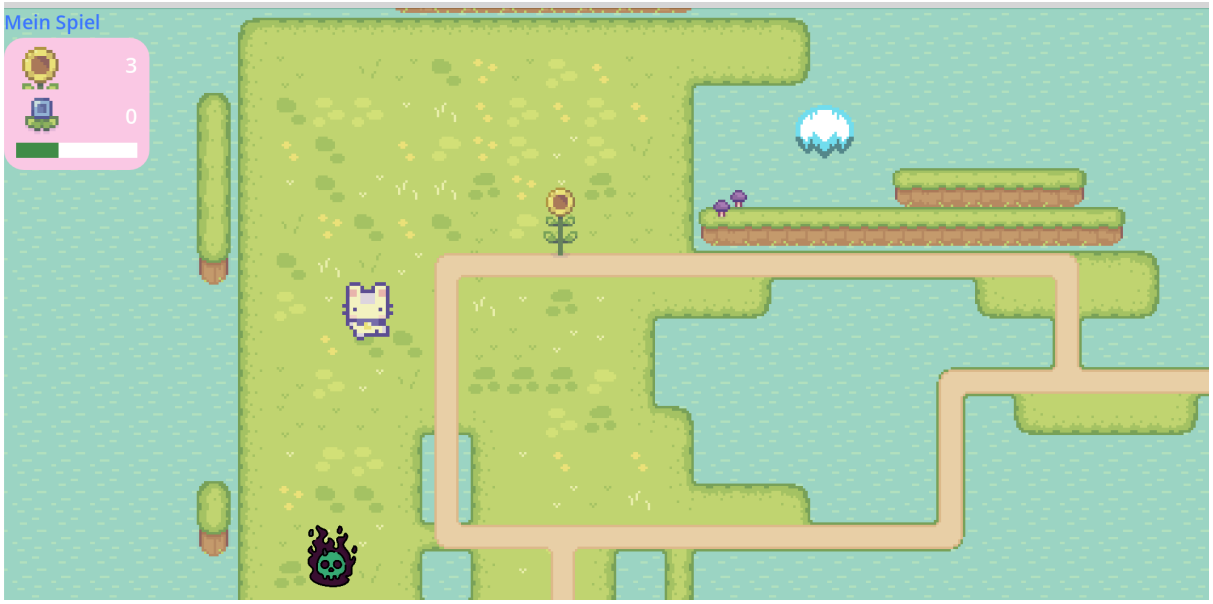


1 Ziele

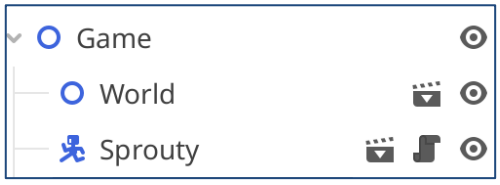
Teil 1: Unser Character ...
- sammelt Blumen (wird auf 150% seiner Größe gesetzt – einmalig, nicht stapelbar)
- trifft auf giftige Pilze (schrumpft um 10% - wiederholt sich)

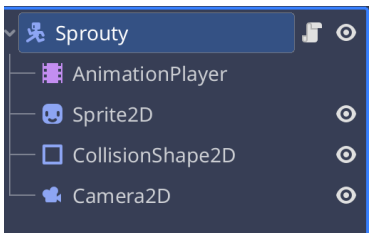
Die Blumen verschwinden, die Pilze bleiben stehen.

Teil 2: Signale wirken sich nicht nur auf den aktuellen Zustand der beteiligten Nodes aus (Blumen und Pilze, Character), sondern können (fast) jede andere Darstellung im Spiel beeinflussen. So können Barrieren verschwinden oder entstehen oder User Interface zeigt die aktuell gesammelten Ressourcen an.

