

GDD Franken Farm

Entrega Final

Vivi Arruda 2110624

Game Concept (GC)

Panorama

Animação procedural similar a *Rain World*. Jogos de plantação como *Stardew Valley*. Jogo que mistura a plantação como atividade relaxante em um momento e como jogo de horda em outro instante.

Descrição

E se um dia sua plantação se revoltasse contra você? Seja um fazendeiro/cientista maluco criando plantas-monstro para extrair os vegetais de maior qualidade e vendê-los no mercadinho orgânico mais próximo! Em Franken Farm você tem toda a calma de um jogo de fazenda misturado com a destruição e caos dos jogos de horda mais sanguinários.

Monte sua base de experimentos científicos de forma cautelosa para evitar a destruição causada por esses tubérculos e leguminosas demoníacas, quer dizer, monte sua fazendinha orgânica com todo carinho e amor :)

Características-Chave

- Animação procedural
- Alternância entre interação do jogo de forma pacífica com forma de combate (dia e noite)
- Tema de fazenda
- Tema de criação de monstros

Gênero

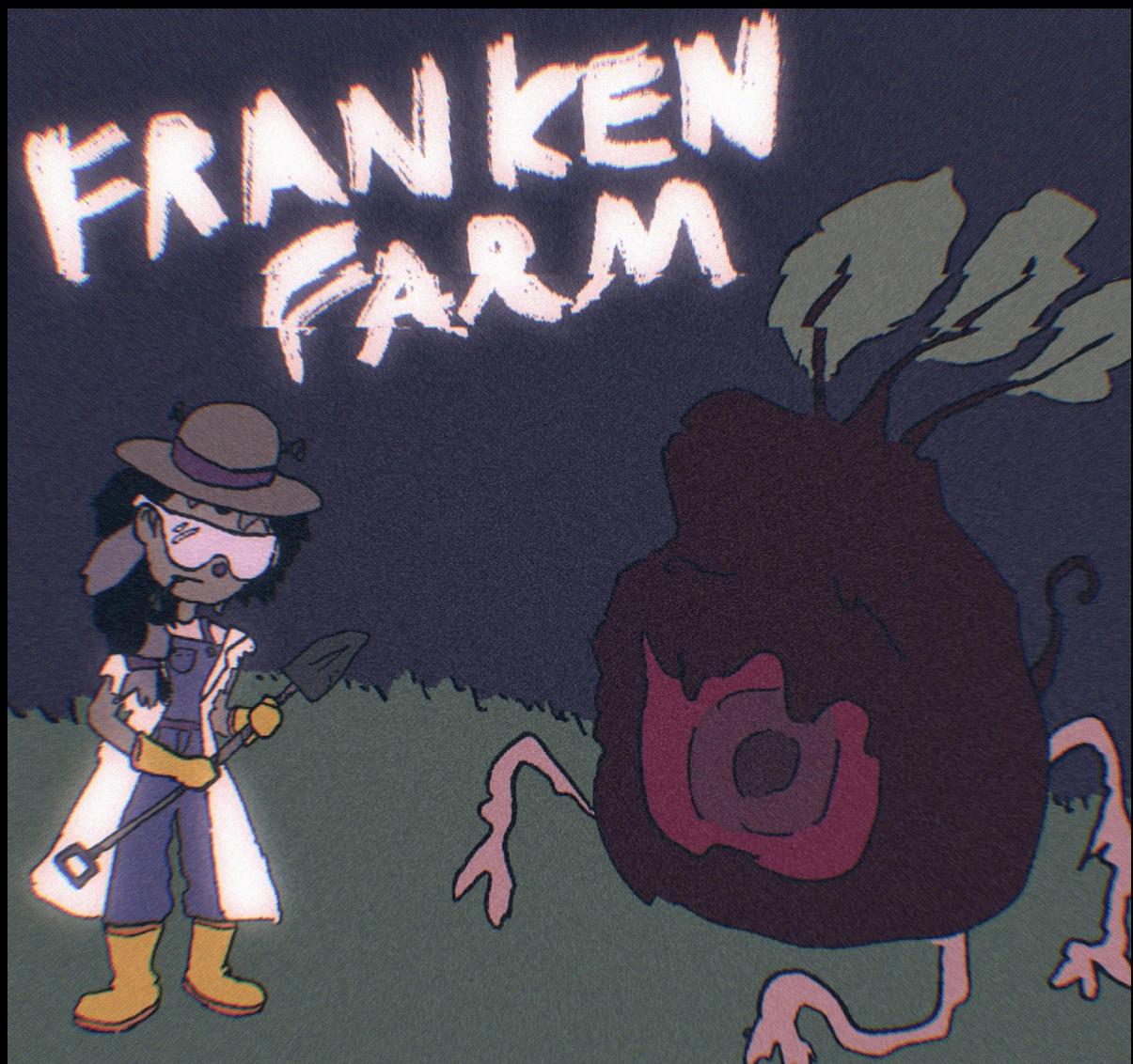
sandbox, plantação, estratégia, sobrevivência, horda

