonus = 0.1 \* stat\_component.strength

var int\_bonus = 0.05 \* stat\_component.intelligence

return base + str\_bonus + int\_bonus

**Example numbers:**

Let’s assume:

| **Stat** | **Value** |
| --- | --- |
| mining.base\_power | 2.0 |
| strength | 20 |
| intelligence | 10 |

**Plug in:**

* Base power = 2.0
* STR bonus = 0.1 \* 20 = 2.0
* INT bonus = 0.05 \* 10 = 0.5

**🔢 Total mining power:**

CopyEdit

2.0 + 2.0 + 0.5 = 4.5

✅ That matches exactly what your debug output showed.

**💥 Rock Health Reduction**

You saw:

yaml

CopyEdit

rock health: 5.5

rock health: 1.0

So the rock's health started at 10.0:

* First hit: 10.0 – 4.5 = **5.5**
* Second hit: 5.5 – 4.5 = **1.0**
* Third hit: 1.0 – 4.5 = **< 0**, so rock is removed

**🧩 Summary**

| **Element** | **Value** |
| --- | --- |
| Base mining power | 2.0 |
| STR bonus (20 × 0.1) | 2.0 |
| INT bonus (10 × 0.05) | 0.5 |
| **Total mining power** | **4.5** |
| Rock HP | 10.0, decreased each hit by 4.5 |

**✅ Option 3: Base Class or Interface**

You create a common script that defines a **universal interaction method**. All interactable objects inherit or include this, and implement their own version of interact().

1. **✅ Step-by-Step Setup**

**1️⃣ Create a Base Script (e.g. interactable.gd)**

gdscript

CopyEdit

# File: res://scripts/interactable.gd

class\_name Interactable

extends Node

func interact(actor):

# Default behavior (optional)

pass

You can attach this to any Node, Node2D, or whatever your interactables extend.

**2️⃣ Inherit or Extend in Rock, Tree, etc.**

**Rock Example (rock.gd):**

gdscript

CopyEdit

extends Interactable # Or load via preload if needed

@export var max\_health: float = 10.0

var current\_health = max\_health

func interact(actor):

var power = actor.skill\_component.get\_mining\_power()

current\_health -= power

current\_health = max(current\_health, 0)

print("Rock HP:", current\_health)

if current\_health <= 0:

queue\_free()

**Tree Example (tree.gd):**

gdscript

CopyEdit

extends Interactable

@export var tree\_health = 8.0

func interact(actor):

var power = actor.skill\_component.get\_chopping\_power()

tree\_health -= power

print("Tree HP:", tree\_health)

if tree\_health <= 0:

queue\_free()

**Enemy Example (enemy.gd):**

gdscript

CopyEdit

extends Interactable

var health = 20

func interact(actor):

var damage = actor.skill\_component.get\_attack\_damage()

health -= damage

print("Enemy HP:", health)

if health <= 0:

die()

func die():

queue\_free()

**3️⃣ In Player Script: Clean, Generic Interaction**

gdscript

CopyEdit

func \_unhandled\_input(event):

if event.is\_action\_pressed("interact\_action"): # e.g., "E" or mouse

var overlapping = $InteractionDetector.get\_overlapping\_bodies()

for body in overlapping:

if body is Interactable:

body.interact(self)

* self is passed in so objects can access the player’s components (e.g. mining power)
* Now your player only cares that it's talking to **something that implements interact()**

**🟢 Pros of This Method**

| **✅ Benefit** | **💬 Description** |
| --- | --- |
| Clean logic | One interact() call works on any object |
| Type-safe | You'll see errors early if an object forgets to implement the method |
| No typos | No risk of silent failures from has\_method() |
| Scales well | Easy to add new objects (e.g. chests, doors, switches) |
| Autocomplete | Godot knows the method signature in the editor |

**⚠️ Optional Tip**

If some interactables **don't inherit from Interactable** (e.g. 3rd-party scripts), you can still mix in dynamic checks as a fallback:

gdscript

CopyEdit

if body is Interactable:

body.interact(self)

elif body.has\_method("interact"):

body.interact(self) # Optional fallback for edge cases

Would you like a quick .tscn + interactable.gd template to test this setup in your Godot project?