2 Einleitung: Starterpack

Das Starterpack (siehe Ordner ‚04_Signals_begin') beinhaltet wieder Character und Welt:

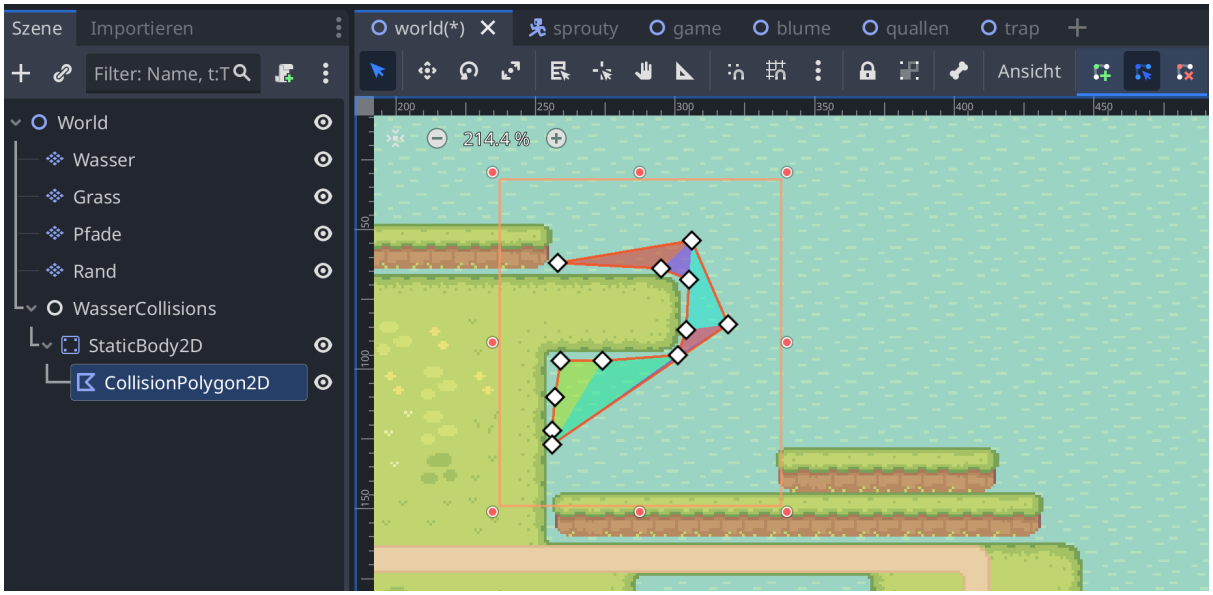




Sollten diese Elemente mal in einem anderen Spiel wiederverwendet werden, kann es gut sein das Charactere und Objekte neu sortiert bzw. umbenannt werden müssen. In diesem Fall müsstet ihr eine Kopie des Originalordners erstellen, und dann alle Veränderungen (umbenennen, verschieben, löschen) **direkt im Godot Editor** vornehmen. Also nicht außerhalb, direkt im File Explorer die Änderungen vornehmen.

Für ‚World' gilt es ein paar Sachen zu beachten:

- **einzelne Tiles** können direkt aus der *Tile Section* in die TileMap gezeichnet werden
- aus der *Terrain S*ection können **automatisiert Gebiete** gezeichnet werden
    - Wasser kommt zuerst als Fläche (Wellen sind animiert – siehe Abschnitt ‚Ressourcen' für ein Beispiel Youtube)
    - Danach kommt Gras als Fläche (hier bietet sich das automatisierte Malen aus dem ‚Terrain' an)
    - Danach Pfade als Linie einzeichnen, geht (noch) nicht als Fläche ...
    - Beliebtes Fettnäpfchen: Wasser könnte mit einem ‚physical layer' und ‚collision shapes' ausgestattet werden, um zu verhindern das der Character über Wasser läuft. **Aber:** TileMapLayer-Instanzen überlappen sich und die Kollisionen des unteren Layers (Wasser) ist weiterhin aktiv– selbst wenn der darüberliegende Layer (Land) eigentlich begehbar sein sollte. Godot erkennt bei der Kollisionsprüfung beide Layer unabhängig von deren Sichtbarkeit oder Layer-Hierarchie.
    - Lösungen:
        - Neues Terrain ‚Rand' als Begrenzung verwenden oder
        - Über dem Wasser einen CollisionPolygon2D platzieren (siehe Screenshot)



Die ‚Game' Scene:

a) kombiniert die einzelne Spieleelemente (World, Character) und später auch Game_UI oder Collectibles etc.