Plataforma

WebGL e Desktop

Artes de Conceito

- Concept Inicial



- Teste de Identidade Visual



- Aquarela



Game Functional Specification (GFS)

Mecânica do Jogo

Core Gameplay

O gameplay consiste em plantar, organizar, comprar, vender e fazer estratégias durante o dia e a noite, sobreviver aos ataques dos monstros.

Gameflow

O jogador faz o turno do dia se preparando para o turno da noite, e a noite ele sobrevive a horda que plantou no dia. O jogo acontece em 1 semana (7 dias) e com um boss na noite final. O objetivo do jogador é sobreviver essa semana e cumprir sua meta de vendas.

Personagens / Unidades

- Player

se move em movimento top down e pode atacar com a arma que está segurando

- Betestein

Monstro beterraba, tem mais vida, se arrasta até o player.

- Cenobra (não implementado para o MVP)

Monstro cenoura, é rápido, se lança de longe.

- Aranion (não implementado para o MVP)

Monstro cebola, consegue subir blocos quebráveis, morre com um ataque, deixa o player com visão limitada se ele for atacado ou se tiver no raio de explosão depois que ele morre.

Elementos do Gameplay

- Blocos quebráveis (não implementado para o MVP)

Player consegue atacar através deles mas não passa por eles. Os monstros não conseguem passar nem atacar através deles (com exceção do Aranion, que consegue subir neles e passar por cima), porém, cada

tentativa de ataque quebra o bloco mais. O bloco terá uma vida própria e os monstros poderão destruí-lo. Player compra esse bloco e escolhe onde colocar.

- **Armadilha (não implementado para o MVP)**

Armadilha de urso para os monstros. O player consegue escolher onde colocar. Desaparece quando é ativada.

- **Máquina de criação (não implementado para o MVP)**

Lugar onde se pode pegar as sementes que serão plantadas de dia. Tem opção de fazer upgrades que modificam os monstros (por exemplo, adicionar uma perna, deixá-los mais fortes e etc.). Modificações deixam os monstros mais fortes, porém o player ganha mais vegetal ao matar o monstro (como ele vende os vegetais, ele acaba ganhando mais dinheiro).

- **Barraquinha de venda (não implementado para o MVP)**

Aberta durante o dia, possibilita venda dos produtos obtidos em troca de dinheiro.

- **Máquina de compra(não implementado para o MVP)**

Possibilita a compra de armas e blocos para se preparar para a noite.

- **Semente**

Elemento que pode ser plantado em áreas de terra e cria um spawner de monstros.

- **Pá**

Arma base do jogo para atacar os monstros.

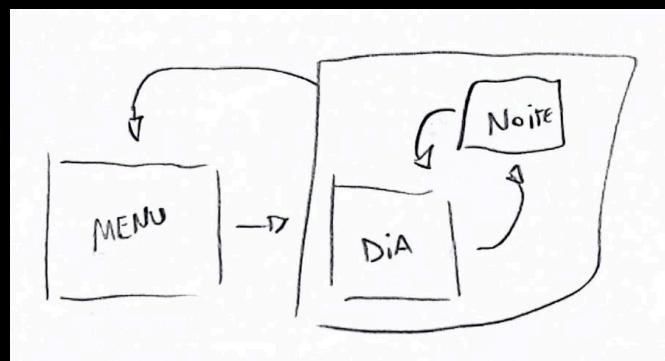
AI/AS

- Algoritmo de perseguição do jogador.
- Comportamento relativo de cada inimigo.
- Animação procedural que amplifica a realidade do movimento dos inimigos.

