



Proyecto final Desarrollo de Aplicaciones Multiplataformas

*“Overcomplicate it!”
por Adrián Chao Béjar*

Índice

1. Requisitos del hardware.....	2
Recomendables.....	2
2. Guía de instalación del entorno.....	2
3. Puesta en marcha del entorno de desarrollo y aplicación.....	3
3.1 Fundamentos.....	3
3.2 Interfaz.....	4
3.3 Cómo ejecutar el juego.....	5
4. Explicación del funcionamiento de la aplicación.....	5
5. Explicación del código y de los procesos más interesantes.....	10
5.1 Base general.....	10
5.2 Modo construcción.....	12
5.2.1 Build Manager.....	12
5.2.2 Build Object.....	15
5.2.3 Test Generator Manager.....	16
5.3 Nave y fases.....	24
5.4 Jugador.....	27
5.5 NPCs.....	33
5.6 Interfaz y Menús.....	37
5.6.1 Menú principal.....	37
5.6.2 Interfaz del jugador.....	41
Dialogo.....	41
HUD.....	44
Menu.....	47
Game Over.....	48
5.7 Cinemáticas.....	48
6. Mayores dificultades encontradas.....	50
6.1 El modo construcción.....	50
6.1.1 Materiales.....	50
6.1.2 Cintas.....	50
6.1.2 Fábricas.....	51
6.2 Rendimiento.....	52
7. Conclusión.....	53
8. Otros materiales de apoyo.....	54
9. Biografía.....	54

1. Requisitos del hardware

Requisitos mínimos

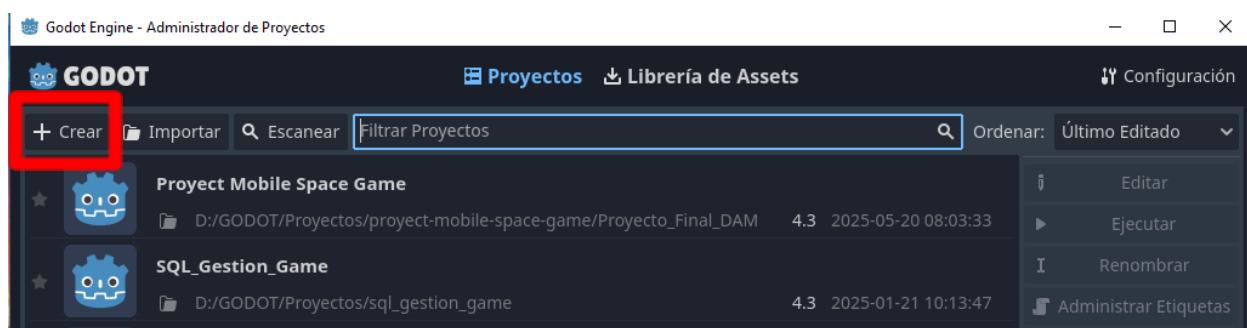
CPU	Intel Core i3 Octava generación
RAM	4GB RAM
Almacenamiento	82Mb
GPU	Intel UHD Graphics
Sistema Operativo	Windows 10/11

2. Guía de instalación del entorno

Godot es un entorno de desarrollo de software libre que está enfocado al desarrollo de videojuegos 2D y 3D. Para descargar el programa deberemos de ir a la página de Godot (<https://godotengine.org/download/windows>) y descargarnos la última versión.



Iniciaremos el instalador y cuando se haya instalado abriremos el programa. Para crear un proyecto deberemos de ir a la parte superior izquierda y le daremos a "Crear".



(Quiero mencionar que cuando intente crear un proyecto en mi ordenador, la ventana aparecía negro debido a que mi ordenador no era muy potente, asi que tuve que modificar project.godot de mi proyecto para cambiar el modo de renderizado a gl_compatibility)

[rendering]

`renderer/rendering_method="gl_compatibility"`

3. Puesta en marcha del entorno de desarrollo y aplicación

3.1 Fundamentos

Godot se basa en 4 pilares fundamentales:

-Nodos

Todos los objetos en godot se componen de nodos. Un nodo es un objeto con unas características donde podemos crear sistemas, objetos o entidades. Estos siguen una jerarquía donde podemos añadir nodos dentro de otros para crear objetos más complejos.

Cada nodo tiene una función única, por ejemplo, el CollisionShape crea una forma de colisión para los StaticBody y CharacterBody, los NavigationMesh crean una máscara para que los NavigationAgents puedan recorrerla, WorldEnvironment modifica el renderizado de la escena, hay incluso un nodo específico para crear una conexión entre jugadores, llamada MultiplayerSyncronizer y MultiplayerSpawner.

-Señales

Las señales son llamadas que tienen todos los nodos que mandan una señal cuando ocurre un evento. Cada nodo tiene su conjunto de eventos y estos eventos se conectan a un script donde se configura su funcionalidad.

Las señales se usan en muchas ocasiones, ya sea para ver si un collider ha entrado en un área, si se ha pulsado un botón en la interfaz o si ha cambiado la jerarquía del objeto.

-Scripts

Los scripts son líneas de códigos que podemos asignar a un nodo para darle una funcionalidad, como en otros motores de videojuegos, aquí programamos el funcionamiento del programa.

Los scripts usan mucho los nodos y señales para funcionar, ya que hay funciones ya hechas en nodos que podemos aprovechar para mejorar la funcionalidad de nuestro programa. Por ejemplo, el nodo Timer crea un temporizador, con él podemos ahorrarnos programar un temporizador interno y limitarnos a llamar a un método cuando llegue a 0.

-Escenas

Las escenas son objetos que se instancian en el juego. Godot fusiona las escenas y los prefabs y hace que sean lo mismo, es decir, que a ojos del motor, un escenario de un nivel y el prefab de un jugador son lo mismo.

Esto hace que el crear escenas y prefabs sea más sencillo. Con simplemente arrastrar el nodo a los archivos podemos crear una escena de cualquier nodo, también podemos crear una escena desde cero.

3.2 Interfaz

Cuando entremos en Godot y abramos un proyecto se nos abrirá el editor, el editor se divide en varias secciones:

-Escena: Muestra la escena actual con todos los nodos que la componen. Podemos modificar desde aquí la jerarquía de los nodos, su visibilidad, ver el tipo de los nodos, si son otras escenas instanciadas, si tiene grupos o si contiene scripts.

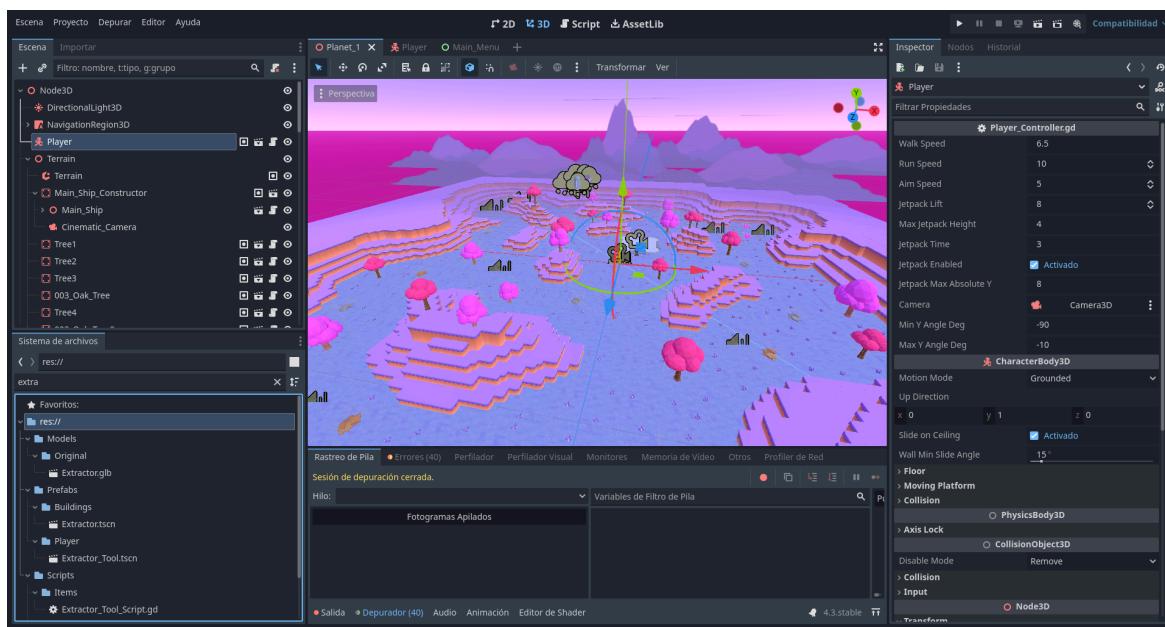
-Sistema de archivos: Aquí se gestionan todos los archivos del proyecto, podemos crear escenas, scripts, materiales y carpetas desde aquí.

-Escena actual: Cuando tenemos una escena abierta, se nos mostrará en el centro. Podemos cambiar la vista de la escena en la parte de arriba, pudiendo cambiar entre vista 2D, 3D, script sirve para ver los scripts que hemos abierto y AssetsLib carga los assets subidos por otros desarrolladores en Godot

-Depuración: Abajo de la escena actual está en menú de depuración, aquí se muestran los errores sintácticos, errores en tiempo de ejecución, las salidas por consola, información sobre el rendimiento del equipo y otras funciones. También podemos abrir el menú para crear animaciones y shaders desde aquí.

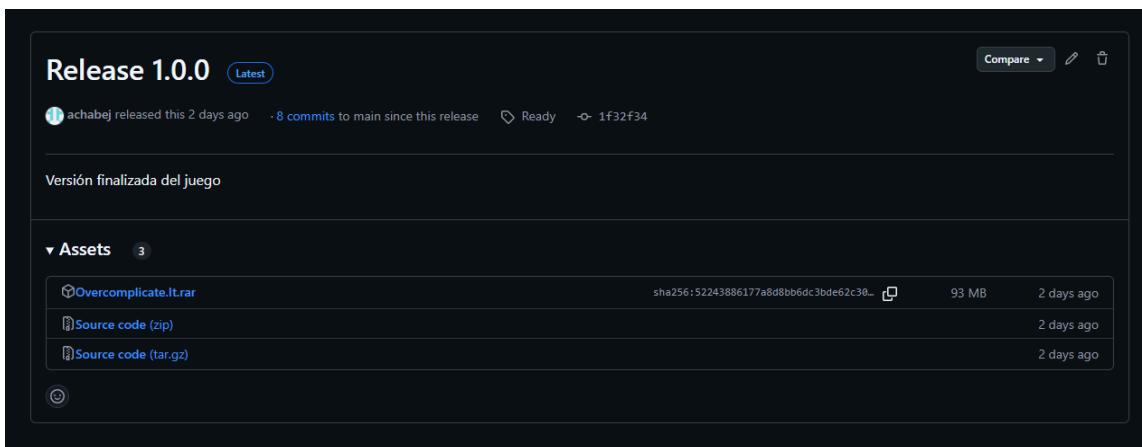
-Inspector: Nos muestra toda la información sobre el nodo seleccionado. La información está dividida en la jerarquía de los nodos, es decir, un Rigidbody tendrá la configuración de un rigidbody, luego el de CollisionObject (del cual hereda) y al final el de Node (de donde todos heredan)

-Nodos: Es una ventana situada al lado del inspector, sirve para modificar el grupo del cual el nodo pertenece y configurar las señales que puede emitir



3.3 Cómo ejecutar el juego

Dentro de git, en la sección de “Releases” se encuentra el ejecutable del juego, simplemente descargaremos el .zip del repositorio y dentro ejecutaremos el .exe llamado “Overcomplicate it!.



4. Explicación del funcionamiento de la aplicación

Overcomplicate it! Es un juego de gestión donde un robot debe escapar de un planeta después de haber explotado su estación. La mecánica principal es la construcción, el jugador puede recolectar recursos del entorno, luego colocar máquinas y cintas con estos materiales para procesar materiales para la nave.

Estos son los controles del juego:

- Mover (*WASD*)
- Propulsarse con un jetpack (*SPACE*)
- Correr (*WASD + SHIFT*)
- Apuntar (*CLICK DERECHO*) y disparar (*CLICK DERECHO + CLICK IZQUIERDO*)
- Rotar la cámara (*RUEDA DEL RATÓN*)
- Activar o desactivar el menú de construcción (*F*).

En modo construir:

- Construir edificios (*CLICK IZQUIERDO*)
- Destruir edificios (*CLICK DERECHO*)
- Rotar edificios (*R*)

Desde el menú de pausa (*ESC*) podemos ver los controles



La interfaz del juego se divide de la siguiente manera:

1. Barra de vida
2. Barra de propulsión (jetpack)
3. Información de la fase actual
4. Inventario del jugador
5. Menú de construcción



En el centro del mapa hay una plataforma de lanzamiento con una entrada, este es el edificio principal donde se llevarán todos los materiales para completar las fases y construir la nave.



Para construir la nave el jugador debe explorar el mapa en busca de materiales que recolectar, estos son Madera, Hierro, Cobre y Cristal. También tendrá que enfrentarse a enemigos que merodean alrededor del mapa.



Hay varios edificios que el jugador puede construir, cada uno con sus propias funcionalidades.

1) Cinta transportadora: Mueve los materiales de una fábrica a otra. Tiene una dirección única, pero pueden aceptar entradas desde sus laterales. (*Hierro: 2*)

2) Extractor: Extrae materiales de los depósitos que se encuentran en el mapa. Los extractores solo se pueden colocar encima de los depósitos. (*Madera: 10, Hierro: 10*)

3) Horno: Funde minerales para convertirlo en lingotes. Los lingotes se pueden hacer de hierro y de cobre. (*Hierro: 20*)

4) Refinería de acero: Fusiona lingotes de hierro y cobre para crear acero. (*Hierro: 15, Cobre: 15, Cristal: 15*)

5) Fábrica de placas: Convierte los lingotes en placas. Las placas pueden ser de hierro, cobre y acero. (*Hierro: 15, Cobre: 10, Cristal: 10*)

6) Fábrica de chips: Crea chips a partir de placas de cobre, placas de acero y cristales. (*Hierro: 20, Cobre: 20, Cristal: 30*)



Para avanzar en la fase, deberemos refinar el material y llevarlo con las cintas hasta la entrada de la plataforma.

En este ejemplo usamos un extractor para extraer hierro, con el horno lo fundimos a lingotes y al final los llevamos hasta la plataforma.



Cuando completamos un requisito, este se pondrá en verde y todo material que se meta del mismo tipo irá directamente al inventario del jugador. Al completar la fase, deberemos de ir hasta la plataforma y pulsar **E** para construir la pieza de la nave



Cada fase irá “sobre complicando ;)” el juego añadiendo más fábricas y requisitos hasta que la completemos. Cuando lleguemos a la última fase, habremos completado el juego, podremos montar en la nave y escapar del planeta.



5. Explicación del código y de los procesos más interesantes

5.1 Base general

La base del juego es el Game_Manager, este script es Global y sirve para gestionar las fases de la nave y el estado del juego. State determina el estado del juego para dictar que sistemas funcionan en cada momento.

```
▼ enum State {
  >I Play, # Modo por defecto
  >I Building, # Modo cuando vamos a colocar un edificio
  >I Destroying, # Modo cuando vamos a destruir un edificio
  >I Die, # Modo cuando morimos
  >I Ending # Modo cuando se inicia la cinematica del final (mayormente para que no se abra el menu)
}

var currentState = State.Play
var input_message
var black_overlay
```

Las fases de la nave es un diccionario de caracteres donde se escribe los materiales necesarios para cada fase (Son 5 fases en total)

```
20
21   #Lista de materiales necesarios para cada fase
22 ▼ var phases : Dictionary = {}
23 ▼ >I "Phase1": {
24   >I >I "Resources": {
25     >I >I >I "Iron_Ingot": 15,
26     >I >I >I "Copper_Ingot": 15
27     >I >I }
28   },
29 ▼ >I "Phase2": {
30   >I >I "Resources": {
31     >I >I >I "Steel_Ingot": 10,
32     >I >I >I "Copper_Ingot": 15,
33     >I >I >I "Cristal": 15
34     >I >I }
35   },
36 ▼ >I "Phase3": {
37   >I >I "Resources": {
38     >I >I >I "Steel_Plate": 10,
39     >I >I >I "Iron_Plate": 15,
40     >I >I >I "Copper_Plate": 15,
41     >I >I }
42   },
43 }
```

El Game_Manager controla varias cosas, como los recursos necesarios para la fase actual, el cambio entre fases, el fundido en negro al entrar en la escena y la actualización de la interfaz del jugador

```

  # Añade material a la fase
  func add_mat(material_type: String, amount: int = 1) -> void:
    # Comprueba que la fase existe, en caso de que entre un numero invalido
    var phase_key = "Phase%d" % current_phase
    if not phases.has(phase_key):
      return
    # Coge la cantidad de material necesario para ese tipo
    var max_amount = phases[phase_key]["Resources"].get(material_type, 0)
    if max_amount == 0:
      return
    if storage.get(material_type, 0) >= max_amount:
      add_mat_to_player(material_type)
    # Si el almacen no tiene el material guardado, lo inicializará
    if not storage.has(material_type):
      storage[material_type] = 0
    # Añade la cantidad al almacen
    storage[material_type] += amount
    # Evita que se desborde el almacen cuando esté lleno
    if storage[material_type] > max_amount:
      storage[material_type] = max_amount
    _update_ui()
    # Esperamos que el jugador confirme la fase acercándose a la nave
    if _check_phase_complete():
      print("✓ Fase completada, esperando confirmación en la nave...")
  