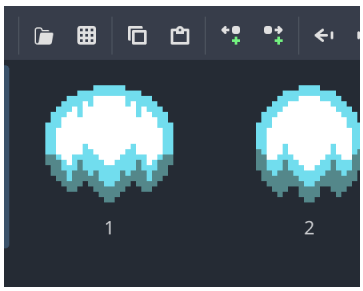
3 Übung

3.1 Lösche die bestehende Welt in den einzelenen TileMapLayern und entwerfe deine eigene Welt (≈ 15 min)

- Verwende sowohl automatisierte Terrains als auch einzelne Tiles

3.2 Quallen / Energy Flames (separate Szene, AnimatedSprite2D)

- Ziel: Szene einer animierten Qualle / Flame, die beim Kontakt mit dem Spieler verschwinden und ein Signal (Trigger und / oder Daten) sendet
- Animierte Qualle: Benötigt ein AnimatedSprite2D, diesem kann das BigEnergyBall.png zugefügt werden (per Raster Icon). Nachdem die vertikale und horizontale Anzahl der Frames eingegeben wurde, wird die Größe einer einzelnen Kachel berechnet (z.B. 24 x 24). Das ‚Autoplay Icon' sorgt dann dafür, dass die Animation nach dem Laden des Spiels auch automatisch startet.



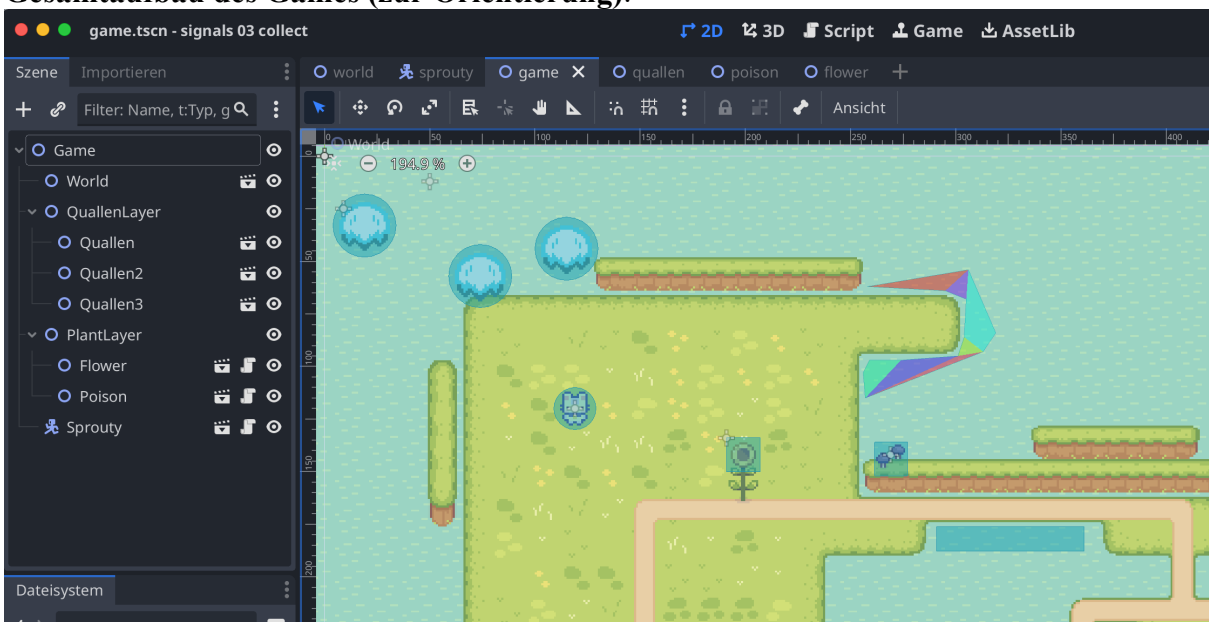
4 Direkte Signalauswertung (Character ⇔Flower)

Das Godot-Signalsystem ist eine Umsetzung des ‚observer patterns' aus der Informatik. Ereignisse, die innerhalb des Spiels stattfinden, senden Signale aus, die von ‚beobachtenden Objekten' registriert werden. Beobachterobjekte reagieren auf die Ereignisse, indem sie ‚Handler-Code'(bzw. die Signalfunktion) ausführen.

