

Workshopopdracht: Je eerste enemy maken in Godot

Doel van de opdracht

In deze workshop maak je stap voor stap je eigen enemy voor de game. Je leert:

- Wat nodes en scenes zijn
- Hoe je een sprite met animatie toevoegt
- Wat variabelen en functies zijn
- Hoe je code schrijft om beweging te maken
- Hoe je de enemy automatisch naar een punt laat bewegen
- Hoe je collision detectie gebruikt
- Hoe je scripts hergebruikt

Belangrijk: We gaan dit in kleine stappen doen. Elke stap bouwt voort op de vorige. Neem de tijd om elk concept te begrijpen voordat je verder gaat! Veel succes, als je vast zit, vraag om hulp!

Het project verkennen

1. Open het project in Godot.
2. Start de game door op de **playknop** rechts boven te drukken (F5).

Je ziet dat er al veel in de game zit: een speler die kan bewegen en aanvallen, platforms, en een hart dat je moet verdedigen. **Alleen de enemies ontbreken nog** — die ga jij maken!

Wat zijn nodes? (Theorie)

Bekijk aan de linkerkant van je scherm de **hiërarchie** (Scene Tree). Hier zie je de onderdelen die ik heb toegevoegd voor de aanwezige functionaliteit.

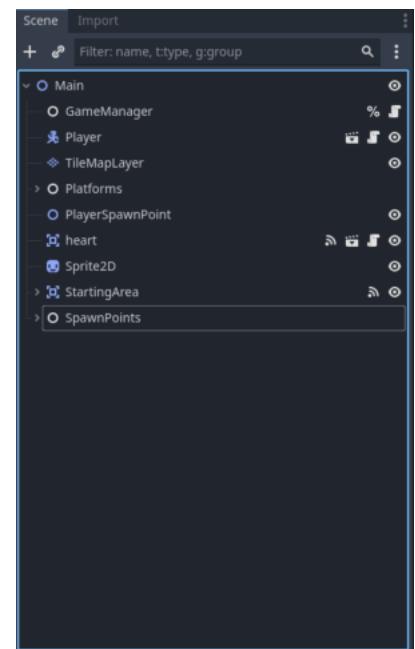
In Godot bestaat een game uit **nodes**. Een node is een bouwblok met een specifieke functie.

Voorbeelden:

- **Node2D**: Heeft een positie op het scherm (x, y coördinaten)
- **Sprite2D**: Toont een afbeelding
- **CharacterBody2D**: Een karakter dat kan bewegen en physics heeft

Waarom is dit handig?

Godot heeft deze nodes als basis functionaliteit al voor je gemaakt, zodat je niet alles zelf hoeft te programmeren. Je combineert verschillende nodes om complexere dingen te maken.



Nodes kunnen andere nodes bevatten:

Bijvoorbeeld, een Enemy node kan bevatten:

- Een AnimatedSprite2D (voor de visuals)
- Een CollisionShape2D (voor botsingen)
- Een Area2D (voor detectie)

Scenes:

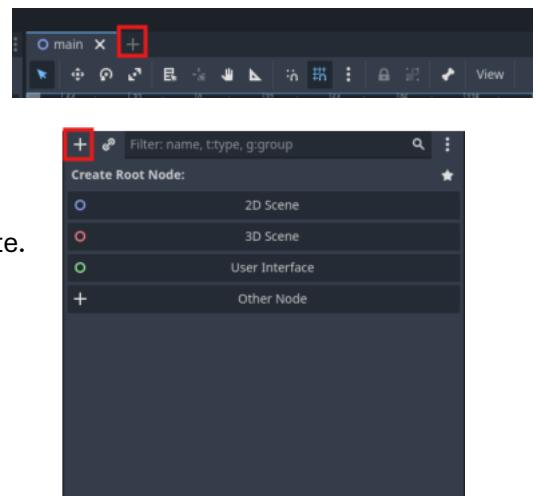
Een collectie van nodes kun je opslaan als een **scene**. Dit is een herbruikbaar onderdeel. Als je 10 enemies wilt, maak je 1 enemy scene en hergebruik je die 10 keer!

Basis van de enemy maken

1. Een nieuwe enemy scene maken

Nu gaan we onze eigen enemy scene maken!

1. Klik linksboven op het **plusje** naast "Main" om een **New Scene** te maken.
2. Je krijgt een leeg scherm. Klik op het **plusje** linksboven (of druk **Ctrl + A**) om een node toe te voegen.
3. Zoek naar **CharacterBody2D** en selecteer die en klik create.
4. Hernoem de node naar **Enemy** (dubbelklik erop).
5. Sla de scene op met **Ctrl + S**.
 - Map: scenes/
 - Bestandsnaam: enemy.tscn



Waarom CharacterBody2D?

Deze node heeft ingebouwde functionaliteit voor beweging en physics. Perfect voor een bewegend karakter!

2. Een animatie toevoegen

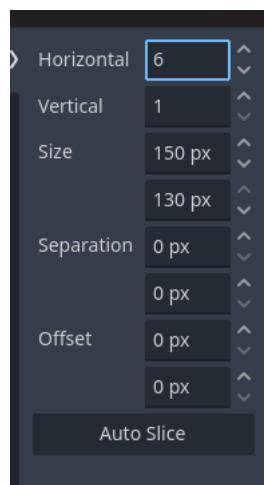
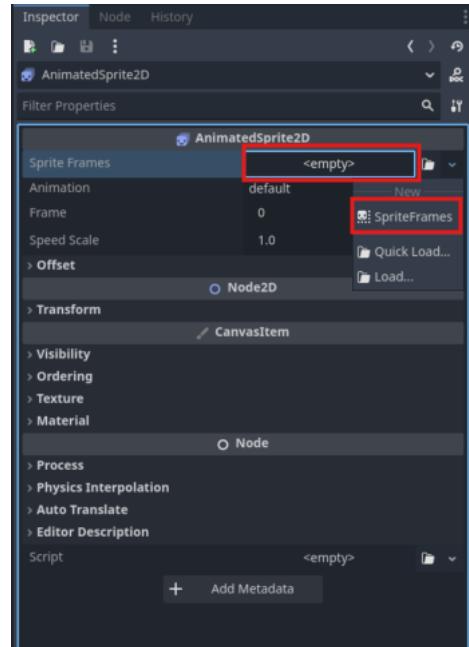
De enemy is nu nog onzichtbaar. We gaan een **AnimatedSprite2D** toevoegen voor de visuals.

1. Zorg dat je **Enemy** node geselecteerd is, door er 1 keer op te klikken.
2. Voeg een nieuwe node toe (**Ctrl + A** of het plusje).
3. Zoek naar **AnimatedSprite2D** en voeg deze toe.
4. Selecteer de AnimatedSprite2D node.
5. Kijk naar het **Inspector** paneel rechts.

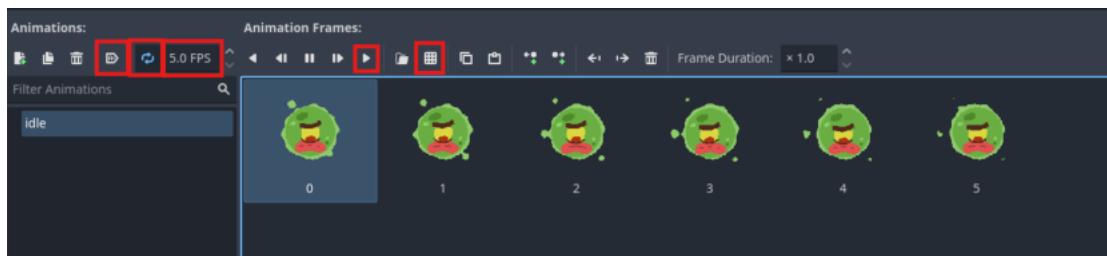
- Bij sprite frames klik op <empty> → **SpriteFrames** onder new.
- Klik nogmaals op **Sprite Frames** om het animatie venster te openen (onderkant scherm).