```

```

80   # Método para que el jugador confirme pasar de fase
81   func _on_player_confirm_phase():
82     if not _check_phase_complete():
83       return # seguridad
84     current_phase += 1
85     emit_signal("phase_changed", current_phase)
86   
```

```

88   # Limpia el almacen si se cambia de fase
89   if current_phase > phases.size():
90     storage.clear()
91     _update_ui()
92     return
93   
```

```

94   storage.clear()
95   _update_phase_storage()
96   _update_ui()
97   
```

```

98   # Inicializar el storage para la fase actual, para que tenga todas las claves con 0
99   func _update_phase_storage() -> void:
100     var phase_key = "Phase%d" % current_phase
101     if phases.has(phase_key):
102       for mat in phases[phase_key]["Resources"].keys():
103         if not storage.has(mat):
104           storage[mat] = 0
105     
```

```

106     # Devuelve un booleano si la fase ha sido completada
107     func _check_phase_complete() -> bool:
108       var phase_key = "Phase%d" % current_phase
109       if not phases.has(phase_key):
110         return false
111       
```

```

112       for mat in phases[phase_key]["Resources"].keys():
113         var required = phases[phase_key]["Resources"][mat]
114         if storage.get(mat, 0) < required:
115           return false
116       return true
  
```

Las escenas del juego se dividen en 3. El menú principal (Main_Menu), la escena del juego (Planet_1) y la cinematica del final (Ending_Scene). Todos los managers que son Globales tienen un método llamado “init”, este método sirve para que todos los managers se inicializan solo cuando se cargue la escena principal y no cuando solo está el Menú.

Estos métodos son llamados desde “Scene_Loader”.

```

1  extends Node3D
2  @onready var black_overlay = $"../CanvasLayer/black_overlay"
3
4  # Called when the node enters the scene tree for the first time.
5  func _ready() -> void:
6    # Inicializar lógicas del juego
7    GameManager.init()
8    ConveyorManager.init()
9    BuildManager.init()
10   
```

```

11   await GameManager.fade_black_overlay(true)
12
  
```

5.2 Modo construcción

Es el sistema más complejo, ya que se divide en varios scripts, los dos fundamentales son "Build_Manager" y "Build_Object".

5.2.1 Build Manager

Build_Manager se encarga de gestionar todo el sistema de construcción, con este script gestionamos el movimiento, rotación y colocación de los edificios, la gestión de sus prefabs, el inventario del jugador, los métodos para cambiarlo y la visualización de la interfaz de construcción.

Todos los edificios se cargan y se guardan en estas variables

```

3   # Prefabs de los edificios
4   var extractor : PackedScene = preload("res://Prefabs/Buildings/Extractor.tscn")
5   var convey_line : PackedScene = preload("res://Prefabs/Buildings/Convey_Line.tscn")
6   var convey_merger : PackedScene = preload("res://Prefabs/Buildings/Convey_Line.tscn")
7   var furnace : PackedScene = preload("res://Prefabs/Buildings/Furnace.tscn")
8   var steelRefinery : PackedScene = preload("res://Prefabs/Buildings/SteelRefinery.tscn")
9   var cristalRefinery : PackedScene
10  var chipsFactory : PackedScene = preload("res://Prefabs/Buildings/ChipsFactory.tscn")
11  var platesFactory : PackedScene = preload("res://Prefabs/Buildings/PlatesFactory.tscn")
12

```

Luego hay varios métodos por cada construcción que se llaman desde los botones de la interfaz. Cuando se selecciona un edificio se llama a "Spawn_Object" que cambia el CurrentSpawnable al edificio seleccionado.

```

# Funciones para spawnear construcciones
func SpawnExtractor():
    >| SpawnObj(extractor)

func SpawnConveyLine():
    >| SpawnObj(convey_line)
    >|
func SpawnConveyMerger():
    >| SpawnObj(convey_merger)

func SpawnFurnace():
    >| SpawnObj(furnace)
    >|
func SpawnSteelRefinery():
    >| SpawnObj(steelRefinery)
    >|
func SpawnChipsFactory():
    >| SpawnObj(chipsFactory)
    >|
func SpawnPlatesFactory():
    >| SpawnObj(platesFactory)

func SpawnObj(obj: PackedScene):
    >| if CurrentSpawnable:
        >| >| CurrentSpawnable.queue_free()

        >| CurrentSpawnable = obj.instantiate()
        >| grid_map.add_child(CurrentSpawnable)
        >| CurrentSpawnable.scale = Vector3.ONE

        >| var collision_shape = CurrentSpawnable.get_node("CollisionShape3D")
    >| if collision_shape:
        >| >| collision_shape.disabled = true

        >| GameManager.currentState = GameManager.State.Building
        >|
        >| # Actualiza el nombre actual y la UI
        >| var name = obj.resource_path.get_file().get_basename()
        >| update_materials_needed_UI(name)

```

CurrentSpawnable es en sí el edificio actual que hemos seleccionado, mientras estemos en modo Construcción, este seguirá al ratón con el método “Move_to_closest_tile” y se colocara con “place_building”.

Move_to_closest_tile también asegura que todos los edificios se coloquen en una cuadrícula que tiene la escena, esta se obtiene a partir de un raycast desde la cámara.

```
# Mueve la construcción a la casilla más cercana
func move_to_closest_tile(delta):
    if CurrentSpawnable == null:
        return

    var result = raycast_from_camera()

    if result:
        var cursor_pos = result.position
        var grid_pos = world_to_cell(cursor_pos)
        TargetGridCell = grid_pos
        var target_position = cell_to_world(grid_pos)

        CurrentSpawnable.global_position = CurrentSpawnable.global_position.lerp(target_position, MoveSpeed * delta)
```

En place_building cuando colocamos un edificio, lo que hacemos en realidad es hacer una copia, esta se instancia con el material que usan todas las fábricas y se cambia su Grupo a “Build” para que empiece a hacer sus funciones cuando esté colocada.

(Test_Generator_Manager es el método funcional de todas las fábricas, ahí se controla la producción según el tipo que sea pero el nodo se llama igual)

```
144    if OnGrid and AbleToBuild and Input.is_action_just_pressed("left_click"):
145        var name = CurrentSpawnable.scene_file_path.get_file().get_basename()
146        if not has_required_materials(name):
147            print("No hay suficientes materiales para construir ", name)
148            return
149
150        if build_sound != null:
151            build_sound.play()
152
153        consume_materials(name)
154        update_materials_needed_UI(name)
155
156        var obj = CurrentSpawnable.duplicate()
157        grid_map.add_child(obj)
158        obj.global_position = target_position
159
160        var collision_shape = obj.get_node("CollisionShape3D")
161        if collision_shape:
162            collision_shape.disabled = false
163
164            change_material(obj, basic_mat)
165            obj.add_to_group("Build")
166            obj.isBuild = true
167
168        if obj.has_node("Test_Generator_Manager"):
169            var gen = obj.get_node("Test_Generator_Manager")
170            if gen.has_method("activate"):
171                gen.activate()
172
173        if Input.is_action_just_pressed("Rotate_Building"):
174            rotate_building()
175
```

La eliminación de edificios se gestiona en handle_deletion, cuando se presiona click derecho sobre un edificio (que esté en el grupo “Build”) este se eliminará de la escena

```

# Elimina una construccion seleccionada
func handle_deletion(delta):
    if Building and Input.is_action_just_pressed("Delete_Building"):
        var result = raycast_from_camera()

        if result:
            var body = result.collider
            if body and body.is_in_group("Build"):
                var name = body.scene_file_path.get_file().get_basename()

                refund_materials(name)
                update_materials_needed_UI(name)

                if build_sound != null:
                    build_sound.play()

                body.queue_free()
                print("Edificio eliminado: ", body.name)

```

Entre todos estos métodos también se controla los materiales que tiene el jugador, cada vez que se coloca o se elimina un edificio se llaman o a “increase_mat” o “decrease_mat” que gestionan los materiales que el jugador consume.

Además de llamar a “Mat_PopUp_Controller” que es una etiqueta que aparece para informar de los materiales que se han usado.

```

#===== Gestión de materiales =====
func increase_mat(mat, amount):
    materiales[mat] += amount
    var canvas_node = Player.get_node("CanvasLayer").get_node("HUD")
    canvas_node.get_node("Materials").update_mat_UI()
    var label = "+%d %s" % [amount, mat]
    canvas_node.get_node("Mat_PopUp_Controller").call("show_material_popup", label, Color.GREEN)

func decrease_mat(mat, amount):
    if(materiales[mat] - amount >= 0):
        materiales[mat] -= amount
    else:
        amount = materiales[mat]
        materiales[mat] = 0
    var canvas_node = Player.get_node("CanvasLayer").get_node("HUD")
    canvas_node.get_node("Materials").update_mat_UI()
    var label = "-%d %s" % [amount, mat]
    canvas_node.get_node("Mat_PopUp_Controller").call("show_material_popup", label, Color.RED)

```

5.2.2 Build Object

Build_Object es un script que se coloca en el nodo padre de todas las construcciones, en él se gestionan las colisiones entre otros objetos para comprobar si es válido construir a partir de una lista de grupos (notBuildList).

Dentro del process se comprueba si el objeto es el CurrentSpawnable y si no choca con ningún grupo prohibido. También cambiamos la máscara del edificio si este está construido para que no altere las funcionalidades de las construcciones cuando no esté construido.

```

extends StaticBody3D

var objects: Array[Node3D] = [] # Lista de objetos válidos con los que estamos en colisión.
var ActiveBuildableObject: bool = true
var isBuild = false

@export var notBuildList: Array[String] = []
@export var SpawnActor: bool = true
@export var Actor: PackedScene

func _process(delta: float) -> void:
    if ActiveBuildableObject and not isBuild:
        var can_build := true
        for obj in objects:
            if isInProhibitedGroup(obj):
                can_build = false
                break
        BuildManager.AbleToBuild = can_build
    if isBuild:
        collision_mask = (1 << 1) | (1 << 2)

```

En apply_material_to_meshes gestionamos el material actual del objeto, se llama desde Build_Manager y dependiendo si es válido o no, se cambiará a Rojo o a Verde. El material se aplica de forma recursiva por cada hijo del nodo "Mesh".

```

# Cambia el material de todos los MeshInstance3D que no estén protegidos
func apply_material_to_meshes(mat: Material) -> void:
    _apply_material_recursive($Mesh, mat)

func _apply_material_recursive(node: Node, mat: Material) -> void:
    for child in node.get_children():
        if child is MeshInstance3D and not child.is_in_group("Ignore_Material_Change"):
            child.set_surface_override_material(0, mat)
            _apply_material_recursive(child, mat) # Recursivo para todos los niveles

```

runSpawn se encarga de la duplicación, simplemente instancia una copia de la construcción y lo añade a la escena.

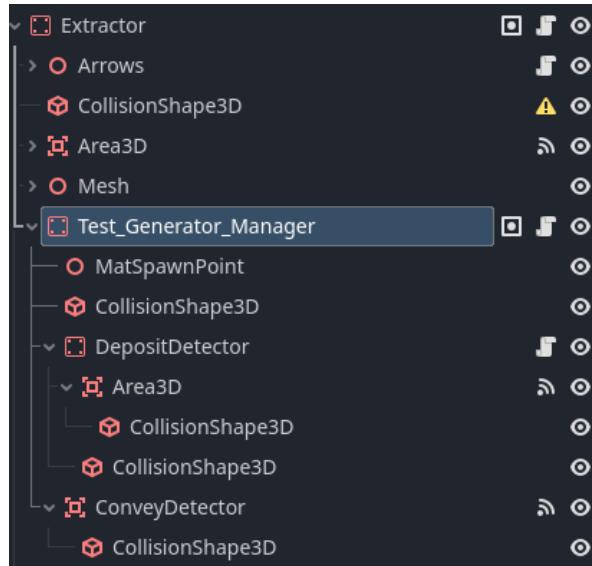
```

# Spawnea una copia del objeto
func runSpawn():
    if SpawnActor:
        var ActorInstance = Actor.instantiate()
        get_tree().root.add_child(ActorInstance)

```

5.2.3 Test Generator Manager

Por cada construcción tienen dentro un nodo que gestiona la función que hacen, este nodo se llama "Test_Generator_Manager". En él se coloca el script según la máquina. Las máquinas pueden ser o Cintas (Convey_Script), Extractores (Extractor_Script) o Máquinas (Machine_Script).



La gestión de las cintas se divide en Convey script y Convey Manager, que es otro script global. La función de Convey_Manager es tener registrado a todos las cintas de la escena, con un reloj interno se llama a "_on_tick_timeout", que manda a todas las cintas una señal para que intenten moverse.

```

1  # ConveyorManager.gd (singleton, debe añadirse a AutoLoad)
2  extends Node
3
4  var conveyors: Array = []
5
6  #Temporizador
7  var tick_timer: Timer
8  var tick_interval := 0.2
9
10 func init():
11     >I tick_timer = Timer.new()
12     >I tick_timer.wait_time = tick_interval
13     >I tick_timer.one_shot = false
14     >I tick_timer.autostart = true
15     >I tick_timer.timeout.connect(_on_tick_timeout)
16     >I add_child(tick_timer)
17
18 func _on_tick_timeout():
19     >I tick_conveyor()
20
21 func register_conveyor(conveyor: Node):
22     >I if conveyor not in conveyors:
23         >I >I conveyors.append(conveyor)
24         >I >I print("Conveyor registrado")
25
26 func unregister_conveyor(conveyor: Node):
27     >I conveyors.erase(conveyor)
28
29 func get_all_conveyors() -> Array:
30     >I return conveyors.duplicate()
31
32 func tick_conveyor():
33     >I for conveyor in conveyors:
34         >I >I if is_instance_valid(conveyor):
35             >I >I >I conveyor.try_move()
36

```

Convey_Script gestiona las cintas individualmente, su función es detectar otras cintas y entradas de material, mover los materiales cuando sea posible y eliminar correctamente los materiales si la cinta es eliminada

```

10  var entrada_rays: Array = []
11  var salida_raycast: RayCast3D = null
12  var entrada_index := 0
13
14  var current_material : CharacterBody3D = null
15  var target_position: Vector3 = Vector3.ZERO
16  var next_convey_manager: Node = null
17  var is_moving := false
18  var active_entry: Node = null
19  var active_entry_timeout_max := 3
20  var active_entry_timeout := 0
21
22  ◀ func _ready():
23    >I ConveyorManager.register_conveyor(self)
24    >I set_physics_process(true)
25
26    >I entrada_rays = $Entradas.get_children()
27    >I salida_raycast = $Convey_Manager/RayCast3D
28
29  ◀ func _process(_delta):
30    >
31    >> if current_material and not is_instance_valid(current_material):
32      >> current_material = null
33    >> if active_entry != null and not is_instance_valid(active_entry):
34      >> active_entry = null
35    >
36    >I $Label30.text = str(current_material)
37    >
38    >I cleanup_invalid_references()
39

```

Cuando entra un material dentro de la cinta, está lo asigna a "Current_material", la variable que guarda el material actual de la cinta.

```

7
8  ◀ func _on_body_entered(body: Node3D) -> void:
9    >I if !get_parent().is_in_group("Build"):
10      >I return
11
12    >I if body.get_parent().is_in_group("Build"):
13      >I var machine_root = find_parent_with_group(body, "Machine")
14    >I if machine_root:
15      >I machine_root.set("current_conveyor", self.get_parent())
16    >I elif body.is_in_group("Material") and current_material == null:
17      >I current_material = body
18    >I if body is CharacterBody3D:
19      >I body.velocity = Vector3.ZERO
20
21
22  ◀ func _on_body_exited(body: Node3D) -> void:
23    >I if body.is_in_group("Material") and body == current_material:
24      >I await get_tree().process_frame
25    >I if not is_instance_valid(body) or not get_overlapping_bodies().has(body):
26      >I current_material = null
27
28