Interface-Jogador

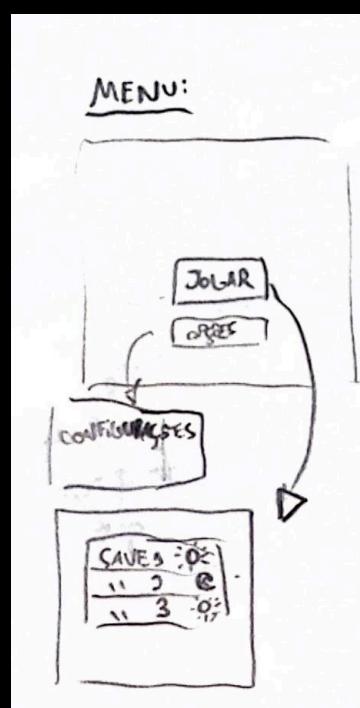
Fluxo de telas

- Esboço



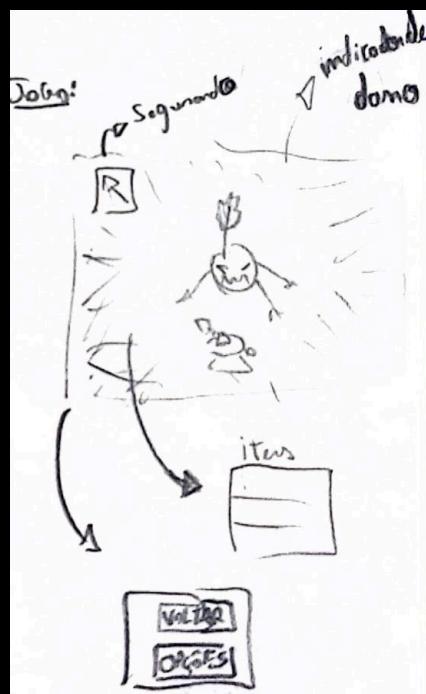
Requisitos funcionais (Wireframe)

- Esboço



Prototipação Visual + GUI

- Esboço



Resultado do MVP

- Tela Inicial



- Dia



- Noite



Arte Audiovisual

Áudio

- Músicas

Música dia, música noite, menu(Não implementada no MVP), música boss(Não implementada no MVP)

- Sonoplastia (FX)

Sons de Menu(Não implementada no MVP), Sons de jogo, Sons de UI(Não implementada no MVP)

Visual

- Identidade Visual

Shader pixelado + filtro de filme antigo(tela dando ruído) + filtro de cor



- GUI

UI em forma de Placas de madeira, remetendo a fazenda (Não implementada no MVP)