Zwischen dem Sender und dem Beobachter besteht eine Kopplung, z.B.:

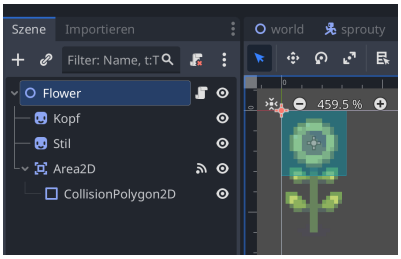
- Trigger (ein mobiler Character)
- Sender / Akteur (eine Blume, wenn ein Character auf sie trifft)
- weitere Signal-Beobachter (eine UI Anzeige, die Aktionen in einem Spielstand anzeigt)

Gesamtaufbau des Games (zur Orientierung):



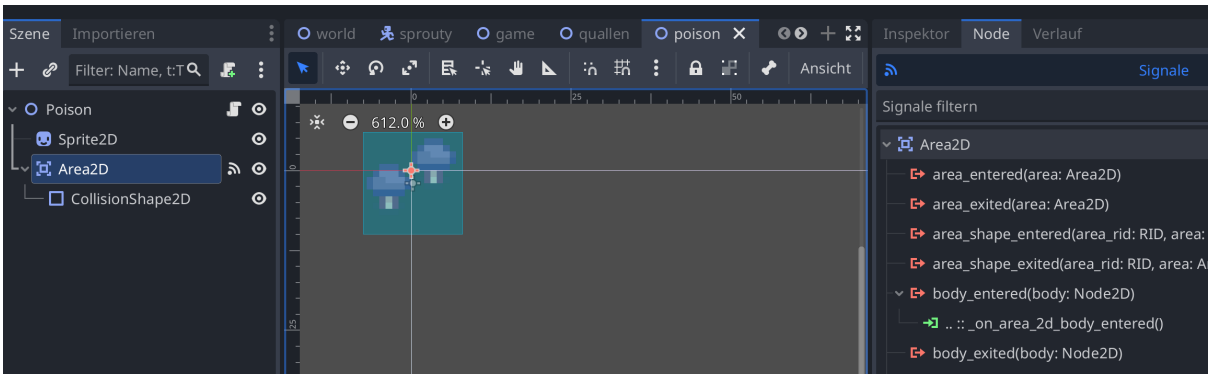
Aufbau ‚sendendes Objekt‘:

- CollisionShapes können nur an ‚Bodys‘ oder ‚Areas‘ hängen, in dieser Situation hat der ‚Area2D‘ Node schon eine Menge vorimplementierter ‚Signale‘
- D.h. die Flower Szene bekommt
    a) ein Sprite2D für die grafische Darstellung
    b) ein Area2D für Signal Initiierung



Signaleinstellung:

- hier braucht es ein Script in der Szene (z.B. im Knoten Poison), um
    a) den emittierenden Node oder den Character zu verändern und/oder
    b) ein Signal für den Rest des Games zu versenden (EventController.emit_signal (“flower_met“)



4.1 Direkt Kopplung zwischen Signal und Wirkung

Bei Signalen gilt es deren Verknüpfung zu beachten:

- Im folgenden Beispiel befinden sich Signal und Handlercode in der selben Szene; d.h. das Signal kann alles in der der Player Szene (per ‚body‘) verändern und alles in der aktuellen Szene (per ‚self‘). Im aktuellen Beispiel ist das die ‚Flower‘-Szene.
- Der Aufbau ist fehleranfälliger, wenn sich z.B. der Name eines Knotens ändert. Spiele können mehrere hundert Knoten haben und nicht immer ist nachvollziehbar, wie oft ein Knoten mit einem obsoleten Namen noch referenziert wird.

