Engraginy: Simulació de Sistemes de Transmissió Mecànica

Biel Alavedra Busquet

January 27, 2026

Resum– Engraginy és un videojoc 3D que busca simular mecàniques de transmissió mecànica, fent ús d'engranatges, eixos, i altres mecanismes amb l'objectiu de construir cadenes de producció i fàbriques. El nucli del projecte és el sistema de potència i transmissió mecànica, basat en grafs que fa ús de càlcul vectorial i senyals per actualitzar el sistema quan es requereix. A més d'un sistema de transport de materials basat en simulacions físiques. Tots aquests sistemes dissenyats i pensats per oferir una experiència totalment reactiva. **Paraules clau–** Godot, Videojoc, Grafs, GDScript, Logística, Simulació 3D.

Abstract– Engraginy is a 3D video game that seeks to simulate mechanical transmission mechanics, utilizing gears, shafts, and other mechanisms with the goal of building production chains and factories. The core of the project is the power and mechanical transmission system, based on graphs that use vector calculation and signals to update the system as required. Additionally, it features a material transport system based on physical simulations. All these systems are designed and intended to offer a fully reactive experience.
Keywords– Godot, Videogame , Graphs, GDScript, Logistics, 3D Simulation.

1 Introducció - Context del treball

El projecte Engraginy neix de la voluntat d'explorar mecàniques de joc més físiques dins del gènere de l'automatització. Mentre que títols referents com *Factorio* i *Satisfactory* utilitzen l'electricitat com un recurs binari o simplificat, aquest treball proposa una xarxa de potència mecànica on cada component (engranatges, eixos, cintes) afecta el rendiment de les cadenes de producció segons les lleis de la cinemàtica bàsica. L'objectiu principal és la creació d'un sistema modular i extensible en el motor Godot 4 que permeti al jugador construir xarxes de potència complexes. Per aconseguir aquest objectiu s'haurà de dissenyar, implementar i testejar tot un videojoc que compleixi amb el següent llistat de requisits:

- Joc en primera persona del gènere automatització
- Menú de construcció que mostri les opcions possibles
- Sistema de càrrega i guardat de partida
- Sistema de construcció basat en graella
- Sistema de simulació de transmissió mecànica
- Sistema de transport d'objectes

El repte tecnològic a afrontar resideix en la propagació dinàmica de la rotació en un entorn 3D, requerint l'ús de càlculs vectorials per determinar la viabilitat de les connexions, i les direccions de transmissió.

2 Motivació

La motivació principal per al desenvolupament d'aquest projecte és aprendre com funciona el desenvolupament d'un videojoc. Aprendre i estudiar el cicle de vida i les diferents fases necessaries per a conseguir un projecte complet. Els meus referents a l'hora de desenvolupar el projecte són videojocs com "Factorio"[5], "Satisfactory"[6], "Dyson sphere program"[7], "Shapez"[8], i "Minecraft: Create Mod"[9] (una modificació del joc no oficial).

D'aquests referents amb els quals tinc més experiència són DSP i Satisfactory, dos videojocs que m'han sorpres positivament per la seva capacitat d'inmersió i estimulació lògica. M'han mostrat que entretingut i mentalment activador pot arribar a ser el gènere. El DSP és un joc molt més ambiciós, es comença en un petit satèl·lit dins d'un cúmul d'estrelles generat aleatòriament, i acaba quan el jugador construeix una esfera de Dyson i viatja entre sistemes solars. Tot això amb una perspectiva isomètrica que dona molt control i molta visió de tot allò que construeixes. En el Satisfactory l'objectiu és completar un ascensor espacial. Es comença en un planeta (el qual sempre és el mateix, el mapa és prefixat). Aquest joc és en primera persona, i totes les màquines que es construeixen són gegantines. Això impedeix una visió completa d'allò que construeixes. En quantitat de màquines les construccions

• E-mail de contacte: biel.alavedra@gmail.com
• Menció: Computació
• Treball tutoritzat per: Enric Marti Godia
• Curs 2025/26

són sempre més petites i molt més lentes de construir.

L'objectiu es desenvolupar un sistema amb mecàniques més similars a Satisfactory, ja que aquesta visió en primera persona dona un control més granular de les construccions a petit nivell, i encaixa molt millor amb les mecàniques que vull incorporar de Minecraft: Create mod, en la majoria dels jocs de l'estil una part molt important és la generació d'energia, es molt ineficient fer fàbriques més grans si no es tenen els recursos per mantenir-les, tant DSP, Factorio i Satisfactory utilitzen energia elèctrica, sigui solar, eòlica, carbó o nuclear. Tots aquests generadors s'acaben connectant a la xarxa elèctrica per donar energia a totes les màquines. En canvi, el Minecraft: Create mod requereix que tot sigui alimentat per energia cinètica, fent ús d'eixos, engranatges grans i petits, cadenes, corretges de transmissió... Com el meu objectiu és fer un joc amb una ambientació prerevolució industrial, i no del tipus fantàstic, crec que aquesta és la millor forma de fer-ho. També això afegeix totes les mecàniques d'haver de connectar les màquines, no només a una velocitat concreta, ja quealgunes seran direccionals, com les cintes per transportar material.