De idle animatie maken:

- Er is al een default animatie genaamd "default". Hernoem deze naar **idle**.
- Klik bovenaan in het midden op het icoontje "**Add frames from sprite sheet**" (grid icoontje bij animation panel).
- Navigeer naar: assets/sprites/enemy/idle.png en klik open
- Stel in:
 - Horizontal:** 6 (6 kolommen)
 - Vertical:** 1 (1 rij)
- Selecteer alle 6 frames van links naar rechts (sleep van links naar rechts).
- Klik op **Add 6 frame(s)** onderaan dit window
- Als het goed is staat **Loop** al aangevinkt, blauw loop icoontje (de animatie moet blijven herhalen).
- Vink **Autoplay** aan, links van de loop (start automatisch als de enemy spawnt).
- Zet de frame rate (Frames Per Second, hoeveel keer de animatie door gaat naar het volgende plaatje per seconde) naar **10.0 FPS**, of een andere test uit wat je een goede frame rate vind, hoe hoger hoe sneller de animatie afspeelt.



Test het, druk op de **Play** icon je zou de animatie nu moeten af zien spelen. Je hebt nu dus van 1 afbeelding meerdere afbeeldingen (frames) gemaakt, die samen een kleine animatie maken!



3. Een script toevoegen

Nu gaan we de **logica** van de enemy programmeren.

1. Selecteer de **Enemy** node (de root, de CharacterBody2D).
2. Klik onderaan bij script op **<empty>** → **New script** (scroll icoon met plusje).
3. Nu komt er een window waar je naast path een map icoontje kan klikken, kies hierbij om de script op te slaan op:
 - Map: scripts/
 - Bestandsnaam: enemy.gd
4. Check dat achter path staat: **res://scripts/enemy.gd**
5. Klik op **Create** om het script te maken

Je ziet nu een leeg script met alleen:

```
extends CharacterBody2D
```

Dit betekent: "Dit script zit op een CharacterBody2D en kan al zijn functies gebruiken."



4. Variabelen - Data opslaan (Theorie + Praktijk)

Wat is een variabele?

Een variabele is een stukje data dat je opslaat om later te gebruiken.

Denk aan een variabele als een **doosje met een label**:

- Het doosje heeft een **naam** (bijvoorbeeld "health")
- In het doosje zit een **waarde** (bijvoorbeeld 3)
- Je kunt de waarde **uitlezen** ("hoeveel health heb ik?")
- Je kunt de waarde **veranderen** ("ik krijg damage, health wordt 2")

Waarom zijn variabelen handig?

Stel je voor: je wilt de health van de enemy op 10 plekken in je code gebruiken. Als je het als variabele opslaat, hoef je maar op 1 plek te veranderen!

Een simpele variabele maken:

Voeg dit toe bovenaan in je script (onder extends CharacterBody2D):

```
var speed = 60
var enemy_name = "Bacterie"
```

Uitleg:

- var betekent "maak een variabele"
- speed is de naam van de variabele

- = 60 is de waarde die erin zit

Verschillende types data:

- **Getallen:** var health = 3
- **Tekst:** var name = "Bacterie" (tussen aanhalingstekens!)
- **Waar/Niet waar:** var is_alive = true

5. De _ready functie - Eenmalige setup

Wat is een functie?

Een functie is een **blok code dat iets doet**. Je kunt het zien als een recept: een stappenplan dat de computer volgt.

De _ready functie:

Godot roept deze functie automatisch aan **1 keer** wanneer de node in de game komt (meestal aan het begin).

Voeg deze functie toe aan je script, onder de variables:

```
func _ready():
    print("Enemy spawned!")
    print(enemy_name)
```

Uitleg:

- func betekent "dit is een functie"
- _ready(): is de naam van de functie
- Alles wat **ingesprongen** staat (met een tab) hoort bij de functie
- print() schrijft iets naar de console (onderaan je scherm)

Test het!

1. Sla je script op (Ctrl + S)
2. Ga terug naar je main scene, en klik op 2D
3. Instantieer een enemy in je Main scene (kettinkje → kies enemy.tscn)



4. Druk op Play (F5)
5. Kijk onderaan naar de **Output** console
6. Klik op het stop icon (F8) om de game te stoppen, zodat we weer veranderingen kunnen maken

Je zou moeten zien in de console onderaan:

```
Enemy spawned!  
Bacterie
```

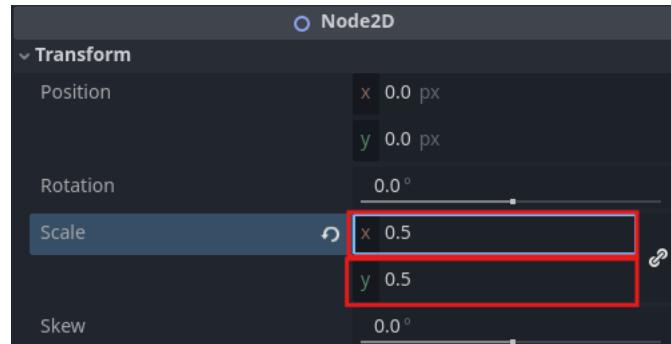
Belangrijk: Indentatie!

GDScript gebruikt **tabs** om te weten wat bij een functie hoort. Vergeet deze niet!

```
func _ready():  
    print("Dit hoort bij ready") # ✓ Correct  
print("Dit hoort NIET bij ready") # X Fout (geen tab)
```

optioneel als je de groote van de enemy niet goed vind:

1. ga naar je enemy scene
2. Klik op de enemy node
3. zoek in het inspector gedeelte naar transform onder Node2D en klap deze uit
4. zet Scale x en y naar de gewilde scale, ik heb hem persoonlijk op 0.5, zorg dat x en y dezelfde waarde hebben.



6. De _process functie - Continu updates

Wat is _process?

Deze functie wordt **elke frame** aangeroepen. Dat kan wel van 60 tot 240 keer per seconde verschillen (afhankelijk van je computer)!

Ga terug naar het script (klik op de **Script** tab bovenaan, of dubbelklik op enemy.gd in de FileSystem)

Voeg deze functie toe onder je _ready functie, zorg dat de func altijd helemaal aan de linkerkant van het script start (**geen tabs**):

```
func _process(delta: float) -> void:  
    position.x += speed
```

Test het!

Start de game. De enemy beweegt nu naar rechts!

Probleem: Hij beweegt **super snel!** Waarom?

- Elke frame wordt speed (60) opgeteld bij position.x
- Op 60 FPS = $60 \times 60 = 3600$ pixels per seconde!
- Op 240 FPS = $240 \times 60 = 14400$ pixels per seconde! (4x sneller op schnellere computers!)

De oplossing: Delta Time

Wat is delta?

Delta is de **tijd tussen frames** (in seconden). Meestal ~0.016 seconden (bij 60 FPS).

Door * delta te doen, normaliseer je de beweging naar **per seconde** in plaats van **per frame**.