Signalfunktion:

```
func _on_area_2d_body_entered(body: Node2D) -> void:
    print ('entering body was: ', body)
    if body.name == 'Sprouty':
        body.scale = Vector2(1.5,1.5)
```

Die letzte Zeile zeigt an was vom Signal bewirkt wird. Folgende Tabelle zeigt verschiedene Wirkungen, die durch das Signal alternativ getriggert werden können, anstelle von `body.scale = Vector2(1.5,1.5)`

- self = Blume
- body = Player

`body.queue_free()`	**Existenz**: Der Player verschwindet; Achtung: damit verschwindet auch die Kamera, die am Player hängt
`body.visible = false`	**Sichtbarkeit**: Player wird unsichtbar; bewegt sich aber noch
`body.scale *= Vector2(1.5,1.5)` `body.scale = Vector2(1.5,1.5)`	**Größe**: Player vergrößert sich (wiederholt) bzw vergrößert sich (einmalig)

`body.speed *= 0.5`
`body.speed = 0.5`
`body.speed = 400`
	Geschwindigkeit: Player-Geschwindigkeit halbiert sich (wiederholt) bzw. halbiert sich (einmalig)
	- wird auf 400 Pixel pro Sekunde gesetzt

`self.queue_free()oder queue_free()`	Die Blume verschwindet bzw. alle vorherigen Anweisungen können wieder analog verwendet werden (ausser ‚speed')

========== Ende Teil 1 ==========

4.2 Signale im Abo-Modell (mittels Event Controler)

Aufgabe des Event-Controller ist vor allem das *Verteilen* und *Koordinieren* von Signalen. Die Reaktionslogik findet dann in den betroffenen Szenen statt. Durch Signale und Event-Controller erhält das Projekt eine **ereignisgetriebene Architektur.**

Signale ‚verschlanken' die Frames (nur tatsächlich ausgelöste Events werden verarbeitet). Somit braucht das Spiel weniger Hardware Ressourcen etc. Ein Frame ist jede Iteration des Game-Loops (→ Update → Physik → Render).

Auch entscheidend für **inklusives Design**: zusätzliche Listener (z. B. für Screenreader-Ausgaben, haptisches Feedback oder alternative Eingabegeräte) können ohne Eingriffe in Kernsysteme angebunden werden – ein essenzieller Faktor für Accessibility-Features.

Team Workflow wird verbessert (?). Designer*innen können Logik per Visual-Scripting (Godot-Signale im Inspector) anschließen, während Programmierer*innen das Backend verantworten. Die Rollen sind klar abgegrenzt, was partizipative Entwicklungsprozesse erleichtert.

4.3 Wie implementiere ich Signale im Abo-Model ?

5 Schritte:

1) Signal definieren
signal im Singleton 'event_controller.gd' hinzufügen
Signal got_poisened

2) Singleton als global verfügbare Autoload-Ressource einrichten
Singleton in Projekteinstellungen / Globals / Autoload eintragen und aktivieren.

- *Script-Datei* (**event_controller.gd**) = Quelltext, der eine Klasse definiert.
- *Node-Instanz* (**EventController**) = zur Laufzeit erzeugtes Objekt dieser Klasse, das Teil des Szenenbaums oder – bei Autoload – Teil der Root-Ebene ist.

2) Signal senden
als Autoload kann das neue Script / der neue Node 'event_controller.gd' in jeder Szene verwendet werden, z.B. in flower.gd
EventController.emit_signal('met_flower').

3) Signal abonnieren
jede Szene, die das Signal auswerten möchte, muss sich zuerst mit dem entsprechenden Singleton, in ihrer _ready() Funktion, verknüpfen und braucht dafür zwei Parameter, Signalname und auszuführende Funktion.
EventController.connect("met_flower", on_event_met_flower)

4) Signal auswerten
die im dritten Schritt angeführte Funktion beinhaltet dann die gewünschte Reaktion, z.B.

```
# diese function wäre dann in der ‘UI‘ Szene
func on_event_met_flower():
	print('UI triggered')
```

5) @onready verwenden
Insbesondere wenn das Signal eine Anzeige in der UI ändern soll. kann der Pfad zum zu verändernden Label recht lang sein. UIs sind meist geschachtelte Control Nodes (Kontrolle, i.S. einer Nutzer:innensteuerung).