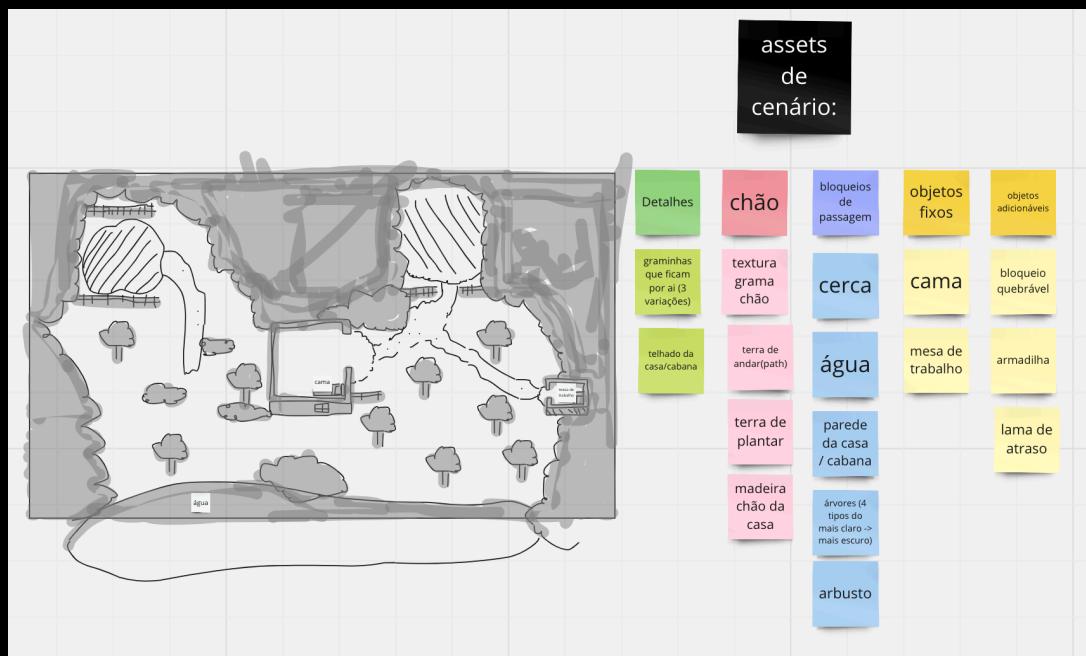
- Terrenos/Ambientes

Fundo difere dos personagens por menor saturação

Requisitos de Nível/Fase

Mapas

- Esboço



- Real



Game technical specification (GTS)

Mecânica do Jogo

Plataforma e Sistema Operacional

WebGL (itch.io) ou Desktop Build

Código

Godot, GDscript

Física e Estatística

Characterbody2D, CollisionShape2D do Godot

AI/AS

Navigation 2D godot

Descrição do Programa

Player

Animação Procedural:

Através do uso de um algoritmo de inverse kinematics foi feito o movimento da perna no arquivo Leg.gd.

```
func update_ik(targetPos):
    var offset = targetPos - global_position
    var disToTarget = offset.length()
    if disToTarget < MIN_DIST:
        offset = (offset / disToTarget) * MIN_DIST
        disToTarget = MIN_DIST
    var baseR = offset.angle()
    if is_nan(baseR):
        baseR = 0
    var baseAngles = sss_calc(lenUpper, lenLower, disToTarget)
    global_rotation = baseAngles["A"] + baseR
    Knee.rotation = baseAngles["C"]
```

`sss_calc()` calcula os ângulos de um triângulo através do valor de seus lados. Assim, ele permite o uso desses valores para mover o ângulo da perna para o `targetPos`.

```
if walking:  
    legDist = 10  
    _position.y = 20  
    if legN == 1:  
        PosX = cos(dir * funcTime * legSpeed) * legArch  
        PosY = sin(dir * funcTime * legSpeed) * stepHeight  
    else:  
        PosX = cos(dir * funcTime * legSpeed + PI) * legArch  
        PosY = sin(dir * funcTime * legSpeed + PI) * stepHeight
```

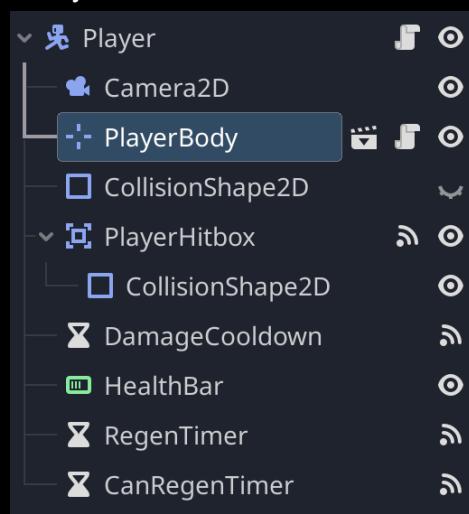
Na função `step()` em `Leg.gd` é feito o movimento da perna baseado em um círculo. O movimento circular, feito para as 2 pernas, cada um com uma fase de `PI` entre si, dá a impressão de movimento de caminhada.