Verander je code naar:

```
func _process(delta: float) -> void:  
    position.x += speed * delta
```

Nu betekent speed = 60: "60 pixels per seconde"

Test het! De enemy beweegt nu even snel op elke computer! 🎉

7. Data types - Variabelen vastzetten (Theorie + Praktijk)

Het probleem:

Je kunt per ongeluk dit doen:

```
var speed = 60  
speed = "I am speed" # Dit mag, maar geeft later problemen!
```

Speed is nu tekst in plaats van een getal. Als je later probeert position.x += speed * delta, crashed je game!

De oplossing: Data types vastzetten

```
var speed: int = 60  
var enemy_name: String = "Bacterie"  
var is_alive: bool = true
```

Uitleg:

- : int betekent "dit mag alleen een heel getal zijn"
- : String betekent "dit mag alleen tekst zijn"
- : bool betekent "dit mag alleen true of false zijn"
- : float betekent "dit mag een getal met komma zijn" (bijvoorbeeld 3.14)

Nu kan je **niet** meer per ongeluk doen:

```
var speed: int = 60  
speed = "I am speed" # ERROR! Dit mag niet!
```

Best practice: Zet altijd het type vast voor veilige code!

Interacties met andere nodes

1. Referenties naar andere nodes - AnimatedSprite

We willen in onze code de animatie kunnen aansturen. Daarvoor hebben we een **referentie** naar de AnimatedSprite2D node nodig.

Voeg deze variabele toe bovenaan de andere variables:

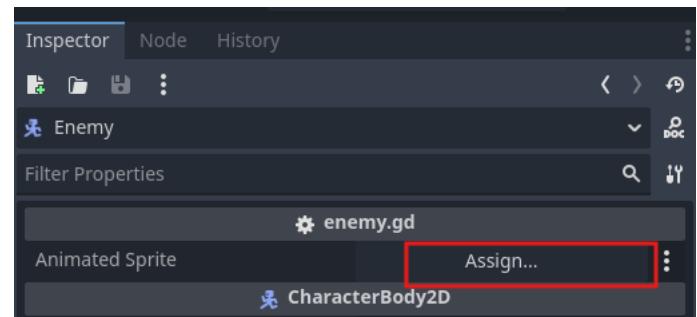
```
@export var animated_sprite: AnimatedSprite2D
```

Wat is @export?

Dit betekent: "Ik wil deze variabele kunnen instellen in de Godot editor (Inspector)."

De variabele verbinden:

1. **Sla je script op** (Ctrl + S)
2. Ga terug naar de **Enemy scene**
3. Selecteer de **Enemy** node (root)
4. Kijk in het **Inspector** paneel rechts
5. Je ziet nu een veld: **Animated Sprite**
6. **Sleep** de AnimatedSprite2D node vanuit de Scene Tree naar dit veld



Nu is de variabele verbonden! Je kunt nu in je code dingen doen met de sprite.

Test het:

Verander je `_ready` functie naar:

```
func _ready():
    animated_sprite.play("idle")
    print("Enemy spawned!")
```

Start de game. De idle animatie speelt nu! (Zou al moeten spelen door Autoplay, maar nu doe je het via code). Je kan dus als je referenties heb naar nodes specifieke functies erop uitvoeren, zoals bij `animated_sprite` de `play` functie! Kijk maar eens als je een punt achter de `animated_sprite` zet, dan geeft de auto complete honderden functies of variables die je zou kunnen gebruiken. Niet alle functies/variables zijn alleen voor de `AnimatedSprite2D`, er is ook veel overlap. Dit komt omdat `AnimatedSprite2D` ook een `Node2D` is, dus hij heeft alle functies/variables van `Node2D` ook bijvoorbeeld (zoals positie).

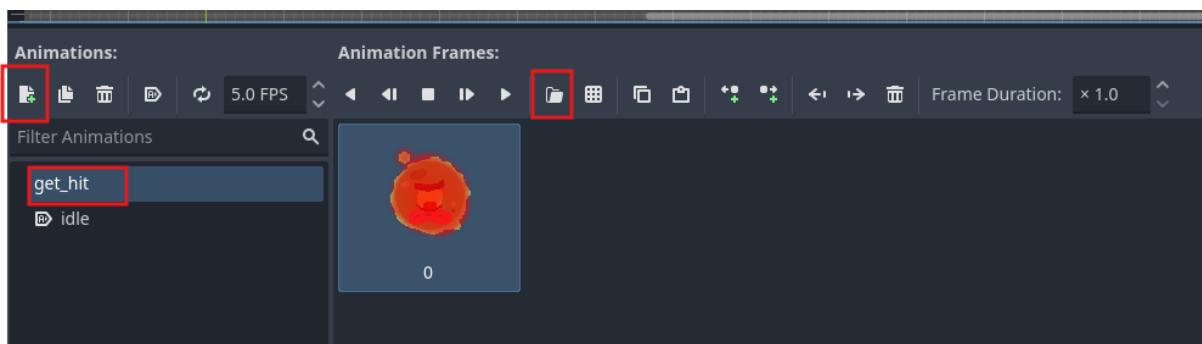
Pro tip: Als je op de "AnimatedSprite2D" gedeelte klikt met ctrl vast, dan ga je naar de documentatie hierover van Godot. Je kan heel veel leren van deze documentatie, elke Godot functie, data type, enzovoort, hebben documentatie.

2. Een tweede animatie - get_hit

We gaan een "get_hit" animatie toevoegen voor als de enemy schade krijgt.

In de Godot editor:

1. Selecteer de **AnimatedSprite2D** node
2. Open het **Sprite Frames** paneel (onderkant scherm)
3. Klik op "Add Animation" (document met plusje)
4. Noem het **get_hit**
5. Omdat deze "animatie" maar 1 frame heeft, hoeft je niet via sprite frames te gaan maar degene ernaast klik op "**Add frame from file**"
6. Kies assets/sprites/enemy/get_hit.png
7. Klik op Loop om hem **uit** te zetten (de hit animatie speelt maar 1 keer)



Test het in code:

Verander je _ready functie naar:

```
func _ready():
    animated_sprite.play("get_hit")
```

Start de game. Je ziet nu de get_hit animatie afspeelt.

3. Signals verbinden – Na get_hit idle

Je merkte misschien dat na de animatie geen animatie meer speelt voor de enemy, dit komt omdat autoplay alleen aan het begin van de game werkt, dus als je de animatie veranderd wordt dit de standaard. Je zou misschien denken dat we in de ready gelijk weer idle kunnen spelen zoals dit:

```
func _ready():
    animated_sprite.play("get_hit") #Wordt meteen gestopt door idle
    animated_sprite.play("idle")
```

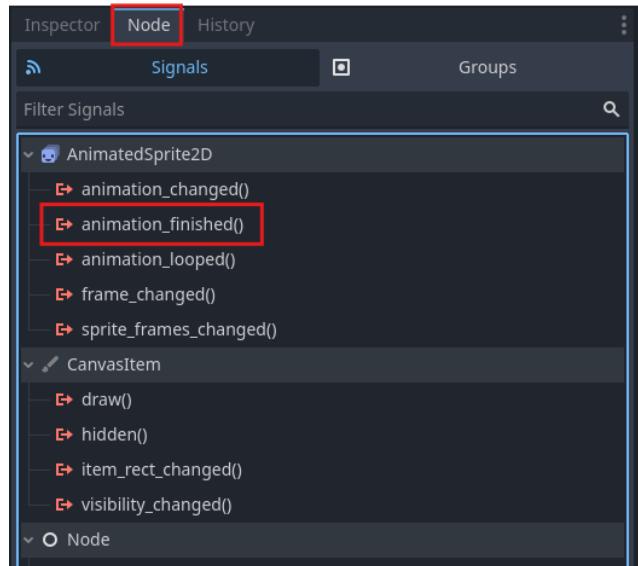
Maar nu verander je meteen het naar idle en laat je get_hit niet eerst afspeelen. Hiervoor kunnen we een signal gebruiken!