```

En `try_move` se usa `active_entry` para comprobar que la cinta de enfrente es válida, si es así, llamará a “`Update_next_convey`” para obtener la posición central de la siguiente cinta para la animación

```
88 # Función principal para intentar mover el material al siguiente conveyor
89 ↵ func try_move() -> void:
90   ↵ cleanup_invalid_references()
91
92
93   ↵ if is_moving or current_material == null or !get_parent().is_in_group("Build"):
94     ↵   ↵ # Si no se puede mover, incrementar timeout si hay active_entry
95   ↵   ↵ if active_entry != null:
96     ↵   ↵   ↵ active_entry_timeout += 1
97     ↵   ↵   ↵ # Si supera timeout, liberar active_entry para evitar deadlock
98   ↵   ↵   ↵ if active_entry_timeout >= active_entry_timeout_max:
99     ↵   ↵   ↵   ↵ active_entry = null
100    ↵   ↵   ↵   ↵ active_entry_timeout = 0
101    ↵   ↵ return
102  ↵ else:
103    ↵   ↵ # Reseteamos timeout cuando estamos activos
104    ↵   ↵ active_entry_timeout = 0
105
106    ↵ update_next_convey()
107
108  ↵ if not is_instance_valid(next_convey_manager):
109    ↵   ↵ next_convey_manager = null
110    ↵   ↵ return
111
112  ↵ if not next_convey_manager.request_entry(self):
113    ↵   ↵ return
```

```
9 func update_next_convey():
10    if raycast and raycast.is_colliding():
11        var collider = raycast.get.collider()
12    if collider:
13        var convey_node = find.parent_with_group(collider, "Convey")
14    if convey_node:
15        var manager = convey_node.get_node_or_null("Convey_Manager")
16    if manager and manager != self and convey_node.is_in_group("Build"):
17        target_position = manager.get_center_position()
18        next_convey_manager = manager
```

Esta parte se encarga de iniciar el movimiento del material a la nueva posición, al ser el material un CharacterBody3D, simplemente actualizaremos la posición del objeto hasta la posición indicada.

```
# Función asincrónica para mover el material suavemente hasta target_pos
func move_material_to(material: CharacterBody3D, target_pos: Vector3, speed: float) -> void:
    while is_instance_valid(material) and material.global_position.distance_to(target_pos) > 0.1:
        if not is_instance_valid(material):
            return
        var direction = (target_pos - material.global_position).normalized()
        material.global_position += direction * speed * get_process_delta_time()
        await get_tree().process_frame

    if is_instance_valid(material):
        material.global_position = target_pos
```

Request_entry impide que la cinta mueva un material si no hay una cinta enfrente o si la cinta ya está ocupada.

```
9
10 func request_entry(from_conveyor: Node) -> bool:
11     # Limpiar active_entry si está muerto o inválido (evita deadlock)
12     if active_entry != null and not is_instance_valid(active_entry):
13         active_entry = null
14
15     if current_material != null or is_moving:
16         return false
17
18     if active_entry == null:
19         active_entry = from_conveyor
20
21     return active_entry == from_conveyor
```

Exit_tree se llama cuando se elimina la cinta, en el se hace una grán comprobación para asegurarse de que se elimina de la lista de cintas, se eliminan todos los materiales que tenga encima y elimina la referencia a active_entry tanto en sí misma como en la cinta de enfrente.

```
9
10 func _exit_tree():
11     ConveyorManager.unregister_conveyor(self)
12     for body in get_overlapping_bodies():
13         if body.is_in_group("Material") and is_instance_valid(body):
14             body.queue_free()
15
16     for conveyor in ConveyorManager.get_all_conveyors():
17         if conveyor.current_material != null:
18             conveyor.try_move()
19         if conveyor.active_entry == self:
20             conveyor.active_entry = null
21         if conveyor.next_convey_manager == self:
22             conveyor.next_convey_manager = null
23             conveyor.is_moving = false
24
25         if is_instance_valid(next_convey_manager) and next_convey_manager.active_entry == self:
26             next_convey_manager.active_entry = null
27
```

Los extractores tienen el script "extractor_script", este script controla la extracción de material y comprueba de que se encuentra en un depósito para producirlo.

```

    extends Node3D
    class_name Extractor

    @export var material_scene: PackedScene
    @export var spawn_position: Node3D
    @export var spawn_interval: float = 3.0

    var spawn_timer: float = 0.0
    var is_active: bool = false
    var blocking_materials := []
    var current_conveyor: Node3D = null
    var can_produce: bool = true
    var blocking_materials_list := []

    # Animación
    @onready var mesh: Node3D = $"../Mesh"
    var anim_time := 0.0
    var original_scale := Vector3.ONE

    func _ready() -> void:
        # Detector de conveyors
        var detector = get_node_or_null("ConveyDetector")
        if detector:
            detector.monitoring = true

```

Para comprobar si el extractor esta en un depósito o no se usa un area3D, este area tiene su propio script. Lo que hace es sobreescibir a comprobacion de Build_object y solo será válido construir cuando el area colisione con un depósito.

En el depósito se encuentra el prefab del material que debe extraer, si lo encuentra, cambia el valor de "material_scene" en extractor_script con el que se encuentra en el deposito.

```

    # Cuando el cuerpo entra en la zona de colisión
    func _on_body_entered(body: Node3D) -> void:
        # Verificar si el cuerpo pertenece al grupo 'Deposit'
        if not get_parent().get_parent().is_in_group("Build") and body.is_in_group("Deposit"):
            print("Depósito detectado: ", body.name)
            # Obtener el prefab del material asignado en el depósito
            mat_scene = body.get("material") # Obtener el prefab del material
            get_parent().material_scene = mat_scene
            is_deposit_detected = true

    # Cuando el cuerpo sale de la zona de colisión
    func _on_body_exited(body: Node3D) -> void:
        if not get_parent().get_parent().is_in_group("Build") and body.is_in_group("Deposit"):
            print("Depósito perdido: ", body.name)
            mat_scene = null # Limpiar el prefab del material
            is_deposit_detected = false

    # Actualiza el estado de construcción del extractor según si se detectó un depósito o no
    func update_extractor_status() -> void:
        # Solo el extractor en modo construcción puede afectar la variable global
        if get_parent().get_parent().is_in_group("Build"):
            return

```

Las máquinas y las cintas tienen ambos un “_on_body_enter” y “_on_body_exit” similar, ya que ambos se están buscando mutuamente para asignarse como “current_conveyor”, esto se hace porque al colocar primero un extractor y luego una cinta, solo se activa el _on_body_enter de la cinta y viceversa.

```

56
57  ↴ func _on_convey_detector_body_entered(body: Node3D) -> void:
58    ↴ if body.is_in_group("Build") and body.is_in_group("Convey") and is_instance_valid(body):
59      ↴ current_conveyor = body
60    ↴ elif body.is_in_group("Material"):
61      ↴ blocking_materials += 1
62      ↴ blocking_materials_list.append(body)
63
64  ↴ func _on_convey_detector_body_exited(body: Node3D) -> void:
65    ↴ if body == current_conveyor:
66      ↴ current_conveyor = null
67    ↴ elif body.is_in_group("Material"):
68      ↴ blocking_materials = max(0, blocking_materials - 1)
69      ↴ blocking_materials_list.erase(body)
70

```

En el “_process” el extractor tiene un contador, cuando este llegue a 0, intentará producir un material. Si está en el grupo “Build”, tiene la escena del material y hay un convey en la salida, instancia el material en la posición de “Spawn_Point”.

```

26  ↴ func _process(delta: float) -> void:
27    ↴ if not is_active:
28      ↴ return
29    ↴
30    ↴ spawn_timer += delta
31    ↴ if spawn_timer >= spawn_interval:
32      ↴ spawn_timer = 0.0
33    ↴ if blocking_materials == 0:
34      ↴ spawn_mat()
35    ↴
36    ↴ # Animación mientras está activo
37    ↴ if is_active:
38      ↴ anim_time += delta * 5.0 # Velocidad de animación
39      ↴ var scale_y = 1.0 + 0.1 * sin(anim_time)
40      ↴ mesh.scale.y = original_scale.y * scale_y
41    ↴ else:
42      ↴ mesh.scale = original_scale
43
44  ↴ func spawn_mat() -> void:
45    ↴ if can_produce and material_scene and spawn_position and is_instance_valid(current_conveyor) and current_co
46    ↴ var mat = material_scene.instantiate()
47    ↴ get_tree().root.add_child(mat)
48    ↴
49    ↴ mat.global_position = spawn_position.global_position
50    ↴ mat.add_to_group("Ignore_Build_Validation")
51    ↴ print("material spawneado")
52    ↴ var manager = current_conveyor.get_node_or_null("Convey_Manager")
53

```

Por último tenemos “machine_script”, este es similar al extractor pero contiene más requisitos para producir materiales. Primero tiene un diccionario de caracteres donde se define las recetas de la máquina, primero se escriben los materiales necesarios (Input) y luego el material que se expulsa (Mat Spawn).

Los materiales se definen con una clave y un valor, siendo el nombre del material y luego la cantidad (Ej: Iron: 20). De esta forma una fábrica puede tener varias recetas para distintos materiales, siempre y cuando estos materiales existan en el Build Manager.

```
# Materiales necesarios
▼ @export var required_materials: Array[Dictionary] = [
  ▼ >I {
    ▷ >I "Inputs": {},
    ▷ >I "Mat Spawn": null
  ▷ }
]
@export var max_storage: int = 10 #Cantidad maxima de material
```

Las fábricas tienen varias entradas, estas entradas están conectadas a “_on_material_entered” que comprueba si el material que recibe se encuentra en la receta, si es así lo añade al almacén de la fábrica.

```
-> 176 ▼ func _on_material_entered(body: Node3D) -> void:
  177   ▷ >I if not get_parent().is_in_group("Build"):
  178     ▷ >I return
  179
  180   ▷ >I if not body.has_method("get_material_type"):
  181     ▷ >I return
  182
  183   ▷ >I var mat_type = body.get_material_type()
  184   ▷ >I #print(mat_type)
  185
  186   ▷ >I if not can_accept_material(mat_type):
  187     ▷ >I # print("🔴 No se puede aceptar más de este material:", mat_type)
  188     ▷ >I return
  189
  190   ▷ >I if material_storage.has(mat_type):
  191     ▷ >I material_storage[mat_type] += 1
  192     ▷ >I body.queue_free()
  193     ▷ >I _check_activation()
```

Spawn_mat es un poco más complejo que en el extractor_script, ya que antes de producir busca entre todas las recetas de la fábrica y se comprueba cual es la adecuada según los materiales que tenga en el almacén

```

87
88  func spawn_mat() -> void:
89    if not spawn_position or not is_instance_valid(current_conveyor):
90      return
91
92    var best_recipe: Dictionary = {}
93    var best_fill_ratio: float = 0.0
94
95    for recipe in required_materials:
96      var inputs: Dictionary = recipe.get("Inputs", {})
97      var min_ratio := INF
98
99      var can_produce := true
100     for material in inputs.keys():
101       var required = inputs[material]
102       var available = material_storage.get(material, 0)
103
104       if available < required:
105         can_produce = false
106         break
107
108       var ratio = float(available) / float(required)
109       min_ratio = min(min_ratio, ratio)
110
111     if can_produce and min_ratio > best_fill_ratio:
112       best_fill_ratio = min_ratio
113       best_recipe = recipe
114

```

Si elige una, restará la cantidad del almacén con el de la receta, cogerá el prefab de "Mat Spawn" y lo instanciará si hay una cinta en la salida.

```

    if best_recipe.is_empty():
      return

    # Consumir materiales
    for material in best_recipe["Inputs"].keys():
      var amount = best_recipe["Inputs"][material]
      material_storage[material] -= amount

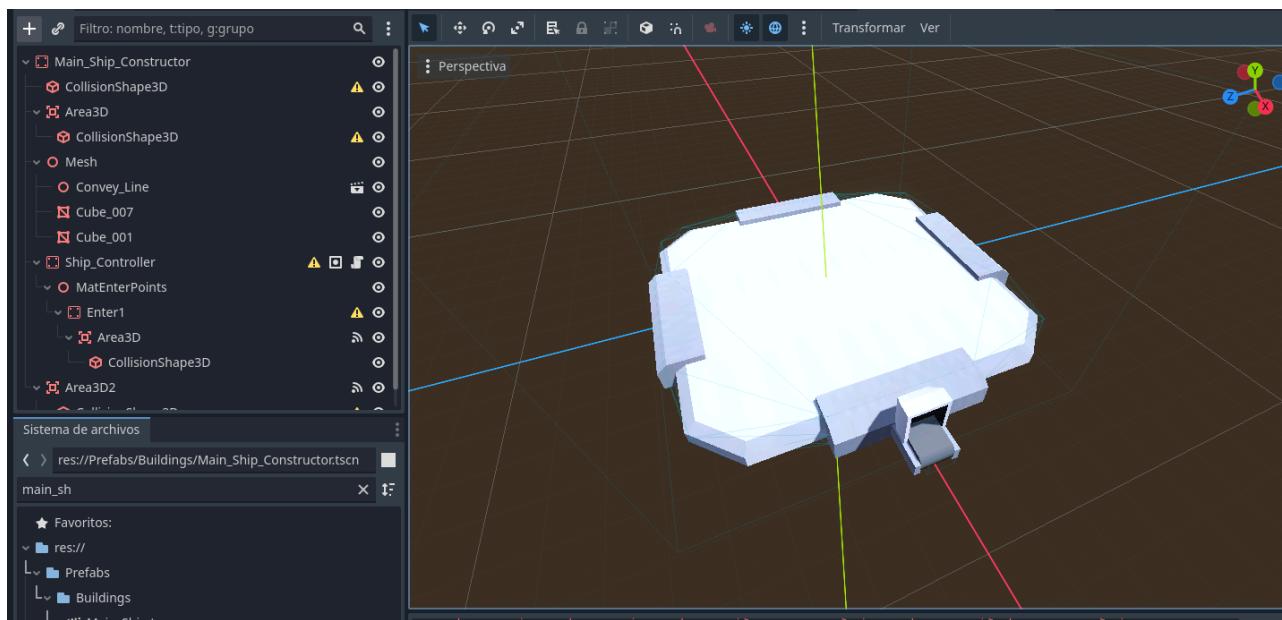
    # Instanciar resultado
    var prefab: PackedScene = best_recipe["Mat Spawn"]
    var instance = prefab.instantiate()
    get_tree().root.add_child(instance)
    instance.global_position = spawn_position.global_position
    instance.add_to_group("Ignore_Build_Validation")

    var manager = current_conveyor.get_node_or_null("Convey_Manager")
    if manager and manager is ConveyScript and manager.current_material == null:
      manager.current_material = instance

```

5.3 Nave y fases

Hemos explicado que las fases se gestionan desde Game_Manager en una lista de caracteres donde se definen los requisitos. Pues para poder contribuir en el avance de las fases tenemos un elemento llamado "Main_Ship_Constructor", que es básicamente la plataforma de lanzamiento.



Dentro de Main_Ship_Constructor contiene el script "Main_Ship_Script", este es el que se encarga de que los materiales se añadan al almacen de Game_Manager y controla el avance entre una fase y otra.

El "_on_area_3d_body_enter" comprueba el material que entra en la plataforma, si el material pertenece a la fase, lo añade, si no se encuentra en la fase actual, lo añadirá al inventario del jugador.

```

8  func _on_area_3d_body_entered(body: Node3D) -> void:
9    if body.has_method("get_material_type"):
10      var material_type = body.get_material_type()
11      print("Material detectado:", material_type)
12
13      # Obtener la fase actual
14      var phase_key = "Phase%d" % GameManager.current_phase
15
16      # Comprobar si el material está dentro de los recursos requeridos en la fase actual
17      if GameManager.phases.has(phase_key):
18        if GameManager.phases[phase_key]["Resources"].has(material_type):
19          GameManager.add_mat(material_type, 1)
20          body.queue_free()
21        else:
22          GameManager.add_mat_to_player(material_type)
23          body.queue_free()
24      else:
25        print("Fase actual no válida:", phase_key)
26
27

```

La intención de “add_mat_to_player” es para que el jugador pueda automatizar la recolección de materiales y facilitar el progreso del juego. Dependiendo del tipo de material, este añadirá al jugador el material en mayor cantidad, incentivando el uso de las fábricas para obtener material

```

160  ↵  match material_type:
161  ↵    ↵    "Iron":
162  ↵    ↵    ↵    BuildManager.increase_mat("Iron", 1)
163  ↵    ↵    "Copper":
164  ↵    ↵    ↵    BuildManager.increase_mat("Copper", 1)
165  ↵    ↵    "Iron_Ingot":
166  ↵    ↵    ↵    BuildManager.increase_mat("Iron", 2)
167  ↵    ↵    "Copper_Ingot":
168  ↵    ↵    ↵    BuildManager.increase_mat("Copper", 2)
169  ↵    ↵    "Iron_Plate":
170  ↵    ↵    ↵    BuildManager.increase_mat("Iron", 3)
171  ↵    ↵    "Copper_Plate":
172  ↵    ↵    ↵    BuildManager.increase_mat("Copper", 3)
173  ↵    ↵    "Steel_Ingot":
174  ↵    ↵    ↵    BuildManager.increase_mat("Iron", 2)
175  ↵    ↵    ↵    BuildManager.increase_mat("Copper", 2)
176  ↵    ↵    "Steel_Plate":
177  ↵    ↵    ↵    BuildManager.increase_mat("Iron", 4)
178  ↵    ↵    ↵    BuildManager.increase_mat("Copper", 4)
179  ↵    ↵    "Cristal":
180  ↵    ↵    ↵    BuildManager.increase_mat("Cristal", 1)
181  ↵    ↵    _:
182  ↵    ↵    ↵    #Esto es una broma, ignoralo
183  ↵    ↵    ↵    var root = get_tree().current_scene
184  ↵    ↵    ↵    var metal_pipe = root.get_node("Audio/Metal_Pipe")
185  ↵    ↵    ↵    metal_pipe.play()
186  ↵    ↵    ↵    _update_ui()
187

```

Cuando se completa una fase, se necesita que el jugador esté cerca de la plataforma para avanzar, el script comprueba que el jugador este dentro y si todas las fases están completas aparecerá un mensaje indicando la acción que va a realizar.

