Ein Bild, das Screenshot, Diagramm, Pixel enthält.

KI-generierte Inhalte können fehlerhaft sein.

# 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:** Blumen und Pilze wirken sich nicht nur auf den aktuellen Zustand des Characters aus, sondern verändern auch ein zentrales Punktekonto (z.B. es entsteht ein Character-Build).

# Einleitung: Starterpack

Das Starterpack (siehe Ordner ‚04\_Signals\_begin‘) beinhaltet Character und Welt:

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein. Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein. Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

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‘)**
  + Danach Gras als Fläche
  + Danach Pfad als Linie, geht nicht als Fläche ...
  + Beliebtes Fettnäpfchen: Wasser könnte mit einem ‚physical layer‘ und ‚collision shapes‘ ausgestattet werden, um zu verhindern das der Cjharacter 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)

Ein Bild, das Screenshot, Text, Karte, Software enthält.

KI-generierte Inhalte können fehlerhaft sein.

Die ‚Game‘ Scene:

* + kombiniert die einzelne Spieleelemente (World, Character) und später auch Game\_UI oder Collectibles etc.

# Übung

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

## 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 dann nach dem Laden des Spiels auch automatisch startet.

Ein Bild, das Screenshot, Diagramm, Grafiken, Pixel enthält.

Automatisch generierte Beschreibung

## Ein erstes Signal ‚Flower made contact‘

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‘ ausführen. Zwischen dem Sender und dem Beobachter besteht eine lose Kopplung, so dass sie unabhängig voneinander funktionieren können.

* Das triggernde Objekt (ein mobiler Character)
* Das sendende Objekt (eine Blume, wenn ein Character auf sie trifft)
* Das beobachtenden Objekt (eine UI Anzeige, die Aktionen in einem Spielstand festhält)

## Sendendes Objekt

**Gesamtaufbau**:

Ein Bild, das Multimedia-Software, Software, Grafiksoftware, Diagramm enthält.

KI-generierte Inhalte können fehlerhaft sein.

**Aufbau ‚sendendes Objekt‘**:

* CollisionShapes können nur an ‚Bodys‘ oder ‚Areas‘ hängen, in dieser Situation hat der ‚Area2D‘ Node schon eine Menge vorimplementierter ‚Signale‘ – in der Regel eine gute Wahl ...

Objektstruktur:

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Signaleinstellung:

Ein Bild, das Text, Software, Multimedia-Software, Grafiksoftware enthält.

KI-generierte Inhalte können fehlerhaft sein.

**Wie setzt sich der Kopf eine 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)

### Direkt Kopplung zwischen Signal und Wirkung

Im folgenden Beispiel wird ein Counter hochgesetzt (flame\_collected) und danach verschwindet die Flame / Qualle. Ist fehleranfälliger, wenn sich z.B. der Name des Knotens ändert.

extends Node2D

@export var value: int = 1 # Wert einer Energieflame, später anpassbar

func \_on\_area\_2d\_body\_entered(body):

print('body: ', body)

if body is Player:

self.queue\_free()

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 self.queue\_free()

* self = Blume
* body = Player

|  |  |
| --- | --- |
| body.queue\_free() | **Existenz**: Der Player verschwindet; Achtung: damit verschwindet auch die Kamera |
| body.visible = false | **Sichtbarkeit**: Player wird unsichtbar; bewegt sich aber noch |
| body.scale \*= Vector2(1.5,1.5) | **Größe**: Player vergrößert sich |
| body.speed \*= 0.5  body.speed = 400 | **Geschwindigkeit**: Player-Geschwindigkeit halbiert sich bzw. 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 ==========================

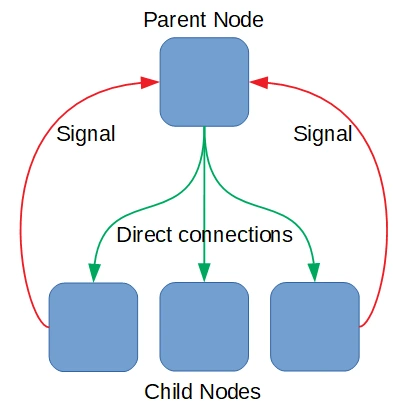
### Lose Kopplung zwischen Signal und Wirkung

Welcher Knoten soll das Signal abonnieren?

* Originäres Problem: es wird nicht empfohlen direkt auf die Methoden des ‚parent nodes‘ oder Nodes in anderen Szenen zuzugreifen.
* Warum? Ein eng gekoppelte Pfad zur einer Methode kann leicht unterbrochen werden, wenn eine Unterknoten im Spiel solo getestet wird. Daher wird die Verbindung von Signalen mit Knoten ‚lose coupling‘ genannt.

Unterschied zwischen loser und enger Verknüpfung zwischen Signalen und Knoten.

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



## Autoload Singletons: Zentrales Punktekonto und zentraler Eventmanager

**Ziel**: einen ‚Energiepunkt‘ je eingesammelte Flamme in die zentrale Punkteübersicht eintragen

**Was sind Autoload Singletons?**

* globale Instanz eines Skripts oder einer Szene
* **Autoload**: wird zum Spielstart geladen und ist somit für alle anderen Skripte und Szenen im Projekt verfügbar
* **Singleton:** es wird immer dieselbe Instanz zur Laufzeit verwendet, d.h. Daten oder Zustände im Singleton bleiben über verschiedene Szenen hinweg persistent

**Wie werden Autoload-Singletons eingerichtet?**

1. Scriptewerden keiner Szene zugeordnet, sondern per Autoload geladen
2. Gehe zu ***Projekt > Projekt-Einstellungen > Autoload***
3. Füge ein Skript oder eine Szene hinzu und gib ihr einen globalen Namen   
   (z.B. GameController)
4. Danach können die Inhalte an jeder Stelle im Projekt über diesen globalen Namen aufgerufen werden

**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)

Beispiele: ‚event\_controller‘ und ‚game\_controller‘ Scripte

1. event\_controller definiert ein Signal

extends Node

signal flame\_collected (value:int)

1. game\_controller speichert Punkteveränderungen UND sendet ein Signal mit dem neuen Punktestand (unter Nutzung des event\_controller)

extends Node

var total\_flames: int = 0

func flame\_collected (value:int):

total\_flames += value

EventController.emit\_signal("flame\_collected", total\_flames)

## Mehrfache Instanzen des Energy Flame Nodes in der World Scene aufrufen

* Beliebig viele Instanzen sind möglich (STRG + D)

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

* um Spielinformation immer im Sichtfeld zu haben, wird die Game\_UI scene
* alternativ kann ein Extra ‚CanvasLayer‘ Node verwendet werden (enable die Einstellung ‚Follow Viewport‘)

## Weitere Nutzung von Signale

Beispiel

* Spieldauer wird angezeigt: Timer Node sendet ein Signal im Sekundentakt um ein Update der Zeitanzeige zu triggern
* 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

Oder auch ...

* Mit jeder Flame wird der Sprite größer oder schneller

# 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>

Ein Bild, das Text, Screenshot, Software, Website enthält.

Automatisch generierte Beschreibung