Wat is een signal?

Een signal is een **notificatie** die Godot stuurt wanneer iets gebeurt. Bijvoorbeeld: "Mijn animatie is klaar!". Elke node heeft verschillende signals die af kunnen sturen

Signals verbinden voor de animatie:

1. Selecteer de **AnimatedSprite2D** node
2. Klik op het **Node** tabblad rechts (naast Inspector)
3. Je ziet een lijst met signals
4. Dubbelklik op **animation_finished()**
5. Zorg dat **Enemy** geselecteerd is als receiver (gehighlight)
6. Ik verander meestal de receiver method iets makkelijker leesbaarder, bijvoorbeeld: "
_on_animation_finished"
7. Klik **Connect**



Nu komt er als het goed is aan de onderkant van je enemy script deze functie met een groen icoontje ernaast:

```
func _on_animation_finished() -> void:  
    pass # Replace with function body.
```

Nu word deze functie aangesproken als een animatie klaar is met spelen, we willen nu dat:

```
func _on_animation_finished() -> void:  
    animated_sprite.play("idle")
```

Test het, en zie dat de idle animatie speelt na de get hit functie

4. GameManager referentie - Het hart vinden

De enemy moet weten waar het hart is om er naartoe te kunnen lopen. Het hart zit in de **GameManager** een node die ik heb gemaakt om de game te beheren.

Voeg deze variabele toe in je enemy:

```
var game_manager: Node
```

Het hart vinden in **_ready**:

```
func _ready():  
    game_manager =  
get_tree().root.get_node("Main").get_node("GameManager")
```

Uitleg:

- `get_tree()` = de hele game
- `.root` = het hoofdniveau
- `.get_node("Main")` = zoek de Main scene
- `.get_node("GameManager")` = zoek de GameManager node daaronder

Check of het werkt:

```
func _ready():
    game_manager =
get_tree().root.get_node("Main").get_node("GameManager")
    print("GameManager found: ", game_manager)
    print("Heart found: ", game_manager.heart_instance)
```

Start de game. In de console zou je moeten zien dat beide gevonden zijn!

5. If statements - Keuzes maken (Theorie)

Tot nu toe voert je code altijd alles uit, van boven naar beneden. Maar soms wil je dat code alleen uitgevoerd wordt **als** aan een bepaalde voorwaarde wordt voldaan.

Voorbeeld uit het echte leven:

ALS het regent

 THEN pak een paraplu

In code:

```
var is_raining = true

if is_raining:
    print("Pak een paraplu!")
```

Hoe werkt een if statement?

1. Godot checkt of de voorwaarde waar is (true)
2. Als het waar is, voert hij de code uit die ingesprongen staat
3. Als het niet waar is (false), slaat hij die code over

Vergelijkingen:

Je kunt verschillende dingen checken:

```
# Is gelijk aan
if health == 0:
    print("Dood!")

# Is NIET gelijk aan
if health != 0:
    print("Nog leven!")

# Groter dan
if health > 5:
    print("Veel health!")

# Kleiner dan
if health < 2:
    print("Bijna dood!")

# Kleiner dan of gelijk aan
if health <= 0:
```

```

        Print ("Zeker weten dood, dit is een veiligere check!")

# Groter dan of gelijk aan
If health >= 5:
    Print ("Veel health, 5 telt nu ook mee!")

# En (beide moeten waar zijn)
if health > 0 and is_alive:
    print("Leeft nog!")

# Of (een van beide moet waar zijn)
if health == 0 or is_poisoned:
    print("Probleem!")

# Niet (omdraaien van true/false)
if not is_alive:
    print("Dood!")

```

Test het zelf:

Voeg dit toe in je _ready functie:

```

func _ready():
    game_manager =
get_tree().root.get_node("Main").get_node("GameManager")

    # Test if statements
    var test_health = 3

    if test_health > 5:
        print("Veel health!")

    if test_health < 5:
        print("Weinig health!")

    if test_health == 3:
        print("Precies 3 health!")

```

Start de game en kijk in de console. Welke berichten zie je? Waarom?

6. Vectors - Posities en richtingen (Theorie)

Wat is een Vector2?

Een Vector2 is een **punt in 2D ruimte** of een **richting**. Het heeft twee getallen:

- **x**: horizontaal (links-rechts)
- **y**: verticaal (boven-onder)

Voorbeelden:

```

# Een positie op het scherm
var hart_positie = Vector2(100, 50)  # x=100, y=50

# Een richting
var naar_rechts = Vector2(1, 0)  # 1 naar rechts, 0 omhoog/omlaag

```

```

var naar_boven = Vector2(0, -1) # 0 links/rechts, -1 omhoog (min is omhoog!)
var schuin = Vector2(1, 1) # naar rechtsonder

```

Posities in Godot:

Elke node heeft een position of global_position, position is relatief aan de parent node en global position is de absolute positie in de wereld, over het algemeen beter om global_postion te gebruiken:

```

print(global_position) # Bijvoorbeeld: (150, 200)
print(global_position.x) # 150
print(global_position.y) # 200

```

Vectoren optellen en aftrekken:

```

var pos_a = Vector2(100, 50)
var pos_b = Vector2(150, 80)

# Verschil berekenen (richting van A naar B)
var richting = pos_b - pos_a
print(richting) # (50, 30)

```

Dit betekent: "Om van A naar B te komen, moet je 50 naar rechts en 30 naar beneden."

Test het:

```

func _ready():
    game_manager =
get_tree().root.get_node("Main").get_node("GameManager")

    # Test vectors
    var enemy_positie = global_position
    var hart_positie = game_manager.heart_instance.global_position

    print("Enemy is hier: ", enemy_positie)
    print("Hart is hier: ", hart_positie)

    var richting_naar_hart = hart_positie - enemy_positie
    print("Richting naar hart: ", richting_naar_hart)

```

7. Beweging naar het hart - Stap 1: Simpel

Nu combineren we if statements en vectors om naar het hart te bewegen! Het is altijd handig om te checken of je referentie variables bestaan in je code voordat we hem gebruiken, dit is handig om te doen voor je GameManager. Dus if game_manager, gebeurt alleen als er een game_manager is, het kan natuurlijk ook andersom met if not game_manger, gebeurt alleen als de game_manager **niet** bestaat.

Update je _process functie:

```

func _process(delta: float) -> void:
    # Check of het hart bestaat (if statement!)
    if not game_manager:
        print("Geen game manager!")
        return # Stop de functie hier

```

```

        if not game_manager.heart_instance:
            print("Geen hart!")
            return

        # Waar is het hart? (Vector2!)
        var hart_positie: Vector2 =
game_manager.heart_instance.global_position
        print("Hart positie: ", hart_positie)

        # Waar is deze enemy?
        var enemy_positie: Vector2 = global_position
        print("Enemy positie: ", enemy_positie)
        position.x += speed * delta

```

Test het!

Start de game. Je ziet nu constant beide posities in de console! Maar beweegt nog alleen maar naar rechts.

Uitleg:

- if not game_manager: = "als game_manager NIET bestaat"
- return = "stop deze functie hier, voer de rest niet uit"
- Dit voorkomt crashes als iets niet bestaat

8. Beweging naar het hart - Stap 2: Richting berekenen

Nu gaan we de richting naar het hart berekenen.