Al pulsar el botón avanzaremos a la siguiente fase, pero si hemos llegado al final, se ejecutará la animación del final.

```

28  ↵  func _on_area_3d_player_entered(body: Node3D) -> void:
29  ↵    ↵    if body.is_in_group("Player"):
30  ↵    ↵    ↵    player_inside = true
31
32
33  ↵  func _on_area_3d_player_exited(body: Node3D) -> void:
34  ↵    ↵    if body.is_in_group("Player"):
35  ↵    ↵    ↵    player_inside = false
36
37  ↵  func _process(_delta: float) -> void:
38  ↵    ↵    if player_inside and GameManager.all_phases_completed:
39  ↵    ↵    ↵    GameManager.input_message.visible = true
40  ↵    ↵    ↵    GameManager.input_message.get_node("RichTextLabel").bbcode_enabled = true
41  ↵    ↵    ↵    GameManager.input_message.get_node("RichTextLabel").text = "[center] Presiona para m
42  ↵    ↵    elif player_inside and GameManager._check_phase_complete():
43  ↵    ↵    ↵    GameManager.input_message.visible = true
44  ↵    ↵    ↵    GameManager.input_message.get_node("RichTextLabel").bbcode_enabled = true
45  ↵    ↵    ↵    GameManager.input_message.get_node("RichTextLabel").text = "[center]Presiona para cons
46  ↵    ↵    else:
47  ↵    ↵    ↵    GameManager.input_message.visible = false
48
49  ↵    ↵    if player_inside and Input.is_action_just_pressed("Interact"):
50  ↵    ↵    ↵    if GameManager.all_phases_completed:
51  ↵    ↵    ↵    ↵    emit_signal("play_ending_anim")
52  ↵    ↵    ↵    else:
53  ↵    ↵    ↵    ↵    print("Interact")
54  ↵    ↵    ↵    ↵    emit_signal("player_ready_for_next_phase")
55  ↵

```

La nave que vamos construyendo tiene un script llamado "Ship_State", que controla la visibilidad de las piezas según la fase. Básicamente las piezas se llaman igual que las fases y cada vez que se complete una, se irá cambiando la visibilidad.

```

1  extends Node
2
3  var phases_part := ["Phase_1", "Phase_2", "Phase_3", "Phase_4", "Phase_5"]
4  @onready var building_sound = $HammerSound
5  @onready var complete_sound = $CompleteSound
6  var build_sound_count
7
8  > func _ready():
17
18  <func _on_phase_changed(new_phase: int):
19    >I print(new_phase)
20
21    >I build_sound_count = 0
22    >I building_sound.play()
23
24  <func _on_building_sound_finished():
25    >I if new_phase - 2 < phases_part.size():
26      >I >I var node_name = phases_part[new_phase - 2]
27      >I >I var node = get_node_or_null(node_name)
28      >I >I if node:
29        >I >I >I node.visible = true
30        >I >I >I _set_colliders_enabled(node, true)
31        >I >I >I _set_particles_emitting(node, true)
32
33  <func _set_colliders_enabled(root: Node, enabled: bool):
34    >I for child in root.get_children():
35      >I >I if child is CollisionShape3D or child is CollisionPolygon3D:
36        >I >I >I child.disabled = not enabled
37        >I >I >I elif child.has_method("get_children"):
38          >I >I >I >I _set_colliders_enabled(child, enabled)
39
40  <func _set_particles_emitting(root: Node, emitting: bool):
41    >I for child in root.get_children():
42      >I >I if child is GPUParticles3D or child is CPUParticles3D:
43        >I >I >I child.emitting = emitting
44        >I >I >I elif child.has_method("get_children"):
45          >I >I >I >I _set_particles_emitting(child, emitting)
46
47  <func _on_complete_sound_finished():
48    >I for child in root.get_children():
49      >I >I if child is CollisionShape3D or child is CollisionPolygon3D:
50        >I >I >I child.disabled = not enabled
51        >I >I >I elif child.has_method("get_children"):
52          >I >I >I >I _set_particles_emitting(child, emitting)

```

A esto se le suma unos métodos para activar los colliders y partículas de cada pieza, como también un método para reproducir los sonidos en un orden específico.

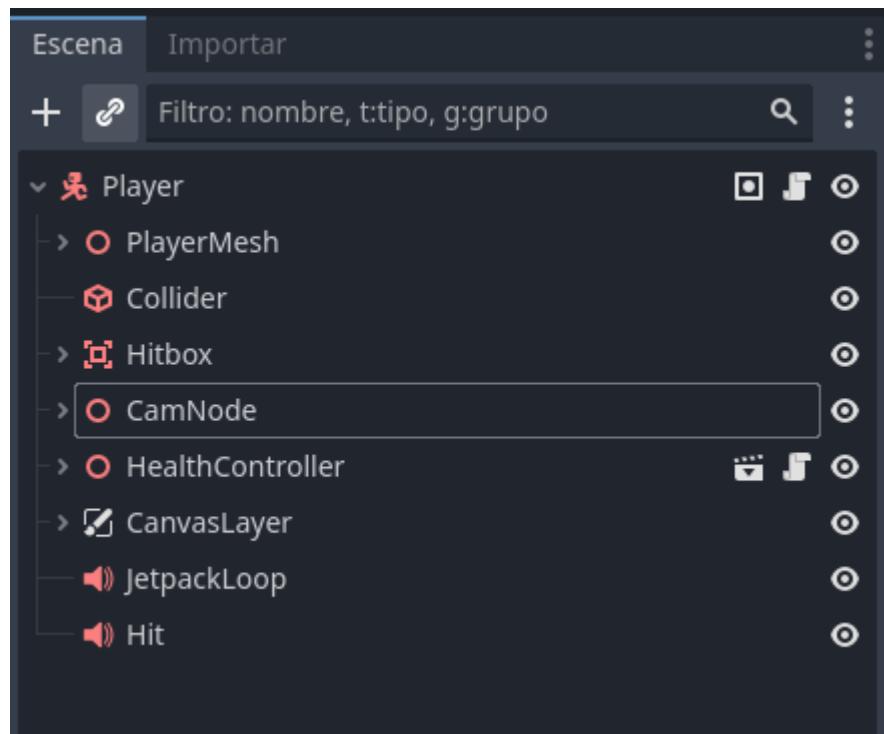
```

42
43
44
45
46
47
48
49
50
51
52

```

5.4 Jugador

El jugador es un CharacterBody3D, un nodo que se usa normalmente para crear al jugador y NPCs. En él contiene el propio jugador con el modelo del personaje, la cámara, el health controller y la interfaz del juego.



El script principal que controla el jugador se llama “Player_Controller”, es el script con más atributos del juego. Las variables que definimos al principio controlan desde la velocidad del jugador, el salto y distancia del jetpack, el control de la cámara y la referencia al modelo del jugador.

```

1  extends CharacterBody3D
2
3  # Configuración de movimiento
4  @export var WALK_SPEED = 6.5
5  @export var RUN_SPEED = 10
6  @export var AIM_SPEED = 3
7
8  # =====JETPACK=====
9  @export var JETPACK_LIFT = 8
10 @export var MAX_JETPACK_HEIGHT = 2.0
11 @export var JETPACK_TIME = 2.0
12 @export var jetpack_enabled: bool = true
13 @onready var jetpack_bar = $"CanvasLayer/HUD/Jetpack_Bar"
14 @export var JETPACK_MAX_ABSOLUTE_Y: float = 10.0
15 var jetpack_cooldown_timer := 0.0
16 var jetpack_min_height := 4.0
17 var current_max_height := 4.0
18 var height_increase_speed := 8
19
20 # Variables adicionales para el empuje del jetpack
21 var lift_min_value := 4.0
22 var current_lift := 4.0
23 var lift_increase_speed := 6.0
24
25 # Variables locales de movimiento
26 var current_speed = WALK_SPEED
27 var jetpack_time_left = JETPACK_TIME
28 var is_jetpacking = false
29 var recharge_delay = 0.2
30 var recharge_timer = 0.0
31 var starting_jetpack_y = 0.0
32
33 # Configuración de la cámara
34 @export var camera: Camera3D
35 @export var min_y_angle_deg: float = -90.0
36 @export var max_y_angle_deg: float = -45.0
37 @onready var cam_node = $CamNode
38 var vertical_angle := deg_to_rad(-45.0) #Angulo principal
39 var target_vertical_angle := deg_to_rad(-45.0) #Angulo siguiente
40 var vertical_angle_lerp_speed := 10.0 #Velocidad de giro
41
42 # Modelo del jugador
43 @onready var player_model = preload("res://addons/Godot_4_3D_Characters/addons/gdquest_gdbot/gbot_skin.tscn")
44 const BLEND_SPEED = 5.0
45 @onready var player_mesh = $PlayerMesh
46 @onready var health_controller = $HealthController
47 @onready var hand_controller = $PlayerMesh/weapon_holder
48 var current_y_rotation := 0.0
49
50 @onready var jetpack_sound: AudioStreamPlayer3D = $JetpackLoop
51
52
53 # Rotación horizontal del jugador
54 const ROTATION_SPEED = 6.0
55 const ROTATION_SHOOT_SPEED = 15.0
56

```

El primer método que encontramos sirve para controlar el input del ratón. Aquí controlamos si se ha movido la rueda del ratón arriba o abajo, dependiendo del input este irá cambiando el valor de "target_vertical_angle" para luego aplicar.

```

61  func _unhandled_input(event: InputEvent) -> void:
62    if event is InputEventMouseButton and event.pressed:
63      var angle_step := deg_to_rad(5) # Sensibilidad del scroll
64
65      if event.button_index == MOUSE_BUTTON_WHEEL_UP:
66        target_vertical_angle -= angle_step
67      elif event.button_index == MOUSE_BUTTON_WHEEL_DOWN:
68        target_vertical_angle += angle_step
69
70      # Limita el ángulo vertical deseado
71      target_vertical_angle = clamp(target_vertical_angle, deg_to_rad(min_y_angle_deg), deg_to_rad(max_y_angle_deg))
72

```

En "_physsic_process" controlamos el estado del jugador. La primera línea sirve para cuando alcanzamos el final, ocultamos la interfaz y evitamos que se ejecute el código de abajo. Luego se llaman a los métodos que controlan el jetpack, la herramienta de extracción y el movimiento. Por último se controla el ángulo de la cámara y el cambio a modo construcción cuando se pulse la tecla "F".

```

73  func _physics_process(delta: float) -> void:
74    if GameManager.currentState == GameManager.State.Ending:
75      player_mesh.visible = false
76      $CanvasLayer/HUD.visible = false
77      $CanvasLayer/Game_Over.visible = false
78      $CanvasLayer/Dialogue.visible = false
79      camera.current = false
80      return
81
82      #Detener el movimiento si hay un dialogo
83      check_dialog_state(delta)
84
85      # Procesa funcionalidades del jugador
86      handle_jetpack(delta)
87      handle_shooting(delta)
88      handle_movement(delta)
89
90      # Interpolación suave hacia el ángulo deseado
91      vertical_angle = lerp(vertical_angle, target_vertical_angle, vertical_angle_lerp_speed * delta)
92
93      # Aplicar la rotación suavizada a la cámara
94      var rot = cam_node.rotation
95      rot.x = vertical_angle
96      cam_node.rotation = rot
97
98      # Cambia al modo construcción si se activa la acción correspondiente
99      if Input.is_action_just_pressed("Building_Mode"):
100        BuildManager.ChangeState()
101

```

“Handle_jetpack” controla el control del salto. La primera parte del código controla el sonido del jetpack y la barra del jetpack, básicamente si se ha pulsado espacio y no se está en el suelo, se irá consumiendo “jetpack_time_left”. Si se acaba, pasará un pequeño delay antes de que se recargue de nuevo.

```

102  ↴ func handle_jetpack(delta: float) -> void:
103    ↴ if DialogManager.is_dialogue_active:
104      ↴ return
105    ↴
106    ↴ $PlayerMesh/Trail_Node.visible = is_jetpacking
107    ↴
108    ↴ if is_jetpacking:
109      ↴ if !jetpack_sound.playing:
110        ↴ jetpack_sound.play()
111    ↴ else:
112      ↴ jetpack_sound.stop()
113    ↴
114    ↴ # Aplica gravedad personalizada si no está en el suelo
115    ↴ if not is_on_floor():
116      ↴ velocity.y += -10.0 * delta
117    ↴ else:
118      ↴ velocity.y = 0
119    ↴ if recharge_timer >= recharge_delay:
120      ↴ if jetpack_time_left < JETPACK_TIME:
121        ↴ jetpack_time_left = min(jetpack_time_left + delta, JETPACK_TIME)
122    ↴ else:
123      ↴ recharge_timer += delta
124    ↴
125    ↴ # Actualiza la barra visual del jetpack
126    ↴ if jetpack_bar:
127      ↴ jetpack_bar.value = (jetpack_time_left / JETPACK_TIME) * jetpack_bar.max_value
128

```

La segunda parte aplica la fuerza del jetpack al jugador. Hay bastante comprobacion ya que no solo estamos elevando al jugador, sino que también hay una pequeña aceleración para que el movimiento sea más natural y hay una altura límite que cuando se alcanza bloquea al jugador para que no pueda elevarse más

```

130  ↴ if not jetpack_enabled:
131    ↴ jetpack_cooldown_timer += delta
132  ↴ if jetpack_cooldown_timer >= 0.15:
133    ↴ jetpack_enabled = true
134    ↴ jetpack_cooldown_timer = 0.0
135    ↴
136    ↴ # Activa el uso del jetpack si es posible
137  ↴ if jetpack_enabled and jetpack_time_left > 0 and Input.is_action_pressed("jump"):
138    ↴ recharge_timer = 0
139    ↴ jetpack_time_left = max(jetpack_time_left - delta, 0)
140    ↴
141  ↴ if not is_jetpacking:
142    ↴ starting_jetpack_y = global_transform.origin.y
143    ↴ current_max_height = jetpack_min_height
144    ↴ current_lift = lift_min_value
145    ↴ is_jetpacking = true
146    ↴
147    ↴ current_lift = min(current_lift + lift_increase_speed * delta, JETPACK_LIFT)
148    ↴ current_max_height = min(current_max_height + height_increase_speed * delta, MAX_JETPACK_HEIGHT)
149    ↴
150    ↴ var relative_max = starting_jetpack_y + current_max_height
151    ↴ var absolute_max = JETPACK_MAX_ABSOLUTE_Y
152    ↴ var effective_max_height = min(relative_max, absolute_max)
153    ↴
154  ↴ if global_transform.origin.y < effective_max_height:
155    ↴ velocity.y = current_lift
156  ↴ else:
157    ↴ velocity.y = 0
158  ↴ else:
159  ↴ if is_jetpacking:
160    ↴ is_jetpacking = false
161    ↴ jetpack_enabled = false

```

“Handle_shooting” controla el disparo de las herramientas, en este caso la de extracción. El disparo del jugador se divide en 2 fases, primero el jugador apunta con click derecho, lo que hace que gire en dirección a donde apunta el ratón, luego al pulsar click derecho mandará una señal a la herramienta, que es un objeto a parte, para que dispare.

```

163  func handle_shooting(delta: float) -> void:
164    if DialogManager.is_dialogue_active:
165      return
166
167    # Obtiene el arma actual del jugador
168    var current_weapon = hand_controller.get_current_weapon()
169
170    # Controla la rotación y el disparo mientras se apunta
171    if Input.is_action_pressed("right_click") and current_weapon and not BuildManager.Building:
172      var offset = -PI * 0.05
173      var screen_pos = camera.unproject_position(global_transform.origin)
174      var mouse_pos = get_viewport().get_mouse_position()
175      var angle = screen_pos.angle_to_point(mouse_pos)
176
177      var direction_to_mouse = Vector3(cos(angle + offset), 0, sin(angle + offset)).normalized()
178      look_in_direction(direction_to_mouse, delta, ROTATION_SHOOT_SPEED)
179
180    if Input.is_action_pressed("left_click"):
181      current_weapon.shoot()
182
183    # Detiene el disparo al soltar el clic
184    if Input.is_action_just_released("left_click") or Input.is_action_just_released("right_click"):
185      if current_weapon and current_weapon.has_method("stop_shooting"):
186        current_weapon.stop_shooting()
187