Wenn ich den gewünschten Node (z.B. ein ‚Label‘ Node für eine gewünschte Anzeige) in eine freie Zeile noch vor die Funktion _ready(): ziehe und dabei die STRG Taste gedrückt halte, dann sollte eine ähnlich strukturierte Zeile automatisch erscheinen:

@onready var flower_count: Label =
$VBoxContainer/PanelContainer/MarginContainer/VBoxContainer/Flower/FlowerCount

Dadurch kann ich nun die kurze Bezeichnung flower_count anstelle des langen Zugriffspfades aus dem Szenenbaum einsetzen. Im konkreten Beispiel müssen dann noch die Formate angepasst werden, d.h. mein Counter ist eine Zahl und mein flower_count.text ist eine String / Text. Daher brauche ich die Funktion str() um eine Zahl in einen Text umzuwandeln.

```
func on_event_met_flower():
        print('UI triggered')
        counter += 1
        flower_count.text = str(counter)
```

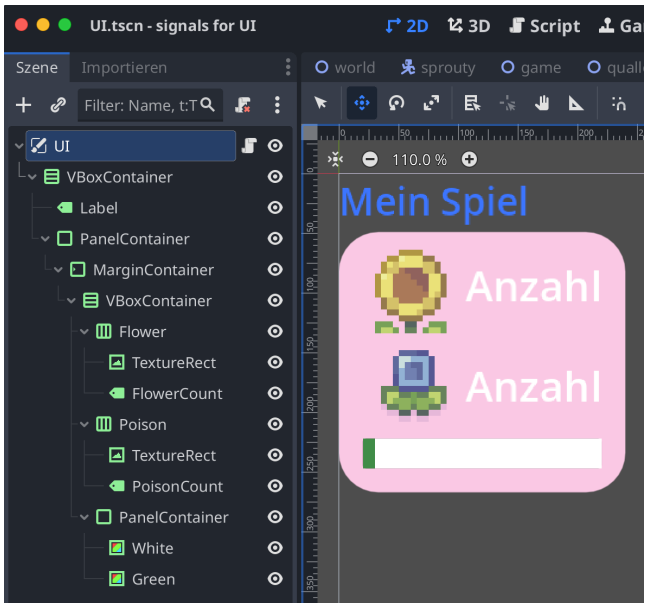
5 UI Szene

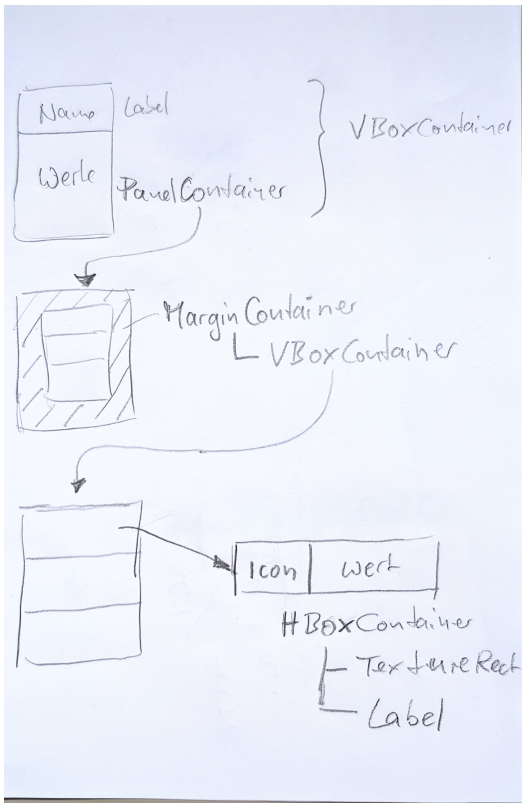
Godot unterscheidet zwischen Layers und Szenenbäumen. Innerhalb einer Canvas entscheidet die Baumhierarchie; zwischen verschiedenen Canvas-IDs entscheidet allein das layer-Feld. Renderer sortiert strictly nach dieser Zahl → Layer 0 zuerst, dann 1, 2 … – unabhängig von Scene-Baum-Position oder Ladezeitpunkt.

➔ siehe auch HUD Beispiel in ‚99_Gemüselauf_Game‘

Vorteile des CanvasLayer-Ansatzes

- **Unabhängig von Kameratransform**: HUD bleibt fixiert, selbst wenn die Kamera zoomt oder rotiert.
- Jeder Layer bekommt explizite Position zugeordnet (0 = Welt, 1 = HUD, 2 = Accessibility, 3 = Debug).
- **Barrierefreiheit**: Screen-Reader- oder High-Contrast-Layer ScreenReader_layer ≥ HUD_layer+1).