Update je _process functie:

```

func _process(delta: float) -> void:
    # Check of het hart bestaat
    if not game_manager or not game_manager.heart_instance:
        return

    # Posities ophalen
    var hart_positie: Vector2 =
game_manager.heart_instance.global_position
    var enemy_positie: Vector2 = global_position

    # Richting berekenen (vectoren aftrekken!)
    var richting: Vector2 = hart_positie - enemy_positie

    print("Richting naar hart: ", richting)

    position.x += speed * delta

```

Test het!

In de console zie je nu bijvoorbeeld: (50, -30)

Dit betekent: "Het hart is 50 pixels naar rechts en 30 pixels omhoog"

Je merkt dat de x steeds kleiner word als de enemy dichter bij het hart komt!

9. Beweging naar het hart - Stap 3: Bewegen (te snel!)

Nu gaan we daadwerkelijk bewegen in die richting.

Update je `_process` functie:

```
func _process(delta: float) -> void:  
    # Check of het hart bestaat  
    if not game_manager or not game_manager.heart_instance:  
        return  
  
    # Richting berekenen  
    var hart_positie: Vector2 =  
game_manager.heart_instance.global_position  
    var direction: Vector2 = hart_positie - global_position  
  
    # Beweeg in die richting  
    position += direction * speed * delta
```

Test het!

De enemy beweegt naar het hart... maar aan het begin veel te snel!

Waarom?

Als richting bijvoorbeeld (100, 50) is, dan betekent dat:

- Beweeg $100 * \text{speed}$ pixels per seconde naar rechts
- Beweeg $50 * \text{speed}$ pixels per seconde omhoog

Maar we willen juist dat de **totale snelheid** altijd 60 pixels per seconde is, ongeacht hoe ver het hart is!

10. Normalized direction - Constante snelheid

We willen dat de enemy altijd even snel beweegt, ongeacht hoe ver het hart is.

De oplossing: `normalized()`

Een normalized vector heeft altijd een **lengte van 1**. Daarna vermenigvuldigen we met onze gewenste speed.

```
func _process(delta: float) -> void:  
    # Check of het hart bestaat  
    if not game_manager or not game_manager.heart_instance:  
        return  
  
    # Bereken richting naar het hart  
    var target_pos: Vector2 =  
game_manager.heart_instance.global_position  
    var direction: Vector2 = (target_pos - global_position)  
  
    # Als we al bij het hart zijn, stop met bewegen  
    if direction.length_squared() < 1.0:  
        return  
  
    # Maak de richting normalized (lengte = 1)
```

```
direction = direction.normalized()

# Beweeg met onze speed
position += direction * speed * delta
```

Naast dat we de snelheid normalizeren, checken nu ook of de enemy dicht genoeg bij het hart is, dit doen we door te kijken of length_squared() klein genoeg is length_squared() is sneller dan length() en checkt hier of de afstand heel klein is,

Test het!

Nu beweegt de enemy met een constante snelheid van 60 pixels per seconde naar het hart!

Bonus: Sprite flip

Voeg dit toe aan de onderkant van _process om de enemy de juiste kant op te laten kijken. Bij deze sprite zie je niet erg veel verschil omdat hij rond is, maar is wel handig om te weten voor andere sprites:

```
# Flip de sprite als we naar links bewegen
if animated_sprite:
    var should_flip_sprite: bool = direction.x < 0
    animated_sprite.flip_h = should_flip_sprite
```

flip_h staat voor flip horizontally en dat doen we als should_flip true is, dus direction.x kleiner is dan 0, dat is zo als hij naar links gaat!

Collisions

1. Collision met de muren

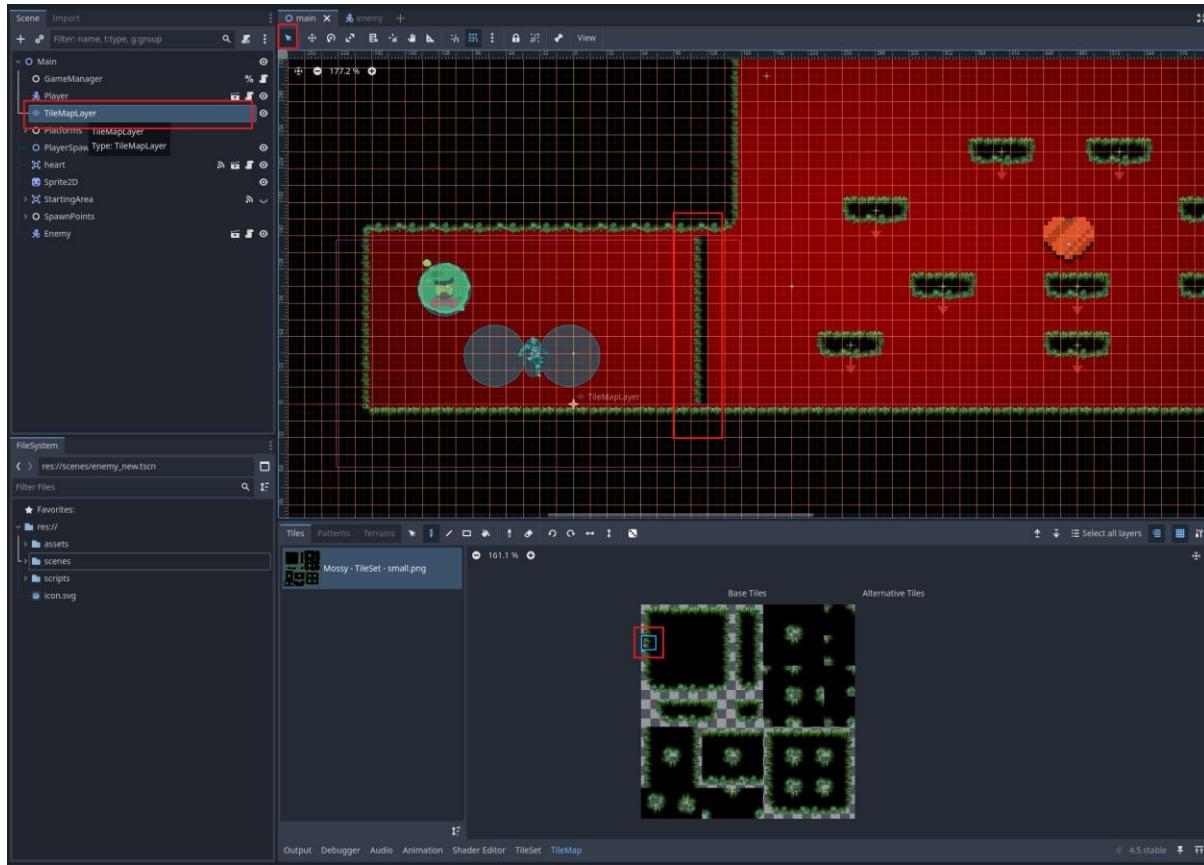
De enemy die we tot nu toe hebben gemaakt gaat door alles heen, misschien heb je dat nog niet erg gemerkt, maar dit is wel handig om nu wat mee te doen!

Test maar door een muur te maken, ik gebruik een TileMapLayer om de map te maken

1. ga naar main scene
2. klik op tileMapLayer, nu zie je een gedeelte onderaan met allemaal verschillende sprites die je als muur kan gebruiken
3. Klik op een tile.
4. Sleep of klik op plekken in het level dan komen tiles waarvan je een muur kan maken.