```

El método de shoot en la herramienta de extracción se encarga de lanzar el rayo del arma, el rayo se ajusta el tamaño según la distancia al objeto que ha colisionado y según si es un depósito o un enemigo enviará activará su respectivo temporizador.

```

20  func shoot():
21    is_shooting = true
22    laser_raycast.enabled = true
23    laser_mesh.visible = true
24
25    if !sound_loop.playing:
26      sound_loop.play()
27
28    laser_raycast.force_raycast_update()
29
30    var hit_distance = default_length
31
32    if laser_raycast.is_colliding():
33      var collider = laser_raycast.get.collider()
34      print(collider)
35
36    if collider.is_in_group("Deposit") or collider.is_in_group("Tree"):
37      current_deposit = collider
38    if extraction_timer.is_stopped():
39      extraction_timer.start()
40    elif collider.is_in_group("Enemy"):
41      current_deposit = null
42    if attack_timer.is_stopped():
43      attack_timer.start() # Solo arrancamos el timer aquí
44    else:
45      current_deposit = null
46      extraction_timer.stop()
47      attack_timer.stop()
48
49    var origin = laser_raycast.global_transform.origin
50    var collision_point = laser_raycast.get_collision_point()
51    hit_distance = origin.distance_to(collision_point) / 8.5
52

```

```

52
53    impact_particles.global_position = collision_point
54    if not impact_particles.emitting:
55      impact_particles.restart()
56      impact_particles.emitting = true
57    else:
58      impact_particles.emitting = false
59      current_deposit = null
60      extraction_timer.stop()
61      attack_timer.stop()
62
63    laser_mesh.scale.z = hit_distance
64
65
66  func stop_shooting():
67    sound_loop.stop()
68
69    is_shooting = false
70    laser_raycast.enabled = false
71    laser_mesh.visible = false
72    impact_particles.emitting = false
73    extraction_timer.stop()
74    attack_timer.stop()
75    current_deposit = null

```

Estos temporizadores ejecutarán su acción correspondiente cuando lleguen a 0, para el deposito sumará el material al inventario del jugador, y para el enemigo recibirá daño en su HealthController.

```

77  ↳ func _on_extractor_timer_timeout() -> void:
78    ↳ if current_deposit and is_shooting:
79      ↳   var mat_name = current_deposit.resource_type
80      ↳   BuildManager.increase_mat(mat_name, 1)
81
82  ↳ func _on_attack_timer_timeout() -> void:
83    ↳ if is_shooting and laser_raycast.is_colliding():
84      ↳   var collider = laser_raycast.get.collider()
85    ↳   if collider.is_in_group("Enemy"):
86      ↳     print("Dañando enemigo por timer")
87      ↳     collider.get_node("HealthController").damage(5)
88      ↳     attack_timer.start() # Reinicia timer para siguiente daño
89

```

"Handle_movement" se encarga de controlar el movimiento del jugador. Dependiendo de que este haciendo, el jugador tendrá una velocidad u otra, por ejemplo, al apuntar su velocidad sera igual a "AIM_SPEED", al correr será "RUN_SPEED" y cuando camine o salte con el jetpack será "WALK_SPEED".

```

188  ↳ func handle_movement(delta: float) -> void:
189    ↳ if DialogManager.is_dialogue_active:
190      ↳   return
191
192    ↳ # Calcula dirección del input y movimiento
193    ↳   var current_weapon = hand_controller.get_current_weapon()
194    ↳   var input_dir := Input.get_vector("move_left", "move_right", "move_up", "move_down")
195    ↳   var input_rot := Input.get_axis("ui_left", "ui_right")
196    ↳   var is_moving = input_dir.length() > 0.1
197    ↳   var direction := (global_basis * Vector3(input_dir.x, 0, input_dir.y)).normalized()
198
199    ↳ # Rota el personaje con teclas de rotación
200    ↳ if input_rot:
201      ↳   global_rotate(up_direction, PI * -1.7 * delta * input_rot)
202      ↳   input_dir = Vector2.ZERO
203
204    ↳ # Determina velocidad y animación según situación
205    ↳ if is_jetpacking:
206      ↳   current_speed = WALK_SPEED
207      ↳   player_model.jump()
208    ↳ elif Input.is_action_pressed("right_click") and current_weapon and not BuildManager.Building:
209      ↳   current_speed = AIM_SPEED
210    ↳ elif Input.is_action_pressed("sprint"):
211      ↳   current_speed = RUN_SPEED
212    ↳ else:
213      ↳   current_speed = WALK_SPEED
214
215    ↳ velocity.x = direction.x * current_speed
216    ↳ velocity.z = direction.z * current_speed

```

La segunda parte del código controla mayormente las animaciones del jugador, dentro de su modelo hay varios métodos que ejecutan una animación. Dependiendo de lo que haga, cambiará su animación, y en el caso de apuntar o correr, cambiará la velocidad de las animaciones para adaptarse al movimiento.

```

218     >>> if is_moving and not (Input.is_action_pressed("right_click") and current_weapon and not BuildManager.Building):
219         >>>     look_in_direction(direction, delta, ROTATION_SPEED)
220
221         >>> move_and_slide()
222
223         >>> # Aplica animaciones según el estado del jugador
224     << 224 if not is_on_floor():
225         >>>     if velocity.y < 0:
226             >>>         player_model.fall()
227         >>>     else:
228             >>>         player_model.jump()
229     << 229 elif is_moving:
230         >>>     var target_blend = 0.0
231         >>>     var anim_speed = 1.0
232
233     << 233 if Input.is_action_pressed("sprint") and not Input.is_action_pressed("right_click"):
234         >>>     target_blend = 1.0
235         >>>     anim_speed = 1.3
236     << 236 elif Input.is_action_pressed("right_click"):
237         >>>     target_blend = 0.1
238         >>>     anim_speed = 0.55
239
240         >>>     player_model.walk_run_blending = lerp(player_model.walk_run_blending, target_blend, delta * BLEND_SPEED)
241         >>>     player_model.set_anim_speed(anim_speed)
242         >>>     player_model.walk()
243     << 243 else:
244         >>>     player_model.walk_run_blending = lerp(player_model.walk_run_blending, 0.0, delta * BLEND_SPEED)
245         >>>     player_model.idle()
246
247

```

Por último está "Check_dialog_state", éste cambia el ángulo y el zoom de la cámara cuando hay un diálogo, oculta la interfaz y elimina el CurrentSpawnable de Buildmanager cuando hay un dialogo.

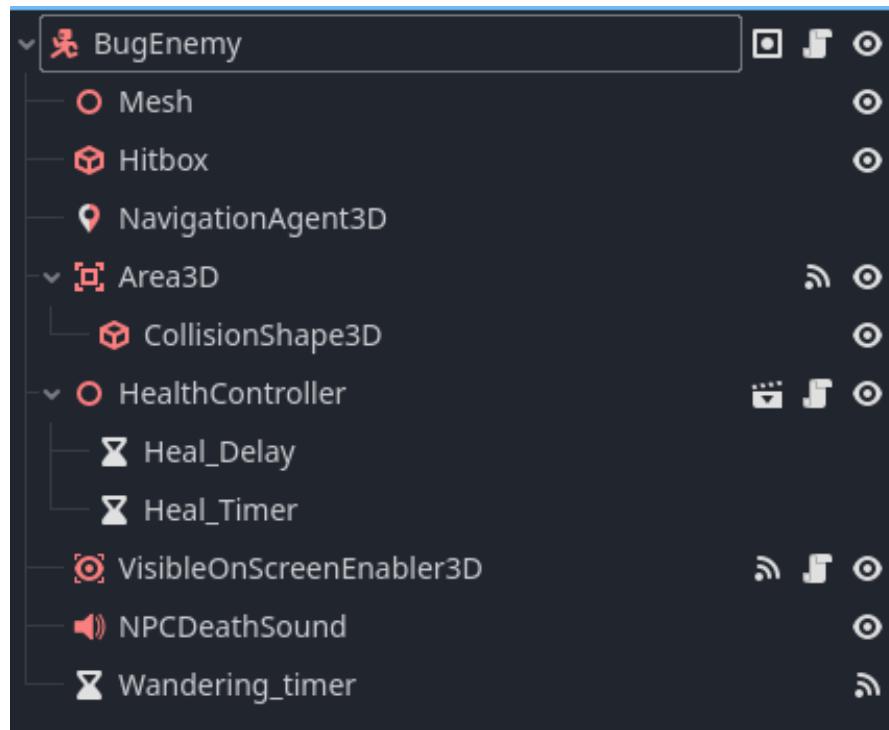
```

253     # Controla el estado del jugador cuando hay un diálogo
254     >>> func check_dialog_state(delta: float) -> void:
255         >>>     # Bloqueo por diálogo activo
256     << 256 if DialogManager.is_dialogue_active:
257         >>>     # Bloquear movimiento, disparo y construcción
258         >>>     $CanvasLayer/HUD.visible = false
259         >>>     velocity = Vector3.ZERO
260         >>>     move_and_slide()
261         >>>     player_model.idle()
262     << 262 if BuildManager.CurrentSpawnable:
263         >>>     BuildManager.CurrentSpawnable = null
264         >>>
265         >>>     hand_controller.hide_weapon()
266         >>>
267         >>>     # Suavemente rotar el jugador hacia la cámara
268         >>>     var to_camera = (camera.global_transform.origin - global_transform.origin)
269         >>>     to_camera.y = 0
270         >>>     to_camera = to_camera.normalized()
271         >>>     look_in_direction(to_camera, delta, 4.0)
272
273         >>>     # Cámara: suavemente rotar y hacer zoom
274         >>>     target_vertical_angle = deg_to_rad(-20.0)
275         >>>     camera.fov = lerp(camera.fov, 30.0, 3.0 * delta) # Zoom in
276
277         >>>     return # No ejecutar más lógica mientras hay diálogo
278
279     << 279 else:
280         >>>     # Restaurar cámara cuando no hay diálogo
281         >>>     $CanvasLayer/HUD.visible = true
282         >>>     camera.fov = lerp(camera.fov, 60.0, 3.0 * delta) # Zoom out
283         >>>     hand_controller.show_weapon()
284
285

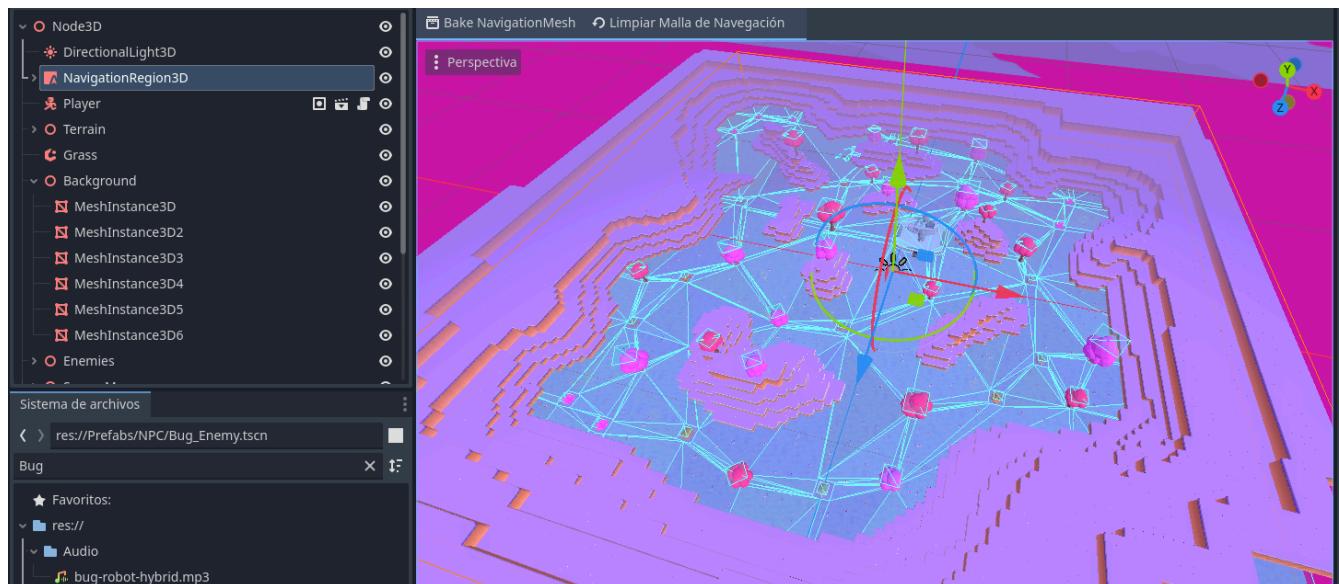
```

5.5 NPCs

Los NPCs (En este caso BugEnemy) son similares al jugador, ambos usan un CharacterBody3D pero estos tienen un nodo extra llamado NavigationAgent3D. Este nodo sirve para que los CharacterBody3Ds puedan moverse a través del escenario.



Con el nodo NavigationAgent3D creamos una malla sobre el escenario, ésta lo usa los NPCs para moverse y evitar colisionar con otros elementos estáticos.



Dentro del script de los NPCs (NPC_Movement), para controlar el estado del enemigo usamos un enum, ahí guardamos los estados que puede tener, como Wandering, Attack o Follow. Luego “current_state” guardará el estado actual.

```

23
24     enum State {
25         >| Wandering,
26         >| Alert,
27         >| Follow,
28         >| Attack,
29         >| Dead
30     }
31
32     var current_state = State.Wandering

```

En el _process se controla qué función debe ejecutarse según el estado de “current_state”, se hace algo similar para las animaciones.

```

62     func _process(delta):
63         >| match current_state:
64             >| >| State.Wandering:
65                 >| >| >| _state_wandering(delta)
66             >| >| State.Alert:
67                 >| >| >| _state_alert(delta)
68             >| >| State.Follow:
69                 >| >| >| _state_follow(delta)
70             >| >| State.Attack:
71                 >| >| >| _state_attack(delta)
72             >| >| State.Dead:
73                 >| >| >| _state_dead()

```

```

74     >|
75         >| match current_state:
76             >| >| >| State.Wandering, State.Follow:
77                 >| >| >| if velocity.length() > 0:
78                     >| >| >| >| NPC_mesh.walk()
79                 >| >| >| else:
80                     >| >| >| >| NPC_mesh.idle()
81             >| >| >| State.Alert:
82                 >| >| >| >| NPC_mesh.idle()
83

```

Cuando el estado sea “Wandering” el NPC se irá moviendo de forma aleatoria, cogerá una posición alrededor de él y caminará hasta llegar al objetivo, luego esperará por unos segundos y volverá a moverse.

```

84     func _state_wandering(delta):
85         >| if is_waiting:
86             >| >| wander_timer += delta
87             >| >| if wander_timer >= wander_wait_time:
88                 >| >| >| is_waiting = false
89                 >| >| >| wander_timer = 0.0
90                 >| >| >| _choose_new_wander_target()
91         >| else:
92             >| >| if nav_agent.is_navigation_finished():
93                 >| >| >| is_waiting = true
94                 >| >| >| wander_timer = 0.0
95                 >| >| >| velocity = Vector3.ZERO
96                 >| >| >| move_and_slide()
97         >| else:
98             >| >| >| var dir = (nav_agent.get_next_path_position() - global_transform.origin).normalized()
99             >| >| >| velocity = dir * speed
100            >| >| >| move_and_slide()
101            >| >| >| _rotate_mesh_towards(dir, delta)

```

Para cambiar de dirección llamamos a “_choose_new_wander_target”, que simplemente elige una posición aleatoria, alrededor del NPC y se lo envia al NavAgent.

```

183  ↴ func _choose_new_wander_target():
184    ↴   var random_direction = Vector3(
185      ↴     randf_range(-20, 20),
186      ↴     0,
187      ↴     randf_range(-20, 20)
188    ).normalized()
189    ↴   var random_target = global_transform.origin + random_direction * wander_radius
190    ↴   nav_agent.target_position = random_target
191

```

Junto a Wandering hay un contador que cuando llega a 0, este asigna una nueva dirección al NPC, esto sirve para evitar que se quede atascado intentando llegar a una posicion que no puede alcanzar.

```

237    ↴   # Cambia de dirección si no se llega al objetivo actual
238  ↴ func _on_wandering_timer_timeout() -> void:
239    ↴   _choose_new_wander_target()
240    ↴   $Wandering_timer.start()
241

```

El estado Alert se llama cuando el jugador se acerca demasiado al NPC, esto se controla con un Area3D.

```

43  ↴ func _on_body_entered(body):
44    ↴   if body == player:
45      ↴   player_in_range = true
46      ↴   current_state = State.Alert
47      ↴   alert_timer = 0.0
48      ↴   print("Jugador detectado, pasando a ALERT")
49
50  ↴ func _on_body_exited(body):
51    ↴   if body == player:
52      ↴   player_in_range = false
53      ↴   current_state = State.Wandering
54      ↴   is_waiting = true
55      ↴   wander_timer = 0.0