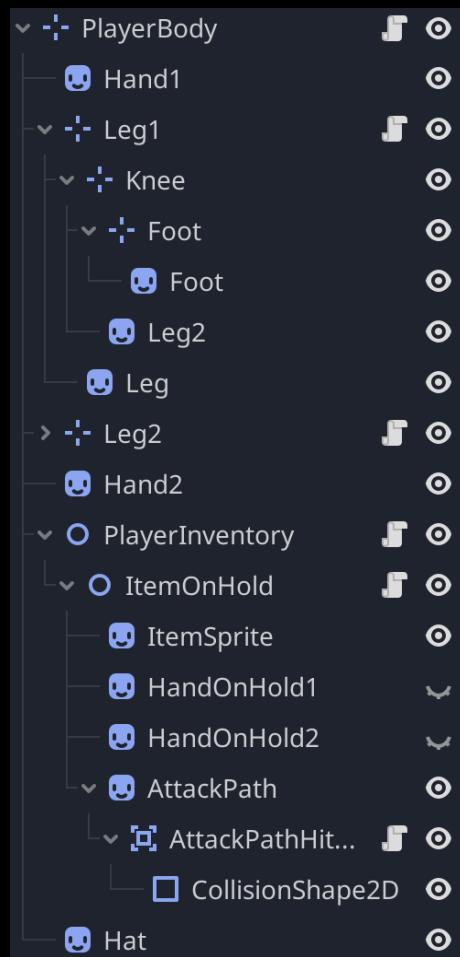
O resto da função `step()` é feita para o personagem ficar oscilando quando estiver parado.

O arquivo `PlayerBody.gd` é responsável por realizar todos os movimentos complementares feitos pelo Node `PlayerBody`, como por exemplo, o movimento do chapéu, o ataque com a pá e o movimento da direção das mãos. Em todos os casos, esses movimentos são calculados por funções matemáticas, assim como as pernas.

Por fim, `Player.gd` faz os demais controles relativos ao player, como controle de vida, movimento e outras variáveis complementares para o funcionamento.

O formato de `Player` e `PlayerBody` é dessa forma.





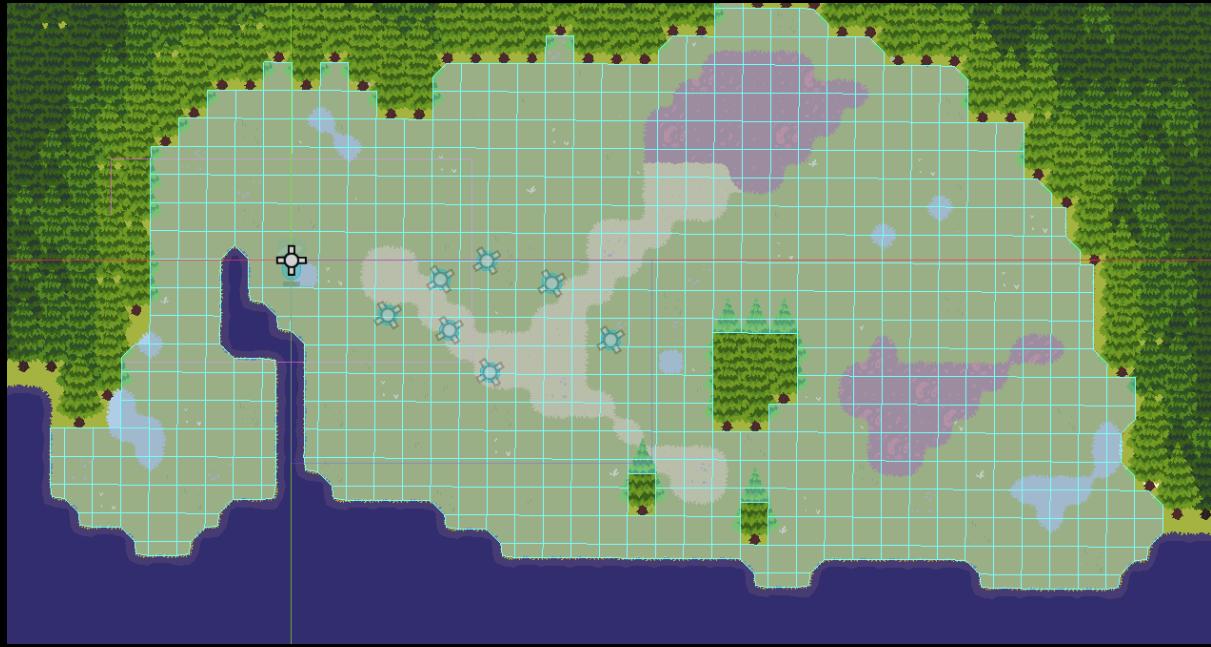
Enemy

Comportamento Base:

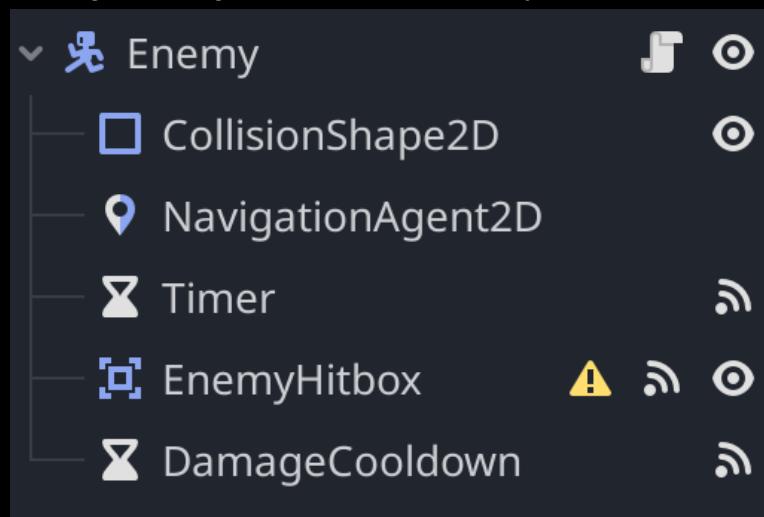
O comportamento base do inimigo segue um formato parecido com o player e é feito em `Enemy.gd`. Nesse arquivo é feito o controle de vida e executado o movimento baseado no algoritmo de perseguição

Perseguição:

A perseguição é feita com base no `NavigationAgent2D` que usa como base a camada de navigation feita em um `TileMap Auxiliar` que define as colisões para inimigos.

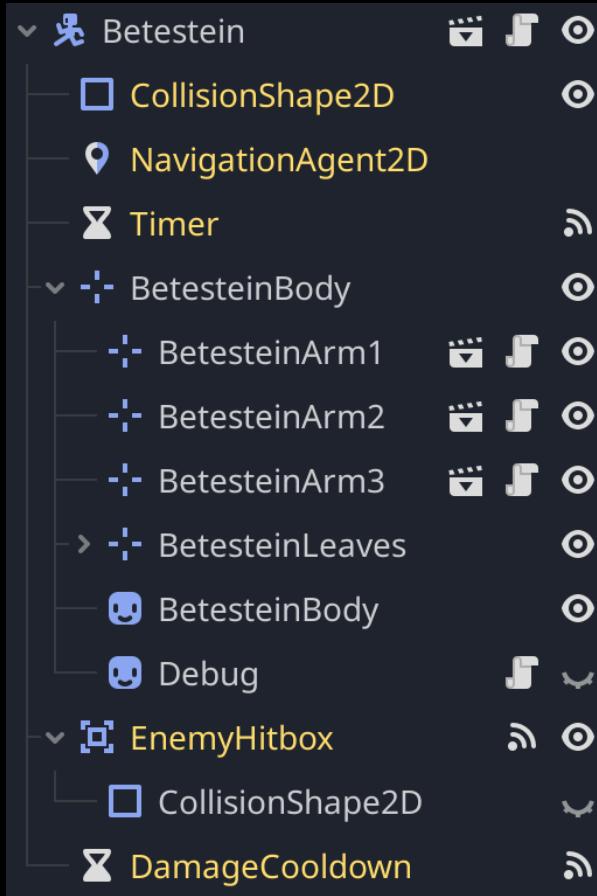


É passado para o inimigo um target, que no caso é o player.



Betestein:

O monstro Betestein é uma expansão da cena de Enemy e Betestein.gd é uma extensão da classe Enemy.gd.



Animação Procedural:

A parte mais relevante da animação procedural relativa ao Betestein é o movimento do braço. Nela é utilizada a mesma função de inverse kinematics para a perna do player. A função `mov()` em `BetesteinArm.gd` executa o movimento do braço. Ela se baseia em achar a posição `targetPos` através da direção que Betestein está caminhando. É utilizado um seno para oscilar a distância que `targetPos` está de Betestein, isso dá a impressão do braço tendo que se esticar para puxar o Betestein para frente. Adicionando oscilações extras no corpo em `Betestein.gd` temos o movimento completo desse monstro.

```

func update_hand():
    Hand.global_rotation = get_parent().get_parent().dir.angle()

func mov(delta):
    var dir
    if flip:
        dir = 1
    else:
        dir = -1
    var dirToTarget = get_parent().get_parent().dir.rotated(faseAngulo)
    var _offset = sin(dir * funcTime * movSpeed+faseMov) * movArch + 250
    if dir*cos(dir*funcTime * movSpeed+faseMov) < 0:
        OpenHand.visible = false
        ClosedHand.visible = true
    else:
        OpenHand.visible = true
        ClosedHand.visible = false
    var targetDir = targetPos - global_position
    angleDist = dirToTarget.angle() - targetDir.angle()
    if abs(angleDist) > PI:
        angleDist = -angleDist/abs(angleDist)*(2*PI - abs(angleDist))
    var angle
    if abs(angleDist) < 0.1:
        angle = dirToTarget.angle()
        angleDist = 0
    else:
        var angleVel = angleDist/abs(angleDist) * delta * 2
        angle = targetDir.angle() + angleVel
    var newDir = Vector2(cos(angle), sin(angle))
    targetPos = global_position + newDir * _offset

```

Note que a targetPos no final é igual a newDir * _offset. _offset representa a oscilação do seno que dá a impressão do braço se esticando para andar. Por sua vez, NewDir representa a direção que o braço vira. NewDir é a direção de movimento do Betestein, mas suavizada para que quando Betestein mude essa direção os braços não virem automaticamente.

Mecânicas de Jogo

Itens:

Todos os itens são criados a partir de um Resource base Item.tres. O item em si é do formato Area2D e pode ser adicionado ao inventário quando o player entra nessa área e clicar em pegar o item. Nesse caso, BaselItem.gd faz com que o item em questão se auto-delete e adicione seu resource ao inventário. Inventory.gd manda sinais item_changed e item_dropped quando itens novos são adicionados.

Por fim, PlayerInventory.gd conecta os sinais de Inventory.gd e responde a eles alterando o item na Node ItemOnHold de Player.

Ciclo Dia e Noite:

Para implementar o ciclo Dia-Noite foi utilizado um arquivo Global.gd em Autoload que define uma variável Global.nightTime que realiza esse controle. Através do uso dessa variável é feito o controle de todas as mecânicas do jogo.

Na troca de dia e noite as músicas respectivas de cada estágio trocam através de uma transição.

Plantação:

Para que seja possível plantar são necessárias 3 coisas: Que o item da semente seja coletável como item, Que seja possível plantar nas áreas de terra e, por fim, a criação do spawner de monstros.

O sistema para que a semente seja coletável já foi implementado com o sistema de itens, e funciona da mesma forma.

Para que seja possível saber se o player está pisando na terra para que a semente seja plantada foi utilizada uma camada de custom data layer no Tilemap com um valor booleano que define se o tile em que o player está é de terra. Se for possível plantar, a variável Global.canPlant é verdadeira.

```
func checkPlayerPlant():
    var tile_player_pos: Vector2i = tile_map.local_to_map(global_position/6.25)
    var tile_data : TileData = tile_map.get_cell_tile_data(0,tile_player_pos)
    if tile_data:
        canPlant = tile_data.get_custom_data("dirt")
        if canPlant:
            Global.playerCanPlant = true
        else:
            Global.playerCanPlant = false
    else:
        print("no tile_data")
```

No PlayerInventory.gd, quando é feito o tratamento do drop do item, é criado um spawner se o tipo de item for Seed e se Global.canPlant for verdadeira. Ou seja, agora, se o player dropar a semente em cima da terra ele cria um spawner de inimigos.

```
func _on_item_dropped():
    if Global.playerCanPlant and inventory.item.name == "Seed" and not Global.nightTime:
        var enemy_spawner = spawner.instantiate()
        enemy_spawner.global_rotation_degrees = rng.randf_range(0, 360)
        enemy_spawner.global_position = global_position
        enemy_spawner.player = parent.get_parent()
        enemy_spawner.enemy_scene = betestein
        parent.get_parent().get_parent().add_child(enemy_spawner)
        Global.numberOfSpawners += 1
        Global.planted = true
```