Als het niet lukt om de geselecteerde tile te plaatsen dan is het probleem misschien dat je in de verkeerde modus zit, zorg ervoor dat je in de select modus zit, cursor icon links boven, als je tiles wil plaatsen

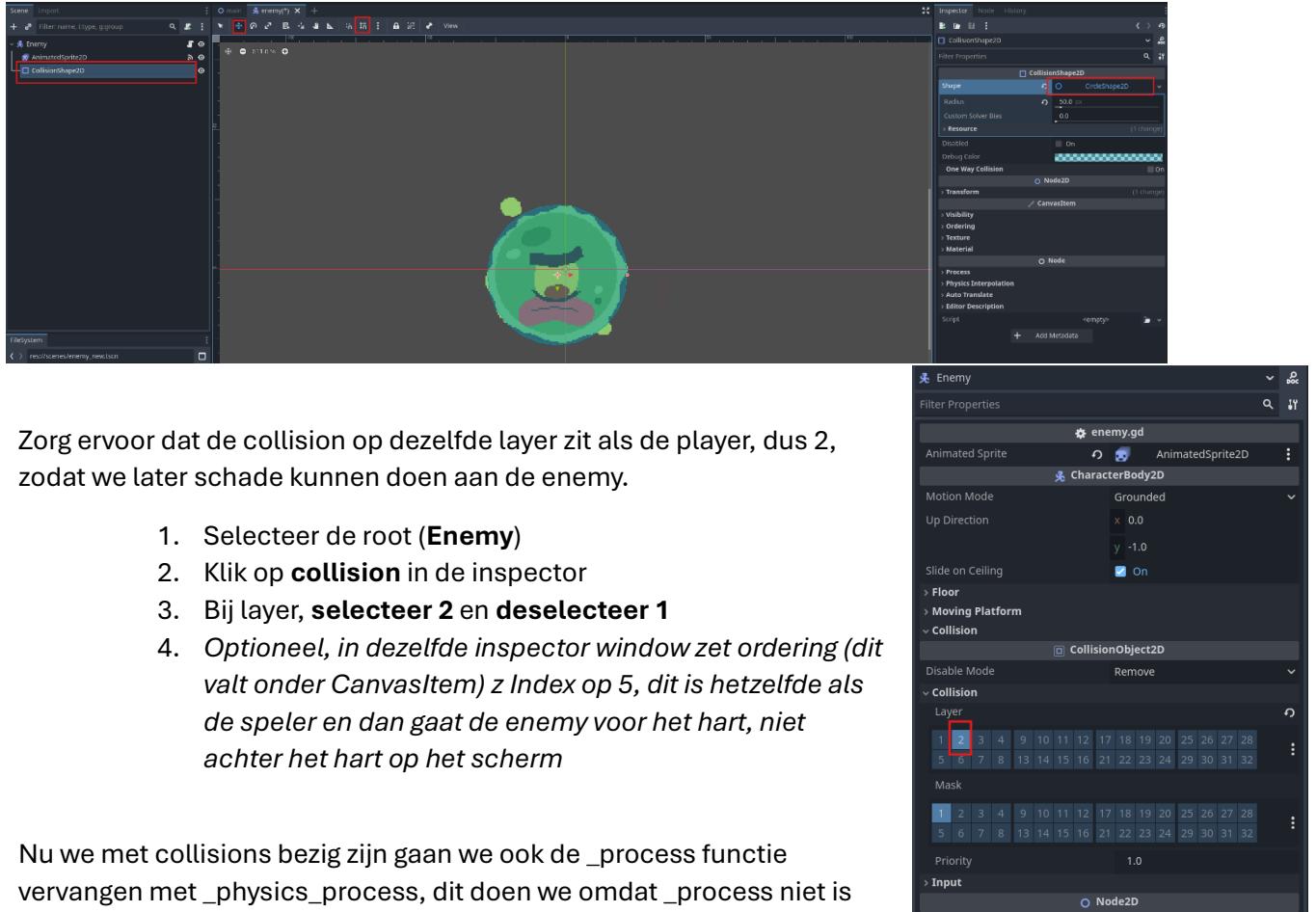
Test en zie dat de enemy recht door de muur gaat, terwijl de player er niet door gaat



Dit komt omdat de Enemy nog geen collision shape heeft, je had misschien al gemerkt dat je enemy een warning gaf "this node has no shape", dit gaan we nu oplossen!

1. Ga naar je enemy scene
2. Selecteer de root node (Enemy)
3. Add node Collision shape 2d

4. In de inspector bij shape, klik op <empty> -> CircleShape2D
5. Nu is er een kleine cirkel die de collision bepaalt, dus die willen we wat groter maken, sleep het oranje balletje aan de zijde van het cirkeltje en maak hem zo groot als je wilt.
6. Als je de collision shape ook wat wil verplaatsen kan dat door w te drukken en dan ben je in de move stand en dan kan je hem slepen zoals je wilt (Zorg ervoor dat grid snap uit staat om precieze aanpassingen te kunnen maken, icoontje bovenaan gehighlight in het plaatje hieronder, moet grijs zijn)



Zorg ervoor dat de collision op dezelfde layer zit als de player, dus 2, zodat we later schade kunnen doen aan de enemy.

1. Selecteer de root (**Enemy**)
2. Klik op **collision** in de inspector
3. Bij layer, **selecteer 2 en deselecteer 1**
4. Optioneel, in dezelfde inspector window zet ordering (dit valt onder *CanvasItem*) z Index op 5, dit is hetzelfde als de speler en dan gaat de enemy voor het hart, niet achter het hart op het scherm

Nu we met collisions bezig zijn gaan we ook de `_process` functie vervangen met `_physics_process`, dit doen we omdat `_process` niet is gemaakt voor collisions en `_physics_process` wel, dus vervang `_process` met dit:

```
func _physics_process(delta: float) -> void:
    # Check of het hart bestaat
    if not game_manager or not game_manager.heart_instance:
        return

    # Bereken richting naar het hart
    var target_pos: Vector2 =
game_manager.heart_instance.global_position
    var direction: Vector2 = (target_pos - global_position)

    # Als we al bij het hart zijn, stop met bewegen
    if direction.length_squared() < 1.0:
        return

    # Maak de richting normalized (lengte = 1)
```

```

direction = direction.normalized()

# Beweeg met onze speed
position += direction * speed * delta

# Flip de sprite als we naar links bewegen
if animated_sprite:
    var should_flip_sprite: bool = direction.x < 0
    animated_sprite.flip_h = should_flip_sprite

move_and_slide()

```

`move_and_slide()` zorgt ervoor dat alle bewegingen goed aan het einde worden doorgevoerd en dat de enemy niet door de muren heen gaat.

Test de game en zie nu dat de enemy niet door de muur heen kan! Je kan de muur verwijderen door weer de tilemap te selecteren en dan rechter muisknop te drukken op de tiles die je weg wil halen, herinner dat je in de select modus moet zitten voor dit!

2. Enemy schade van player

We kunnen nu nog niet heel veel doen aan de enemy toch? Dus wil ik nu gaan maken dat de enemy schade kan krijgen van de speler.

In dit project heb ik al best wat functionaliteit gemaakt die je kan gebruiken, het lijkt me nu een goed punt om te gaan kijken naar de rest van het project.

Links onder zie je de folders staan van de game, onder scenes staat `player.tscn` dubbel klik erop, je ziet dat de speler 2 hitboxen heeft, `HitboxRight` en `HitboxLeft`. Als je op één van de hitboxen klikt zie je dat ze het script hebben `hitbox.gd` klik daar op.