```

“_state_alert” simplemente espera unos segundos antes de llamar a Follow. Esto se hace para simular la “impresion” del NPC al detectar al jugador.

```

func _state_alert(delta):
  ↴   alert_timer += delta
  ↴   velocity = Vector3.ZERO
  ↴   move_and_slide()
  ↴
  ↴   # Mirar hacia el jugador sin moverse
  ↴   if player:
  ↴     ↴   var dir = (player.global_transform.origin - global_transform.origin).normalized()
  ↴     ↴   _rotate_mesh_towards(dir, delta)
  ↴
  ↴   # Pasar a Follow si el tiempo de alerta se cumple
  ↴   if alert_timer >= alert_time:
  ↴     ↴   current_state = State.Follow
  ↴     ↴   print("Tiempo alerta terminado, pasando a FOLLOW")

```

En el estado Follow el NPC seguirá la posición del jugador hasta que esté suficientemente cerca, con lo que llamará a Attack, o que el jugador se aleje demasiado, en tal caso pasará de nuevo a Wandering.

```

127  ↴ func _state_follow(delta):
128    ↵  if not player_in_range:
129      ↵    # Si jugador ya no está en el área, regresa a wander y idle
130      ↵    current_state = State.Wandering
131      ↵    is_waiting = true
132      ↵    wander_timer = 0.0
133      ↵    velocity = Vector3.ZERO
134      ↵    move_and_slide()
135      ↵    NPC_mesh.idle() # Forzar idle para que no parezca corriendo en el sitio
136      ↵    print("Jugador fuera de rango, pasando a WANDERING")
137      ↵    return
138
139    ↵    # Moverse hacia el jugador
140    ↵    var dir = (player.global_transform.origin - global_transform.origin).normalized()
141    ↵    nav_agent.target_position = player.global_transform.origin
142    ↵    velocity = dir * speed
143    ↵    move_and_slide()
144    ↵    _rotate_mesh_towards(dir, delta)
145
146    ↵    # Si está muy cerca, atacar
147    ↵    if (player.global_transform.origin - global_transform.origin).length() <= attack_distance:
148      ↵      current_state = State.Attack
149      ↵      attack_timer = 0.0
150      ↵      print("Jugador cerca, pasando a ATTACK")

```

El estado Attack activa el contador “attack_timer”, cuando este llegue a 0 mandará una señal al healthcontroller del jugador para dañarlo, Mientras el jugador esté en el rango, este seguirá atacando.

```

152  ↴ func _state_attack(delta):
153    ↵  if not is_attacking:
154      ↵    NPC_mesh.attack()
155      ↵    player.get_node("HealthController").damage(5)
156      ↵    is_attacking = true
157      ↵    attack_timer = 0.0
158
159    ↵    attack_timer += delta
160
161    ↵    velocity = Vector3.ZERO
162    ↵    move_and_slide()
163
164    ↵    if player:
165      ↵      var dir = (player.global_transform.origin - global_transform.origin).normalized()
166      ↵      _rotate_mesh_towards(dir, delta)
167
168    ↵    # Duración del ataque
169    ↵    if attack_timer >= 0.8:
170      ↵      is_attacking = false
171
172    ↵    # Si jugador salió del área, regresa a wander
173    ↵    if not player_in_range:
174      ↵      current_state = State.Wandering
175    ↵    else:
176      ↵      # Si jugador está fuera del rango de ataque, vuelve a follow
177      ↵      var dist = (player.global_transform.origin - global_transform.origin).length()
178    ↵      if dist > attack_distance:
179      ↵        current_state = State.Follow

```

Por último, el estado Dead ejecuta una animación antes de eliminar el NPC de la escena.

```

218  ↴ func _state_dead():
219    ↴   mesh.power_off()
220    ↴
221    ↴   $NPCDeathSound.play()
222    ↴
223    ↴   var tween = create_tween()
224
225    ↴   # Crear el primer tween y configurar easing
226    ↴   var step1 = tween.tween_property(self, "scale", Vector3.ONE * 0.2, 1)
227    ↴   step1.set_trans(Tween.TRANS_SINE)
228    ↴   step1.set_ease(Tween.EASE_OUT)
229    ↴
230    ↴   $Wandering_timer.start()|
231    ↴
232    ↴   # Esperar a que termine step1
233    ↴   await step1.finished
234
235    ↴   queue_free()
236

```

5.6 Interfaz y Menús

5.6.1 Menú principal

El menú principal es una escena a parte que se ejecuta al principio. El menú se divide en 2 nodos, “menu_default” y “configuration”.



“Menu_default” contiene los botones principales, estos botones son controlados en “Main_menu_controller” el cual se encarga en su mayoría de ejecutar las animaciones del menu.

El _process controla la animacion para el movimiento de la estacion espacial y de los colores cambiantes del titulo.

```

35  ↳ func _process(delta: float) -> void:
36    >I   total_time += delta
37
38    >I   # Animación de la estación
39    >I   var y_offset = sin(total_time * 1.2) * 5.0
40    >I   space_station_sprite.position.y = 100 + y_offset
41
42    >I   #Animación del titulo
43    >I   var r = (sin(total_time * speed) + 1.0) * 0.5
44    >I   var g = (sin(total_time * speed + TAU / 3) + 1.0) * 0.5
45    >I   var b = (sin(total_time * speed + 2.0 * TAU / 3) + 1.0) * 0.5
46    >I   title_text.add_theme_color_override("font_shadow_color", Color(r, g, b, 1.0))
47
48  ↳ if not play_pressed and total_time >= delay:
49    >I   anim_time += delta
50  ↳ if anim_time <= title_grow_duration:
51    >I   >I   var t = anim_time / title_grow_duration
52    >I   >I   var scale = lerp(0.1, 1.2, ease_out_cubic(t))
53    >I   >I   title_text.scale = Vector2(scale, scale)
54  ↳ elif not has_shown_menu:
55    >I   >I   show_menu_fade_in()
56    >I   >I   has_shown_menu = true
57

```

Los métodos de los botones son en verdad bastante sencillos, para el de “Jugar” carga la escena de Planet_1, “Opciones” oculta el menú y muestra la configuración y “Salir” termina la ejecución del juego, el resto del código son animaciones y transiciones para hacer el menú más llamativo.

```

76   # Funcion para iniciar el juego
77  ↳ func _on_play_btn_pressed() -> void:
78    >I   play_pressed = true
79    >I   menu_panel.visible = false
80    >I   title_text.visible = false
81    >I   start_play_transition()
82
83   # Transicion entre el menu y el juego
84  > func start_play_transition(): 🎯
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114   # Carga la siguiente escena
115  > func _load_next_scene(): 🎯
116
117
118
119  ↳ func _on_options_btn_pressed() -> void:
120    >I   $Menu_default.visible = false
121    >I   $Configuration.visible = true
122
123  ↳ func _on_exit_btn_pressed() -> void:
124    >I   get_tree().quit()

```

“Configuration” es el menú para configurar las opciones del juego, este menú aparece tanto en el menú principal como en el menú de pausa. Las opciones que tenemos para configurar son el volumen, el tipo de ventana y la resolución.



El sistema de la configuración se divide en 2 scripts, “Configuration_script” y “Configuration_manager”. El primero se encuentra en el nodo de Configuration. Cuando se carga la escena coge los valores de “Configuration_manager” y los aplica a los botones.

```
1  extends Control
2
3  @onready var option_screen = $Screen_Option
4  @onready var option_resolution = $Resolution_Option
5  @onready var slider_volume = $Volume_Slider
6
7  var resolutions = {
8    "720x576": Vector2i(720, 576),
9    "1280x720": Vector2i(1280, 720),
10   "1920x1080": Vector2i(1920, 1080),
11   "2560x1440": Vector2i(2560, 1440)
12 }
13
14 > func _ready() -> void:
15
33
34  func _load_current_settings() -> void:
35    # Volumen
36    slider_volume.value = ConfigurationManager.volume
37
38    # Modo de pantalla
39    select_option_by_text(option_screen, ConfigurationManager.screen)
40    ConfigurationManager.set_screen_mode(ConfigurationManager.screen)
41
42    # Resolución
43    select_option_by_text(option_resolution, ConfigurationManager.resolution)
44    ConfigurationManager.set_resolution(ConfigurationManager.resolution)
```

Cuando se cambia un valor, se llamará a los siguientes métodos, estos envían a “Configuration_manager” los nuevos valores para cada apartado

```

45  func _on_slider_volume_changed(value: float) -> void:
46    ConfigurationManager.set_volume(value)
47    ConfigurationManager.save_settings()
48
49  func _on_option_screen_selected(index: int) -> void:
50    var mode = option_screen.get_item_text(index)
51    ConfigurationManager.set_screen_mode(mode)
52    ConfigurationManager.set_screen_mode(mode)
53    ConfigurationManager.save_settings()
54
55  func _on_option_resolution_selected(index: int) -> void:
56    var res_text = option_resolution.get_item_text(index)
57    ConfigurationManager.set_resolution(res_text)
58    ConfigurationManager.set_resolution(res_text)
59    ConfigurationManager.save_settings()

```

Cada vez que cambiamos la configuración, llamamos a “save_settings”. Este método guarda en un archivo los valores actuales de la configuración para que al iniciar el juego, se mantengan los valores establecidos.

```

36  func save_settings() -> void:
37    var file := ConfigFile.new()
38    file.set_value("audio", "volume", volume)
39    file.set_value("video", "screen", screen)
40    file.set_value("video", "resolution", resolution)
41
42    var err := file.save(CONFIG_PATH)
43    print("Guardando configuración en:", ProjectSettings.globalize_path(CONFIG_PATH))
44    if err != OK:
45      push_error("Error al guardar configuración: %s" % err)
46
47  func load_settings() -> void:
48    print("Cargando configuración desde:", ProjectSettings.globalize_path(CONFIG_PATH))
49    var file := ConfigFile.new()
50    var err := file.load(CONFIG_PATH)
51
52    if err == OK:
53      if file.has_section_key("audio", "volume"):
54        set_volume(file.get_value("audio", "volume"))
55      if file.has_section_key("video", "screen"):
56        set_screen_mode(file.get_value("video", "screen"))
57      if file.has_section_key("video", "resolution"):
58        set_resolution(file.get_value("video", "resolution"))
59    else:
60      print("No se encontró configuración guardada. Creando archivo nuevo.")
61      save_settings()

```

“Configuration_manager” es un script global, en él se guarda las variables de cada configuración y se ejecuta los cambios realizados. Para el volumen se modifica el AudioServer con “set_bus_volume_db”, para la pantalla y la resolución el DisplayServer con “window_set_mode” y “window_set_size” respectivamente.

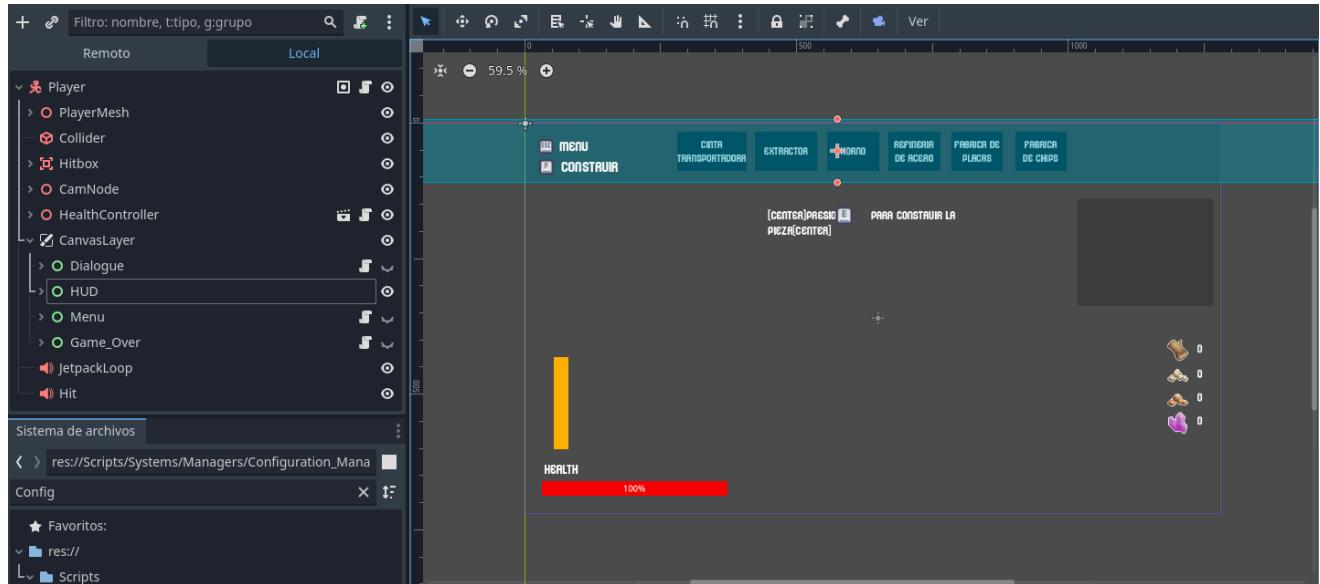
```

 9  func _ready():
10    >I  load_settings()
11
12  func set_volume(new_volume: float) -> void:
13    >I  volume = new_volume
14    >I  AudioServer.set_bus_volume_db(AudioServer.get_bus_index("Master"), linear_to_db(volume / 100.0))
15
16  func set_screen_mode(new_mode: String) -> void:
17    >I  screen = new_mode
18
19  >I  match new_mode:
20    >I  >I  "Ventana":
21      >I  >I  >I  DisplayServer.window_set_mode(DisplayServer.WINDOW_MODE_WINDOWED)
22    >I  >I  "Pantalla completa":
23      >I  >I  >I  DisplayServer.window_set_mode(DisplayServer.WINDOW_MODE_FULLSCREEN)
24    >I  >I  "Sin bordes":
25      >I  >I  >I  DisplayServer.window_set_mode(DisplayServer.WINDOW_MODE_EXCLUSIVE_FULLSCREEN)
26
27  func set_resolution(res_string: String) -> void:
28    >I  resolution = res_string
29
30    >I  var parts = res_string.split("x")
31  >I  if parts.size() == 2:
32    >I  >I  var width = int(parts[0])
33    >I  >I  var height = int(parts[1])
34    >I  >I  >I  DisplayServer.window_set_size(Vector2i(width, height))

```

5.6.2 Interfaz del jugador

La interfaz del jugador se divide en 4 nodos, Dialogue, HUD, Menu y Game_Over



Dialogo

Dialogue es el nodo que muestra el diálogo del jugador, cuando queremos mostrar un diálogo llamamos a “update_message”, lo que hacemos es dividir el texto en cadenas y vamos mostrando los caracteres.

```

72 func _on_type_timer_timeout() -> void:
73     if current_index >= full_text.length():
74         type_timer.stop()
75         is_typing = false
76         return
77
78     # Detectar comandos del tipo [0.05]
79     if full_text[current_index] == "[":
80         var end_index = full_text.find("]", current_index)
81         if end_index != -1:
82             var command = full_text.substr(current_index + 1, end_index - current_index - 1)
83             process_command(command)
84             current_index = end_index + 1
85             return
86
87     # Si no es un comando, mostrar el carácter
88     content.visible_characters += 1
89     current_index += 1
90
91     # Reproducir sonido aleatorio de escritura
92     if (current_index % char_beep_interval == 0) and beeps_list.size() > 0:
93         if not dialog_sound.playing:
94             random_sound = beeps_list[randi() % beeps_list.size()]
95             dialog_sound.stream = random_sound
96             dialog_sound.play()

```

Dentro del texto hay comandos que se ponen para dictar un ritmo en el texto, los comandos se escribe entre corchetes y dentro de ellos se escribe el comando. El comando puede ser [pause=0.5] para determinar una pausa y [0.5] para determinar la velocidad del texto.

```

func _process_command(command: String) -> void:
    #Si es un numero cambia la velocidad de las letras
    if command.is_valid_float():
        type_timer.wait_time = command.to_float()
    #Si es pause pausa el texto por x segundos
    elif command.begins_with("pause="):
        duration = command.replace("pause=", "").to_float()
        type_timer.stop()
        pause_timer.start(duration)

```

Para actualizar el texto usamos `update_message`, que limpia la cadena de comandos antes de mostrarla y ajusta el tamaño y formato del texto.

```

25  func update_message(message: String, text_size: float, text_veloc: float) -> void:
26    is_typing = true
27    full_text = message
28    current_index = 0
29
30    var clean_text = ""
31    var i = 0
32    while i < message.length():
33      if message[i] == "[":
34        var end_index = message.find("]", i)
35        if end_index == -1:
36          clean_text += message[i]
37          i += 1
38        continue
39        i = end_index + 1 # saltar comando
40      else:
41        clean_text += message[i]
42        i += 1
43
44    content.text = clean_text
45    content.visible_characters = 0
46    content.add_theme_font_size_override("normal_font_size", text_size)
47    type_timer.wait_time = text_veloc
48    type_timer.start()

```

Junto a “Dialog_script” tenemos también “Dialog_manager”, este se encarga de guardar todos los diálogos del juego y actualizar el mensaje que se debe mostrar según la fase en la que esté.