O spawner por sua vez instancia inimigos com um intervalo de tempo se estiver de noite. Cada spawner cria 5 inimigos com 30 segundos de intervalo entre cada e depois se auto-deleta. Esse controle do Spawner é feito em Spawner.gd.

```

func spawn_enemies():
    if enemy_scene == null:
        print("Enemy scene is not assigned")
        return
    spawnedEnemies += 1
    var enemyInstance = enemy_scene.instantiate()
    enemyInstance.player = player
    enemyInstance.z_index = 1
    enemyInstance.global_position = global_position
    get_parent().add_child(enemyInstance)

func _on_spawn_timer_timeout():
    canSpawn = true

func spawnHandling():
    if canSpawn and Global.nightTime:
        spawn_enemies()
        canSpawn = false

func _physics_process(_delta):
    if spawnedEnemies >= number_of_enemies:
        canSpawn = false
        Global.numberOfSpawners -= 1
        queue_free()
        spawnHandling()

```

Combate:

Se estiver com no mínimo uma semente plantada, o player pode clicar no botão NIGHT e começar a noite. Depois que todos os spawner são auto-deletados a noite acaba.

O combate do player com o enemy é baseado simplesmente na verificação se o enemy está na Area2D que representa o hitbox do player ou se a Node com o ataque do player está no hitbox do inimigo enquanto o player está atacando.

O Betestein tem vida igual ao número de braços, a cada vida perdida ele perde um braço e fica mais lento.

Player por sua vez tem sua vida conectada a uma barra de vida que exibe sua vida quando ela está menor que o total. Além disso, o recurso visual do shader de ruído aumenta sua intensidade quando o player está perdendo, isso serve para dar um feedback de que o player está em perigo e assim aumentar a tensão do jogo. Esse efeito é complementado com o som de ruído.

Recursos Visuais

Shader Pixelado:

O shader pixelado acontece durante o jogo todo e é grande parte da estilização do jogo. Ele se baseia no cálculo de diminuir e aumentar a tela de volta, assim "reduzindo a qualidade" e portanto deixando mais pixelado.

```
shader_type canvas_item;

uniform int amount = 75;

void fragment()
{
    vec2 grid_uv = round(UV * float(amount)) / float(amount);
    vec4 text = texture(TEXTURE, grid_uv);
    COLOR = text;
}
```

Shader Ruído:

O shader ruído é feito com configurações de randomização que simulam o comportamento de uma TV velha.

```
shader_type canvas_item;

uniform sampler2D SCREEN_TEXTURE: hint_screen_texture, repeat_disable, filter_nearest;
uniform float crt_brightness: hint_range(0.0, 1.0) = 1.0;
uniform float crt_white_noise: hint_range(0.0, 1.0) = 0.0;

float random(vec2 uv) {
    return fract(sin(dot(uv, vec2(12.9898, 78.233))) * 43758.5453);
}

void fragment() {
    float time = TIME;

    vec2 screen_uv = SCREEN_UV;
    vec3 color = texture(SCREEN_TEXTURE, screen_uv).rgb * (1.0 - crt_brightness * 0.5);
    color *= crt_brightness*5.0;
    vec2 noise_pixel = vec2(floor((screen_uv.x / SCREEN_PIXEL_SIZE.x) / 3.0) * 3.0, screen_uv.y) - sin(UV * PI);
    float white_noise = random(noise_pixel + vec2(sin(time + PI / 4.0), cos(time + PI / 8.0)));
    color = mix(color, vec3(white_noise), crt_white_noise);

    COLOR = vec4(color, 1.0);
}
```

Ele responde a tela da noite proporcionalmente a quantidade de vida do personagem. Utilizando NightFilter.gd, que muda os parâmetros do shader para criar a diferença entre noite e dia e as transições entre eles.

Considerações Finais

As mecânicas bases para gerar uma prova de conceito foram implementadas, porém não houve tempo para implementar elementos o suficiente de design de fase para gerar uma gameplay interessante.

Além disso, como era a minha primeira vez testando o Godot eu ainda não conhecia muitos recursos que eu poderia implementar nos jogos, o que fez com que parte do código não tenha sido feita da forma mais eficiente ou organizada.

Com base nisso, no futuro os objetivos seriam a reestruturação do código, visando generalizar o que for possível para que facilite a criação de mais recursos no futuro, e a criação de uma fase que proporcione uma gameplay mais interessante e elaborada, mesmo que para um MVP.