Hitbox is best een kort script met veel uitleg, dus neem je tijd om dit even door te lezen, maar de TLDR is als er een body in deze hitbox komt (zoals de `CharacterBody2D` van onze Enemy) dan gaat hij kijken of die body de `get_hit` functie heeft en of die niet zichzelf aanvalt. Als dat zo is word de `get_hit` functie op de body aangesproken

Uit deze informatie kunnen we afleiden dat als we een `get_hit` functie maken in onze enemy dan word deze aangesproken door de hitbox van de speler.

Voeg in het enemy script dit variabel toe bovenaan:

```
var health: int = 3
```

Voeg deze functie toe:

```

func get_hit(dmg: int, _attacker_pos) -> void:
    # Verminder health
    health -= dmg

    print("Enemy got hit for ", dmg, " damage, and has ", health,
" health left!")

    # Als dood, verwijder de enemy
    if health <= 0:
        print("Enemy died!")
        queue_free()

```

`_attacker_pos` kan je negeren, het moet er wel zijn omdat de hitbox attacker position meegeeft, maar de enemy heeft hem niet nodig, als je een `_` voor de parameter zet dan zeg je tegen godot dat je hem niet gaat gebruiken en dan geeft hij geen warning. `queue_free()` is een functie die je kan gebruiken om jezelf (de enemy waar dit script bij hoort) te verwijderen van de huidige game, daarna is de enemy weg en werkt niks meer uit dit script.

Test het!

Start de game. Val de enemy aan met de speler. De enemy verdwijnt na 3 hits! 

3. Knockback toevoegen

Nu gaan we de enemy wegduwen als hij geraakt wordt (knockback), zodat we kunnen stoppen de enemy naar het hart gaan.

Voeg deze variables/constants toe:

```
const KNOCKBACK_X = 100
const KNOCKBACK_Y = -50

var is_stunned: bool = false
```

Constants (const) zijn eigenlijk hetzelfde als variables alleen kan je ze op andere plekken in je code niet meer aanpassen, dus bijvoorbeeld hier weten we dat we de knockback altijd constant willen hebben, dus geven we aan hoeveel het moet zijn 1 keer, en dan voor de rest kunnen we het alleen maar uitlezen.

Update de `get_hit` functie:

```
func get_hit(dmg: int, _attacker_pos) -> void:
    # Verminder health
    health -= dmg

    print("Enemy got hit for ", dmg, " damage, and has ", health,
" health left!")

    # Bereken knockback richting (weg van het hart)
    var direction = (global_position -
game_manager.heart_instance.position).normalized()
    velocity = direction * KNOCKBACK_X
    velocity.y = KNOCKBACK_Y

    # Zet in stun state
    is_stunned = true

    # Als dood, verwijder de enemy
    if health <= 0:
        queue_free()
    else:
        animated_sprite.play("get_hit")
```

In dit stukje code word dus bepaalt wat de andere kant op is als het hart, voor de enemy. En dan geeft hij een specifieke velocity en de velocity word behandeld in de `_physics_process` functie door middel van `move_and_slide()`. We hebben ook meteen de eerder gemaakte `get_hit` animatie gebruikt.

Update je `_physics_process` functie (helemaal bovenaan toevoegen):

```

func _physics_process(delta: float) -> void:
    # Als gestunned, beweeg alleen met knockback velocity
    if is_stunned:
        # Apply friction to slow down the knockback
        var friction = 100.0 # Hoe hoger, hoe sneller hij stopt
        velocity = velocity.move_toward(Vector2.ZERO, friction *
delta)
        # Continue moving with knockback velocity
        move_and_slide()

        # If the enemy has almost stopped moving, end the stun
        if velocity.length() < 15:
            is_stunned = false
        return # Skip all other logic while stunned
    # ... rest van je code

```

Dit zorgt er dus voor dat de enemy niks meer doet als hij stunned is, behalve de knockback die die heeft gekregen uitvoert en als de velocity weer klein is, dus genoeg knockback heeft gekregen dan is de enemy niet meer stunned. De velocity word steeds verlaagd met friction die aangewezen word

Test het!

Start de game. Val de enemy aan. Hij wordt nu weggeduwd! 

4. Een hitbox toevoegen aan de enemy

Het is nu nog een beetje makkelijk hé, de enemy kan nog niks tegen doen, we willen dus dat de enemy de speler kan aanvallen als de speler de enemy aanraakt, zodat ze niet door elkaar heen gaan. Maar ik had al voor de player zoals jullie hebben gezien functionaliteit gemaakt om damage te geven en te krijgen! Een belangrijk concept van coderen is herbruikbaarheid, dat gaan we nu toepassen! De hitbox die we eerder hebben bekeken gaan we nu ook toevoegen aan de enemy

1. Klik op instantiate child scene
2. Klik op Hitbox
3. Nu heeft de enemy een hitbox, alleen je ziet de warning al, de hitbox heeft een specifieke collision shape nodig
4. Selecteer de hitbox en druk ctrl + a
5. Voeg een **CollisionShape2D** toe als kind van Hitbox
6. Selecteer de CollisionShape2D
7. Kies in het Inspector bij **Shape: New CircleShape2D**
8. Pas de grootte aan zodat het de enemy omsluit (sleep de rode cirkel en verplaats eventueel)

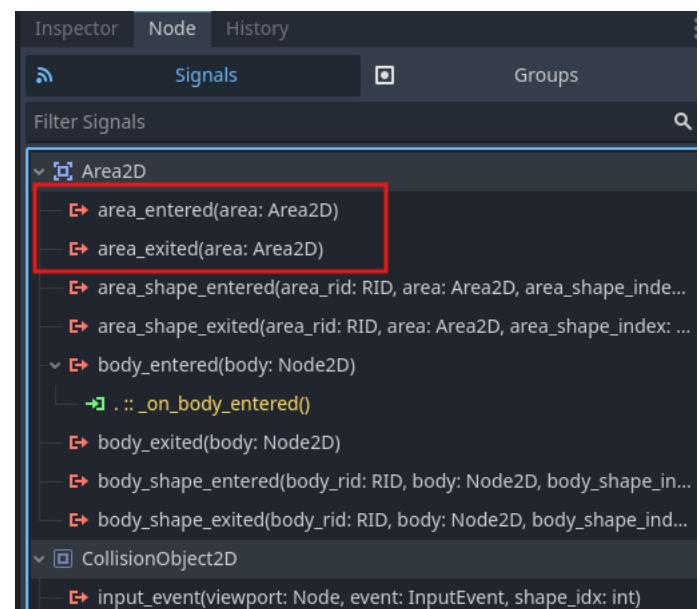
Test het uit, als het goed is werkt alles goed en kan je speler damage krijgen van de enemy, dit ging erg snel toch! Omdat we de eerder gemaakte hitbox konden hergebruiken en omdat de speler al de get_hit functie heeft.

5. Heart collision signal

We kunnen dezelfde hitbox nu gebruiken om schade te doen aan het hart! Het enige probleem is dat het hart een Area2D is geen CharacterBody, dus deze word niet aangesproken vanuit de hitboxes zijn `_on_body_entered` functie. We gaan hier een ander signal voor gebruiken, `_on_area_entered`.

Signal verbinden:

1. Selecteer de **Hitbox** node van de enemy
2. Klik op het **Node** tabblad rechts (naast Inspector)
3. Je ziet een lijst met signals
4. Dubbelklik op **area_entered**
5. Zorg dat **Enemy** geselecteerd is als receiver
6. Klik **Connect**



Godot maakt nu automatisch een functie aan in je script:

```
func _on_hitbox_area_entered(area: Area2D) -> void:  
    pass # Replace with function body.
```

Doe hetzelfde voor **area_exited**.

6. Heart attack logica

Nu gaan we de enemy het hart laten aanvallen!