Los diálogos se guardan en una lista numerada, la enumeración representa la fase y cada fase tiene varias líneas de diálogo. Aquí se aprecia mejor el uso de los comandos del texto, cambiando la velocidad y pausando en ciertos puntos para hacer el diálogo más interactivo.

```

1  extends Node
2  signal dialog_finished
3
4  var dialogue_text := {
5    1: [
6      "[0.04]Oh no!!![pause=0.3] Ha explotado la estación!![pause=0.4] Bueno...[pause=0.3] espero q
7      "[0.04]De todas formas tengo que encontrar una manera de salir de aqui,[pause=0.2] será mejor
8      "[0.04]Debería de empezar por la base"
9    ],
10   2: [
11     "[0.04]Con una buena base todo debería salir bien...[pause=0.3][0.05] eso espero.",
12     "[0.04]Tengo que buscar más materiales para hacer los motores"
13   ],
14   3: []
15   "[0.04]Guay! Parece ser suficiente para salir del planeta... creo que aun le falta algo",
16   "[0.04]Ya sé, le añadiré unas alas para mayor aerodinámica",
17 }

```

En “show_dialogues_for_phase” pasamos el número de la fase, activamos el nodo del diálogo y actualizamos el texto. “_hide_dialogue” esconde el diálogo cuando ya hemos mostrado todas las líneas.

```

    ↴ func show_dialogues_for_phase(phase: int) -> void:
    ↴ | if not dialogue_text.has(phase) or dialogue_node == null:
    ↴ | | return

    ↴ | is_dialogue_active = true
    ↴ | dialogue_list = dialogue_text[phase]
    ↴ | dialogue_index = 0
    ↴ | isShowing = true
    ↴ | dialogue_node.show()
    ↴ | _update_text()

    ↴ func _hide_dialogue():
    ↴ | if dialogue_node:
    ↴ | | dialogue_node.hide()
    ↴ | isShowing = false
    ↴ | is_dialogue_active = false
    ↴ | emit_signal("dialog_finished")

```

En _process, para pasar al siguiente diálogo es necesario que el jugador mande un input manualmente, es decir, que cuando se presione clic izquierdo se llamará al método “on_click_advance”.

```

42 ↴ func _process(delta: float) -> void:
43 | if is_instance_valid(dialogue_node) and Input.is_action_just_pressed("left_click"):
44 | | DialogManager.on_click_advance()

```

“On_click_advance” tiene doble funcionalidad, si se hace click mientras se está escribiendo el texto, se omitirá la animación y mostrará el texto entero, si ya se ha completado pasará al siguiente diálogo.

```

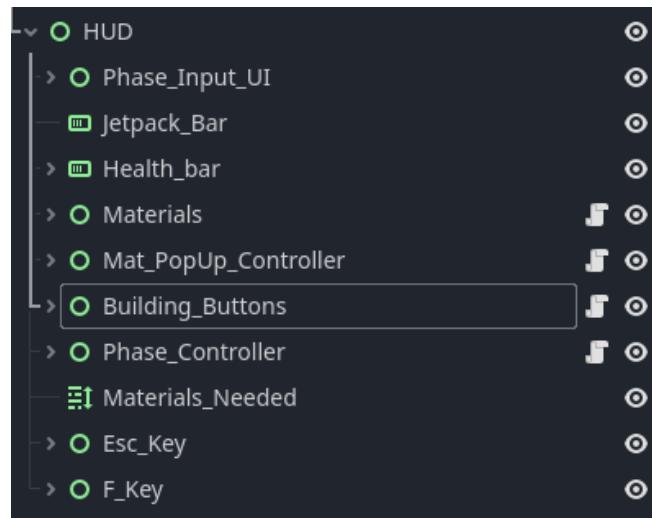
    ↴ func on_click_advance():
    ↴ | if not isShowing:
    ↴ | | return

    ↴ | # Si está escribiendo, mostrar todo el texto directamente
    ↴ | if dialogue_node.get("isTyping"):
    ↴ | | dialogue_node.call("skipTyping")
    ↴ | else:
    ↴ | | dialogue_index += 1
    ↴ | | _update_text()

```

HUD

La interfaz del jugador tiene varios nodos, algunos son estáticos, no tienen funcionalidad y solo sirven para informar al jugador. Pero luego hay otros que sí tienen código.



“Material_Manager” en Materials muestra el inventario actual del jugador, accediendo a los valores de BuildManager.

```

func _ready() -> void:
    update_mat_UI()

func update_mat_UI():
    for mat in get_children():
        var mat_name = mat.name
        if BuildManager.materiales.has(mat_name):
            var amount = BuildManager.materiales[mat_name]
            if amount >= 0:
                mat.show() # o mat.visible = true
                var label = mat.get_node("Amount") as RichTextLabel
                if label:
                    label.text = str(amount)
            else:
                mat.hide()
        else:
            mat.hide()

```

“PopUpManager” dentro de “Mat_PopUp_Controller” es el que usamos en BuildManager para mostrar los materiales que el jugador ha obtenido o consumido.

El popup es un prefab de una etiqueta que llamamos desde una posición, cuando aparece le escribimos el valor del material, el color según lo que haya conseguido y subirá hacia arriba hasta pasado 0.2 segundos

```

1  extends Node
2
3  @export var popup_scene: PackedScene
4  @export var popup_spawn_point: Node2D
5
6  var popup_queue: Array = []
7  var processing_queue := false
8
9  ↘ func show_material_popup(text: String, color: Color = Color.WHITE) -> void:
10    >I  popup_queue.append({"text": text, "color": color})
11  ↘>I  if not processing_queue:
12    >I  >I  _process_queue()
13
14  # Corrutina para procesar la cola con retardo
15  ↘ func _process_queue() -> void:
16    >I  processing_queue = true
17  ↘>I  while popup_queue.size() > 0:
18    >I  >I  var entry = popup_queue.pop_front()
19    >I  >I  var popup = popup_scene.instantiate() as Label
20    >I  >I  popup.text = entry.text
21    >I  >I  popup.add_theme_color_override("font_color", entry.color)
22    >I  >I  popup.position = Vector2.ZERO # En el origen local de popup_spawn_point
23    >I  >I  popup_spawn_point.add_child(popup) # Se instancia como hijo directamente
24
25    >I  >I  var anim = popup.get_node("AnimationPlayer")
26    >I  >I  anim.play("float_up")
27
28    >I  >I  await get_tree().create_timer(0.2).timeout
29    >I  processing_queue = false

```

“Build_buttons_controller” situado en “Building_buttons” se encarga de controlar los botones de las construcciones.

Los botones de las construcciones van apareciendo según vaya avanzando las fases, este comportamiento se define en la siguiente lista:

```

1  extends Control
2
3  ↘ @onready var unlocks_for_phase := {
4    >I  1: [$BtnBuildExtractor, $BtnBuildConveyLine,$BtnBuildFurnace],
5    >I  2: [$BtnBuildSteelRefinery],
6    >I  3: [$BtnBuildPlatesFactory],
7    >I  4: [$BtnBuildChipsFactory]
8  }
9

```

La enumeración es idéntica a como funciona de diálogos, con lo que cuando se llame a “_on_phase_change”, desbloqueará el botón que esté en el número de la fase actual.

```

16  func _on_game_manager_initialized():
17    >I   GameManager.connect("phase_changed", Callable(self, "_on_phase_changed"))
18    >I   _on_phase_changed(GameManager.current_phase)
19
20  func _on_phase_changed(new_phase: int) -> void:
21    >I   for fase in unlocks_for_phase.keys():
22      >I   >I   var visible : bool = fase <= new_phase
23    >I   >I   for btn in unlocks_for_phase[fase]:
24      >I   >I   >I   btn.visible = visible
25

```

Cada botón tiene un método que llama a su respectiva construcción en BuildManager, la cual cargará en CurrentSpawnable el prefab correspondiente.

```

→ 26  func _on_btn_build_extractor_button_down() -> void:
27    >I   BuildManager.SpawnExtractor()
28
→ 29  func _on_btn_build_convey_merger_button_down() -> void:
30    >I   BuildManager.SpawnConveyMerger()
31
→ 32  func _on_btn_build_furnace_button_down() -> void:
33    >I   BuildManager.SpawnFurnace()
34
35  func _on_btn_build_storage_button_down() -> void:
36    >I   BuildManager.SpawnStorage()
37
→ 38  func _on_btn_build_steel_refinery_button_down() -> void:
39    >I   BuildManager.SpawnSteelRefinery()
40
41  func _on_btn_build_cristal_refinery_button_down() -> void:
42    >I   BuildManager.SpawnCristalRefinery()
43
→ 44  func _on_btn_build_chips_factory_button_down() -> void:
45    >I   BuildManager.SpawnChipsFactory()
46
→ 47  func _on_btn_build_plates_factory_button_down() -> void:
48    >I   BuildManager.SpawnPlatesFactory()

```

Y por cada botón, hay una función “_on_mouse_enter” que muestra en un texto información sobre esa construcción.

```

→ 59  func _on_btn_build_convey_line_mouse_entered() -> void:
60    >I   $Building_Info.text = "[center]Mueve materiales entre fábricas[center]"
61
→ 62  func _on_btn_build_extractor_mouse_entered() -> void:
63    >I   $Building_Info.text = "[center]Extrae materiales de depósitos[center]"
64
→ 65  func _on_btn_build_furnace_mouse_entered() -> void:
66    >I   $Building_Info.text = "[center]Fundé los metales en lingotes[center]"
67
→ 68  func _on_btn_build_steel_refinery_mouse_entered() -> void:
69    >I   $Building_Info.text = "[center]Fundé lingotes de hierro y cobre para hacer acero[center]"
70
→ 71  func _on_btn_build_plates_factory_mouse_entered() -> void:
72    >I   $Building_Info.text = "[center]Convierte los lingotes en placas[center]"
73
→ 74  func _on_btn_build_chips_factory_mouse_entered() -> void:
75    >I   $Building_Info.text = "[center]Crea chips a partir de placas de acero, placas de cobre y cristal[center]"
76
→ 77  func _on_btn_line_mouse_exited() -> void:
78    >I   $Building_Info.text = ""
79

```

Menu

El menú es la interfaz que aparece cuando le damos a ESC. Aquí se muestran todos los controles del juego, junto a varias opciones. Lo que hace mayormente el código es gestionar la visibilidad de los menús y la visibilidad de la interfaz. (`get_tree().paused` pausa la escena actual)

```

1  extends Control
2
3  var active = false
4
5  ↳ func _process(delta: float) -> void:
6    ↳ if Input.is_action_just_pressed("Menu") and GameManager.currentState != GameManager.State.Die:
7      ↳   active = !active
8
9    ↳   if $Configuration.visible or $Exit_Notify.visible:
10      ↳     "$Menu_Default".visible = active
11      ↳     $Configuration.visible = !active
12      ↳     $Exit_Notify.visible = !active
13
14    ↳     ".../HUD".visible = !active
15    ↳     visible = active
16    ↳     get_tree().paused = active
17

```

Si le damos a "Continuar", se cerrará el menú y volveremos al juego.

```

→ 18  ↳ func _on_resume_btn_pressed() -> void:
19    ↳   active = false
20    ↳   visible = false
21

```

Si le damos a "Opciones", nos mostrará el mismo menú de opciones como en el menú principal.

```

→ 22
→ 23  ↳ func _on_options_btn_pressed() -> void:
24    ↳   $Menu_Default.visible = false
25    ↳   $Configuration.visible = true
26

```

Si le damos a "Salir" nos saldrá una notificación preguntando si de verdad queremos salir antes de volver al menú principal

```

→ 27
→ 28  ↳ func _on_exit_btn_pressed() -> void:
29    ↳   $Menu_Default.visible = false
30    ↳   $Exit_Notify.visible = true
31
32  ↳
33
→ 34  ↳ func _on_confirm_btn_pressed() -> void:
35    ↳   get_tree().paused = false
36    ↳   get_tree().change_scene_to_file("res://Scenes/Main_Menu.tscn")
37
38  ↳ func _on_cancel_btn_pressed() -> void:
39    ↳   $Menu_Default.visible = true
40    ↳   $Exit_Notify.visible = false
41

```

Game Over

La pantalla de game over aparecerá cuando la vida del jugador llegue a 0. Aparecerá con un fundido y pausará la escena. Al pulsar el botón “Volver al menú” se retomará la ejecución de la escena y volveremos al menú principal.

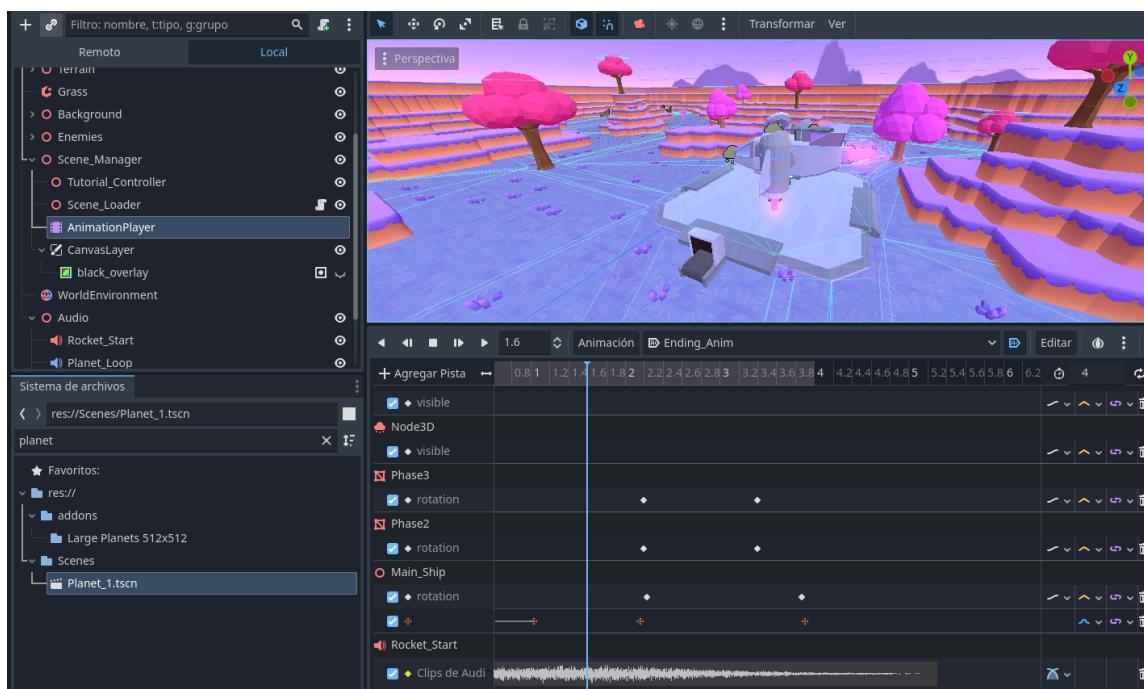
```

1  extends Control
2
3  func _process(delta: float) -> void:
4    if GameManager.currentState == GameManager.State.Die:
5      get_tree().paused = true
6
7  func show_game_over():
8    modulate.a = 0
9    visible = true
10
11   # Fade-in desde negro
12   var tween := create_tween()
13   tween.tween_property(self, "modulate:a", 1.0, 1.2).set_trans(Tween.TRANS_SINE).set_ease(Tween.EAS
14
15   # Esperar a que termine el fade-in antes de continuar
16   await tween.finished
17
18   GameManager.currentState = GameManager.State.Die
19
20  func _on_confirm_btn_pressed() -> void:
21    get_tree().paused = false
22    GameManager.currentState = GameManager.State.Play
23    get_tree().change_scene_to_file("res://Scenes/Main_Menu.ts scn")
24

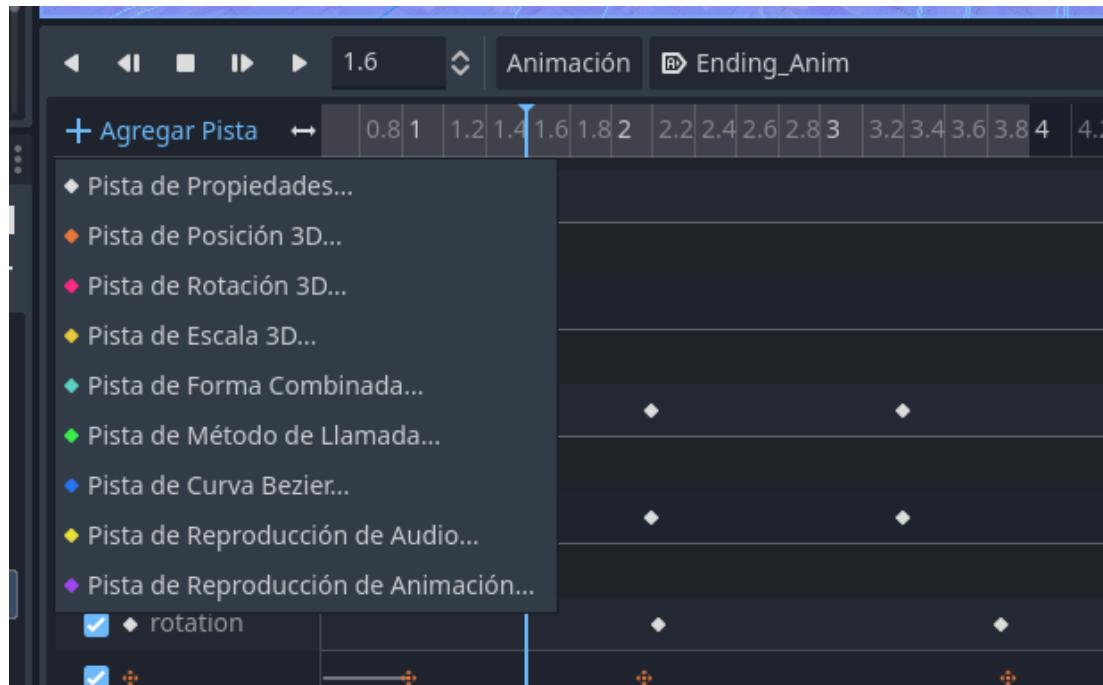
```