- **Healthbar** verwendet 2 x ColorRect
- Minimum Size im Inspector entspricht der Länge des Lebensbalken (z.B. 200)
- Container Sizing des grünen Balkens wird auf ‘Zum Anfang verkleinern’ gesetzt

6 Übung

1) Implementiere ein CollisionPolygon2D, um an einer Stelle die Bewegungsfreiheit deines Characters einzuschränken.

2) Unter Assets gibt es ein Bild ‚auge.png'. Kontakt mit dem Auge sollte alle Wege ausblenden und erneuter Kontakt sollte wieder alle Wege einblenden.

3) Zur Zeit werden nur die Blumen aber nicht die giftigen Pilze getrackt. Zähle die Pilzkontakte und reduziere den Lebensbalken um jeweils 10%.

4) Erstelle Deine Eigene HUD / UI.

6.1 Weitere Nutzung von Signale

Beispiel
- Spieldauer wird angezeigt: Timer Node sendet ein Signal im Sekundentakt um ein Update der Zeitanzeige zu triggern
  ➔ siehe Beispiel ‚99_Collectibles_mit_Timer'
- Per Escape Taste erscheint dann ‚Pause' und die Zeitanzeige bleibt stehen: InputController (ein Autoload Singleton) beobachtet Tastatureingaben und triggert ein Signal sobald die Escape Taste betätigt wird

7 Background-Wissen

7.1 Wie setzt sich der Kopf einer Signalfunktion zusammen?

Beispiel `_on_area_2d_body_entered(body)`

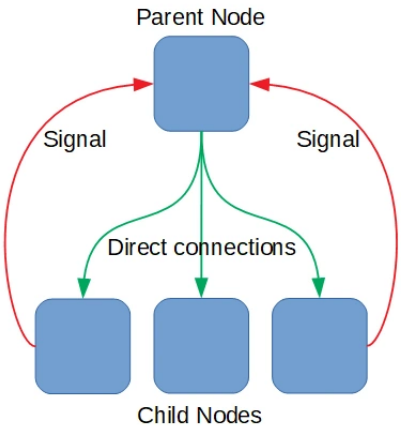
➔ **Präfix**, **emittierender Node-Name** und **Signal-Name**

- Das Präfix `_on_` signalisiert, dass diese Funktion ein Signal-Callback ist, d. h., sie wird **automatisch aufgerufen**, wenn ein spezifisches Signal von einem Node ausgelöst wird
- Der Name `area_2d` bezieht sich auf Signalsender -Node (könnte auch `flame_area_2d` heißen)
- `body_entered` ist der Name des Signals, das vom `Area2D`-Node ausgesendet wird
- Das Argument `body` ist eine Referenz auf den Körper (Node), der das Signal ausgelöst hat (typischerweise ein `CharacterBody2D`, `RigidBody2D`, oder ein anderer PhysicsBody2D). Der Node-Name kann dann auch per `print(body.name)` `abgefragt` werden.

7.2 Lose und enge Verknüpfung

Unterschied zwischen loser und enger Verknüpfung zwischen Signalen und Knoten.

... siehe auch https://gdscript.com/solutions/signals-godot/:



7.3 Typische Singletons

- *Game Controller* (z.B. Spielerstatus wie Gesundheit, Inventar abspeichern
- *Audio Controller* (z.B. Wechsel der Hintergrundmusik, Lautstärkeregelung, Soundeffekte wie Schritte, Einstellungen im Audio Controller bleiben über Szenen hinweg erhalten)
- *Input Controller* (z.B. Pause, Mini-Map, Erklärungen, Inventarübersicht anzeigen)

8 Ressourcen

How to Create Animated TileSets in Godot 4
https://www.youtube.com/watch?v=AO-pqAvzowk&ab_channel=DevDuck

How To Create Collectible Objects In Godot (Tutorial)
Quelle: https://www.youtube.com/watch?v=Rh_8UXjYTn4