Voeg deze variabelen toe bovenaan:

```
const DAMAGE = 1  
@export var attack_cooldown: float = 3.0  
var is_attacking_heart = false  
var _attack_timer: float = 0.0
```

Update je collision functies:

```
func _on_hitbox_area_entered(area: Area2D) -> void:  
    # Check of we het hart raken  
    if area == game_manager.heart_instance:  
        is_attacking_heart = true  
        print("Attacking heart!")  
  
        # Direct damage doen  
        if _attack_timer <= 0.0:
```

```

        area.call_deferred("take_damage", DAMAGE)
        _attack_timer = attack_cooldown

func _on_hitbox_area_exited(area: Area2D) -> void:
    # Check of we het hart verlaten
    if area == game_manager.heart_instance:
        is_attacking_heart = false
        print("Stopped attacking heart")

```

Update je `_physics_process` functie (bovenaan toevoegen, voor de movement code)

```

func _physics_process(delta: float) -> void:
    # Als gestunned, beweeg alleen met knockback velocity
    if is_stunned:
        # Wrijving toepassen om de terugslag te vertragen
        var friction = 100.0 # Hoe hoger, hoe sneller hij stopt
        velocity = velocity.move_toward(Vector2.ZERO, friction * delta)
        # Blijf bewegen met de terugslag-snelheid
        move_and_slide()

        # Als de vijand bijna is gestopt met bewegen, beëindig de
        stun
        if velocity.length() < 15:
            is_stunned = false
            return # Sla alle andere logica over terwijl hij
        gestunned is
        # Tel de attack timer af
        _attack_timer = max(0.0, _attack_timer - delta)

        # Als we het hart aanvallen en cooldown voorbij is
        if is_attacking_heart and _attack_timer <= 0:
            game_manager.heart_instance.call_deferred("take_damage",
            DAMAGE)
            _attack_timer = attack_cooldown

    # ... rest van je code (movement)

```

Test het!

Start de game. De enemy loopt naar het hart en valt het aan elke 3 seconde! Je ziet de health van het hart dalen in de console.

Alle code samen - Finale versie

Je hebt nu een volledige werkende enemy! Hier is de complete code:

```

extends CharacterBody2D

# Stats
var speed: float = 80.0
var health: int = 3
var attack_cooldown: float = 3.0

# Constanten
const DAMAGE = 1

```

```

const KNOCKBACK_X = 100
const KNOCKBACK_Y = -50

# Referenties
@export var animated_sprite: AnimatedSprite2D
var game_manager: Node

# State
var is_stunned: bool = false
var is_attacking_heart: bool = false
var _attack_timer: float = 0.0

func _ready():
    game_manager =
get_tree().root.get_node("Main").get_node("GameManager")

func _physics_process(delta: float) -> void:
    # Als gestunned, beweeg alleen met knockback velocity
    if is_stunned:
        # Wrijving toepassen om de terugslag te vertragen
        var friction = 100.0 # Hoe hoger, hoe sneller hij stopt
        velocity = velocity.move_toward(Vector2.ZERO, friction *
delta)
        # Blijf bewegen met de terugslag-snelheid
        move_and_slide()

        # Als de vijand bijna is gestopt met bewegen, beëindig de
stun
        if velocity.length() < 15:
            is_stunned = false
        return # Sla alle andere logica over terwijl hij
gestunned is

    # Tel de attack timer af
    _attack_timer = max(0.0, _attack_timer - delta)

    # Als we het hart aanvallen en cooldown voorbij is
    if is_attacking_heart and _attack_timer <= 0:
        game_manager.heart_instance.call_deferred("take_damage",
DAMAGE)
        _attack_timer = attack_cooldown

    # Check of het hart bestaat
    if not game_manager or not game_manager.heart_instance:
        return

    # Bereken richting naar het hart
    var target_pos: Vector2 =
game_manager.heart_instance.global_position
    var direction: Vector2 = (target_pos - global_position)

    # Als we al bij het hart zijn, stop met bewegen
    if direction.length_squared() < 1.0:
        return

    # Maak de richting normalized (lengte = 1)

```

```

        direction = direction.normalized()

        # Beweeg met onze speed
        position += direction * speed * delta

        # Flip de sprite als we naar links bewegen
        if animated_sprite:
            var should_flip_sprite: bool = direction.x < 0
            animated_sprite.flip_h = should_flip_sprite

        move_and_slide()

func get_hit(dmg: int, _attacker_pos) -> void:
    # Verminder health
    health -= dmg

    print("Enemy got hit for ", dmg, " damage, and has ", health,
" health left!")

    # Bereken knockback richting (weg van het hart)
    var direction = (global_position -
game_manager.heart_instance.position).normalized()
    velocity = direction * KNOCKBACK_X
    velocity.y = KNOCKBACK_Y

    # Zet in stun state
    is_stunned = true

    # Als dood, verwijder de enemy
    if health <= 0:
        queue_free()
    else:
        animated_sprite.play("get_hit")

func _on_animation_finished() -> void:
    animated_sprite.play("idle")

func _on_hitbox_area_entered(area: Area2D) -> void:
    # Check of we het hart raken
    if area == game_manager.heart_instance:
        is_attacking_heart = true
        print("Attacking heart!")

        # Direct damage doen
        if _attack_timer <= 0.0:
            area.call_deferred("take_damage", DAMAGE)
            _attack_timer = attack_cooldown

func _on_hitbox_area_exited(area: Area2D) -> void:
    # Check of we het hart verlaten
    if area == game_manager.heart_instance:
        is_attacking_heart = false
        print("Stopped attacking heart")

```

Wat je nu hebt geleerd

- Wat nodes en scenes zijn
- Wat variabelen zijn en waarom ze handig zijn
- Wat functies zijn (_ready, _process, _physics_process)
- Wat delta time is en waarom je het gebruikt
- Data types en waarom je ze vastlegt
- Hoe je referenties maakt naar andere nodes
- Hoe je animaties aanstuurt via code
- Hoe je beweging naar een doel programmeert
- Wat normalized vectors zijn
- Wat signals zijn en hoe je ze verbindt
- Hoe je collision detectie gebruikt
- Hoe je scenes/scripts hergebruikt zoals bij hitbox
- Hoe je knockback implementeert

Gefeliciteerd! Je hebt een complete enemy gemaakt! 🎮

Extra uitdagingen

Makkelijk:

- Exporteer speed, health, attack_cooldown zodat je ze per enemy kunt aanpassen in de inspector
- Maak een eigen enemy met assets die je online vind, dus een nieuw scene maken met alle animatie instellen, enzovoort. Maar hetzelfde enemy script gebruiken, je kan deze enemy andere stats geven, zoals sneller, sterker, enzovoort
- Bij deze nieuwe enemy kan je extra animatie's toevoegen zoals attack animatie of ren animatie

Gemiddeld:

- Voeg een health bar toe boven de enemy, tip gebruik progress bar
- Voeg gewenste sound effects toe zoals geluid als je enemy damage krijgt of aanvalt
- Voeg een muziekje op de achtergrond toe
- Pas de map aan

Moeilijk:

- Spawn enemies in waves (golven), hiervoor kan je de game manager gebruiken, tip je kan scenes gebruiken in je code door soon soort code:

```
@export var enemy_scene: PackedScene  
get_tree().root.get_node("Main").add_child(enemy_scene.instantiate())
```

- Maak een boss enemy met meer health en speciale attacks

- Implementeer verschillende AI patterns (sommige rennen weg als low health)
- Maak een complete game loop ervan, door menu's toe te voegen zoals game over screen, retry knop, win screen, main menu