5.7 Cinemáticas

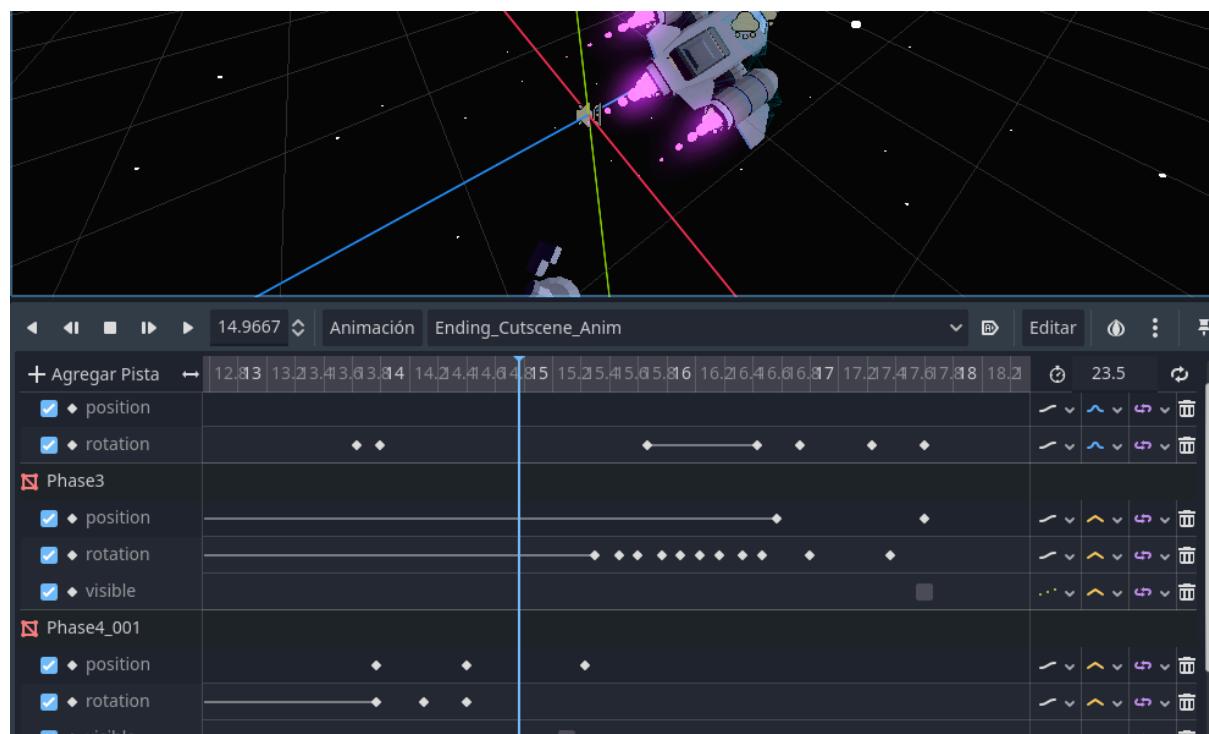
He creado cinemáticas para el final del juego, estas se ejecutan mediante un AnimationPlayer.



Para hacer una animación debemos de crear una pulsando el botón “Animación > Nuevo”, luego le daremos a “Añadir pista” para poder modificar los valores de los elementos que queremos animar. Entre los valores que podemos alterar están la posición, rotación y escala, visibilidad, llamada a métodos y reproducción de audio u otras animaciones.



Luego de añadir las pistas, simplemente tendremos que ir moviendo y ajustando el objeto y creando claves para guardar su estado en la línea de tiempo. En el código llamaremos a esta animación con `"animation_player.play("Ending_Cutscene_Anim")"`



6. Mayores dificultades encontradas

6.1 El modo construcción

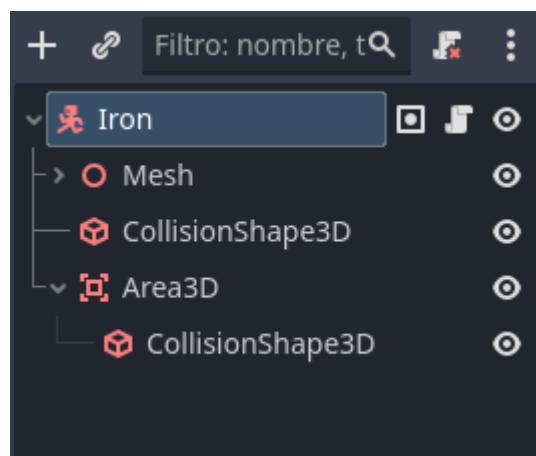
6.1.1 Materiales

Al ser la mecánica más compleja, es con la que he tenido mayores problemas. Durante todo el desarrollo siempre he tenido problemas para mover los materiales.

En parte era porque no sabía bien que nodo usar para los materiales, primero fueron Staticbodys los cuales realizaban una animación para moverse, pero no me convenció ya que el movimiento era muy pausado.

Luego probé con Rigidbodys aplicando una fuerza en la dirección de la cinta, aunque el movimiento era más fluido no podía controlarlo muy bien y los materiales se salían de las cintas o no paraban donde debían.

Al final opté por un CharacterBody, es más fácil de controlar y se mueve de forma fluida por las cintas, aunque usarlo solo para los materiales no es muy recomendable, pero a este punto funcionaba y por el tamaño del juego no iba a influir mucho.



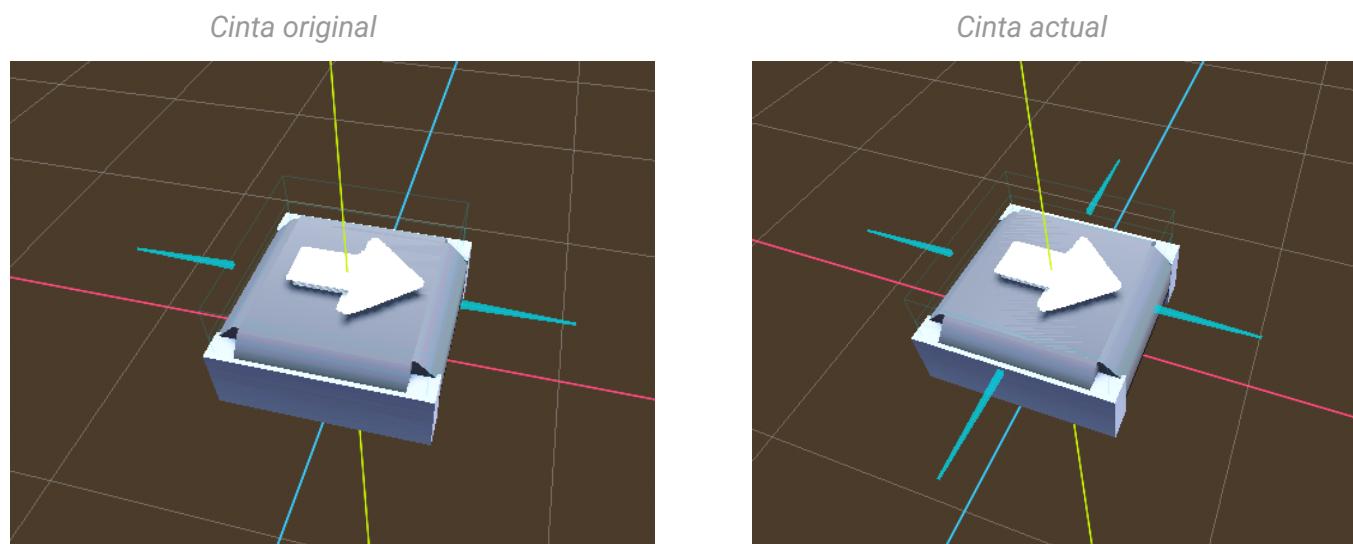
6.1.2 Cintas

Mi idea original con las cintas era que hubiera de varios tipos, habiendo primero la cinta normal, la cual sigue una dirección fija, luego el "Merger", que sería una cinta para unir varias entradas, y el "Splitter", que dividiría la salida del material.

Cuando hice la primera cinta esta no funcionaba muy bien, tenia 1 collider para detectar las cintas de delante pero no funcionaban siempre, asi que los cambié a 2 Raycast, uno por delante y otro por detrás.

A pesar de la mejora, las cintas no se detectaban bien y el material no avanzaba bien. Así que me puse a desarrollar el merger y luego vería que hacer con la cinta normal. Al crear el merger me di cuenta que este funcionaba mejor que la propia cinta al tener 2 raycast más en los lados.

Así es como solucioné un problema simplemente pasando a otra cosa.



6.1.2 Fábricas

No hubo mucho problema al desarrollar las fábricas, pero si hay un problema relacionado con estas. Resulta que los nodos de godot, independientemente si son StaticBodys o CharacterBodys, van a dejar de funcionar cuando la cámara no lo está cargando.

Por esta razón cuando el jugador se aleja de una fábrica y luego vuelve, esta tendrá el material atascado en la salida y no se moverá. Es un problema bastante frustrante ya que paraba la producción y ralentizaba el gameplay.

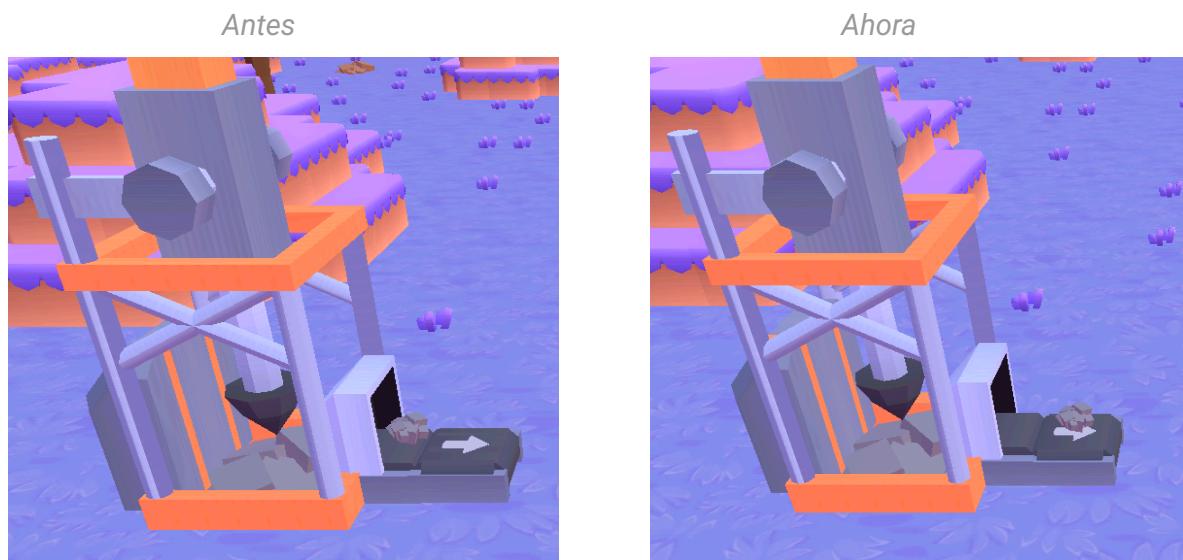


Intenté solucionarlo de varias maneras, forzando el funcionamiento de las fábricas y las cintas fuera de cámara, cambiando de nuevo el tipo de material, que el material desapareciera cuando entra en cámara... Incluso hice que las fábricas no funcionarán fuera de cámara, pero eso empeoró la experiencia de juego.

Así que... para solucionarlo lo único que hice fué mover el punto de salida de todas las fábricas un poco hacia adelante. La razón de esto es porque antes las fábricas tenían su salida más hacia adentro, eso dejaba el material más atrás de la cinta que se conectaba.

Esto impedía que las fábricas estuvieran a 1 cinta de distancia entre ellas, porque el material no llegaba a la entrada y no se iba a mover si no había al menos 2 cintas conectadas.

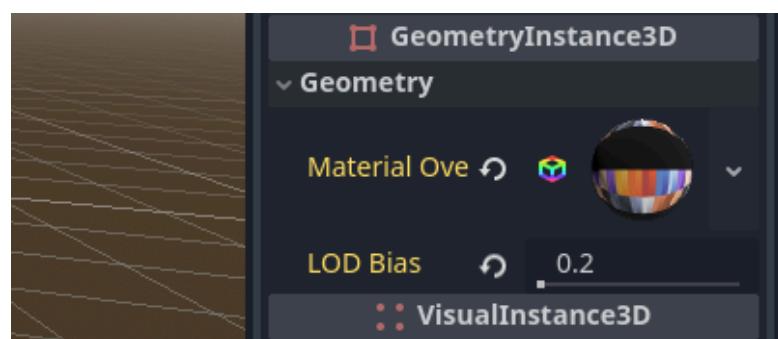
De esta forma, el material puede entrar directamente en la fábrica y no bloqueaba la salida... siempre y cuando haya una cinta de distancia.



6.2 Rendimiento

El elemento que más recursos consumía en el juego era el terreno. Esto se debía a que tenía varias capas de terreno, había muchas zonas que eran innecesarias ya que no se podían ver pero afectan al rendimiento.

Además de quitar el terreno innecesario también reducí el LOD (Level Of Detail) de todos los objetos del escenario. De por si no tienen muchos polígonos, pero al haber muchos tenía que reducirlo, por suerte no afectó mucho visualmente.



6.3 Github

La exportación del proyecto a github dio algunos problemas, ya que a veces al mover o eliminar archivos me llevaba por delante algún recursos necesarios para el repositorio. Debido a esto tuve que crear al menos 2 repositorios porque había eliminado sin querer los archivos de git.

6.4 Exportación del juego

Hay un error que no sé exactamente porqué sucede pero, cuando exporté el juego en un ejecutable, le puse un ícono. Al principio no cargaba correctamente y solo aparecía el logo por defecto, pero cuando lo exporté sin querer con 2 puntos (..) ahora si que aparecía.

Si intento quitar el punto sobrante el ícono desaparece, así que lo tuve que dejar así.

Nombre	Fecha de modificación
Overcomplicate it!..console.exe	10/06/2025 14:59
Overcomplicate it!..exe	10/06/2025 14:59
Overcomplicate it!..pck	10/06/2025 14:59

7. Conclusión

Este proyecto ha sido muy útil para mi, ya que nunca había hecho un juego como tal de principio a fin.

Tengo que decir que en el aspecto de diseño considero que he cometido algunos errores, el género que he elegido no es el más adecuado para empezar, son juegos que suelen ser juegos más complejos, con muchas recetas y fábricas y materiales. El mío al ser más simple no espera competir con esos títulos.

Además, en mi vida he hecho yo un juego de gestión, el sistema que he creado es muy bruto y poco eficiente. Hay muchos sistemas de optimización que otros juegos del género usan que son bastante impresionantes, pero debido al tiempo que tenía y de la escala que he decidido tomar, no es tan grave, pero en el fondo creo que podría haberlo hecho mejor si hubiera tenido más tiempo.

También está el mensaje del juego, el “sobre complicar una tarea que es más sencilla de lo que parece” es algo que se me ocurrió a mediados del desarrollo. Es una idea que me siento muy identificado, se puede apreciar apreciar bastante por todos los sistemas que he creado para el juego.

Mi idea resumida es que he hecho más de lo que debería, pero estoy contento de haberlo hecho.

8. Otros materiales de apoyo

9. Biografía

//Modelo del jugador

<https://github.com/gdquest-demos/godot-4-3D-Characters?tab=readme-ov-file>

//Sistema de dialogo

<https://wordeater-dev.itch.io/bittersweet-birthday/devlog/224241/howto-a-simple-dialogue-system-in-godot>

//Textura planetas

<https://screamingbrainstudios.itch.io/2d-planet-pack-2>

//Fondo espacio

<https://screamingbrainstudios.itch.io/seamless-space-backgrounds>

//Musica

<https://freesound.org/>

//Sonidos

<https://dmochas-assets.itch.io/dmochas-bleeps-pack?download>

<https://upbeat.io/sfx/category/gaming>