D'entre tots els gèneres de videojocs perquè he escollit automatització? Considero que aquest és un gènere que segueix molt la filosofia d'un programador, o en general la de tots els enginyers: dividir i vèncer. L'inici d'aquests jocs és senzill, hi ha unes poques màquines i cal fer processos simples. Per exemple, amb un extractor de recursos automàtic i cal processar els recurso s en brut per tal d'obtenir el recurs processat (de mena de ferro a lingots de ferro). Però això ràpidament canvia, agafant d'exemple el joc DSP. Si vols fer una placa de circuits es necessita un forn de fosa que converteixi el mena de ferro a plaques de ferro, un altre forn de fosa que converteixi coure en brut a plaques de coure i un assemblador que agafi aquests dos materials i els converteixi en la placa de circuits. Això és només un dels primers passos, on cada vegada es disposa de més i més materials diferents, processos diferents i els objectes requerits són cada vegada més complexes. Per força no es poden afrontar tots aquests problemes simultàniament, cal dividir els problemes en mòduls que es puguin replicar cada vegada que calgui per a resoldre aquell problema.

Dins de la comunitat de fans d'aquest gènere de videojoc hi ha molts enginyers i gent a que agrada molt optimitzar processos i fer construccions el màxim d'eficients possibles. Per tot això no només considero el gènere com a entretingut, sinó que també en certa manera és educatiu, ja que força al jugador tant sí com no a organitzar-se, pensar formes eficients de construir coses, com connectar les entrades i sortides de totes les fàbriques, calcular que tanta entrada es necessita per la sortida objectiva i construir d'acord amb això. Per completar un joc d'aquests cal ser organitzat, metòdic i pensar en solucions segons el problema que es tingui.

3 DESENVOLUPAMENT I ESTRUCTURA

El programari s'ha estructurat seguint els principis de la programació orientada a objectes (POO) mitjançant el sistema de nodes de Godot. La classe base `Building` defineix els comportaments d'interacció, mentre que `PowerNode` hereta aquesta base per incloure la lògica de ports i connexions. Engranatges, eixos, generadors i màquines hereten d'aquest `PowerNode` per tal de definir més concretament les propietats específiques de cada cas.

En la figura, Fig. 1, es pot observar un diagrama de mòduls amb les classes més importants del projecte, el Jugador controla directament la càmera, el moviment del model de jugador, i la interfície d'usuari, que conté els menús per seleccionar i col·locar objectes. Els objectes col·locats per l'usuari, `Building` tots aniran acompanyats d'un context component, aquest permet la interacció (si així ho necessita) i la posterior eliminació de l'edifici si així ho vol el jugador. Aquests `Building` podran ser `PowerNode`, elements de la xarxa de transmissió mecànica que detallaré més endavant. Els `PowerNode` seran gestionats per al `PowerGridManager` i tots contindran una sèrie de `PowerNodePort`, per gestionar les connexions entre ells.

3.1 Conceptes de Godot: Nodes, Escenes i Senyals

Previament a la descripció técnica del projecte és necesari explicar el funcionament de tres conceptes clau: Nodes, Escenes i Senyals. Aquests termes apareixeran recurrentment al llarg de l'artícle. A l'utilitzar Godot l'estructura interna del projecte ha de seguir els requisits del motor. Això implica fer ús del sistema de nodes i escenes per crear tots els objectes, i aprofitar les senyals per simplificar connexions entre objectes.

3.1.1 Nodes

Cada element del joc és un node amb unes propietats i funcions diferents. Hi ha tres famílies principals: nodes 3D, nodes 2D i nodes de "GUI" (Interfície d'Usuari).

Dins de la família de nodes 3D, que són els més rellevants per la simulació, tenim exemples com:

- Camera3D
- StaticBody3D
- MeshInstance3D (per als models visuals)
- CollisionShape3D (per a les físiques)
- RayCast3D (per a detectar objectes)

Tots aquests nodes hereten de la classe base "Node3D". D'aquesta classe obtenen les propietats essencials de qualsevol objecte a l'espai: la posició, la rotació i l'escala.

Els nodes de "GUI" ens permeten la creació interfícies d'usuari on s'utilitza l'estructura jeràrquica per encapsular cada element d'interfície com ara:

- Container
- Label
- TextureRect
- Button
- Panel
- ItemList
- etc.

Els nodes 2D no són rellevants per al projecte, però molts d'ells tenen equivalents en els nodes 3D.

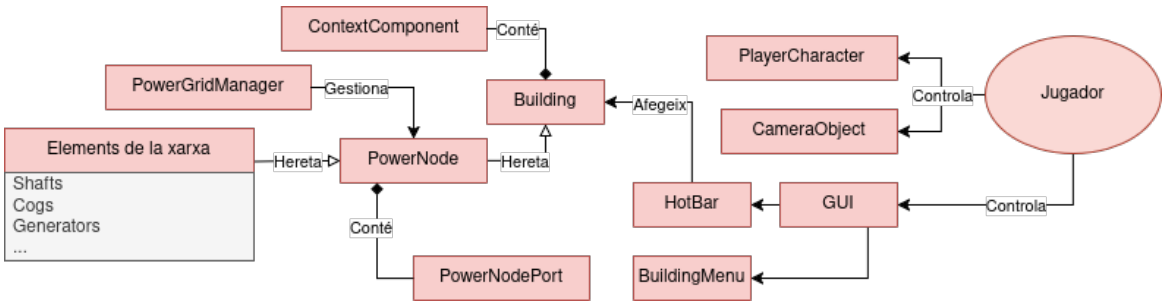

Fig. 1: Diagrama de mòduls

3.1.2 Escenes

Les escenes seran definides per l'usuari i són una col·lecció de nodes de qualsevol classe, una escena pot ser el jugador, la interfície d'usuari o un nivell sencer d'un joc. L'organització i contingut d'aquestes escenes respon a les decisions de disseny i arquitectura del programari. Aquestes escenes es poden instanciar dins d'altres escenes, la qual cosa permet dividir problemes grans en mòduls més petits i manejables.

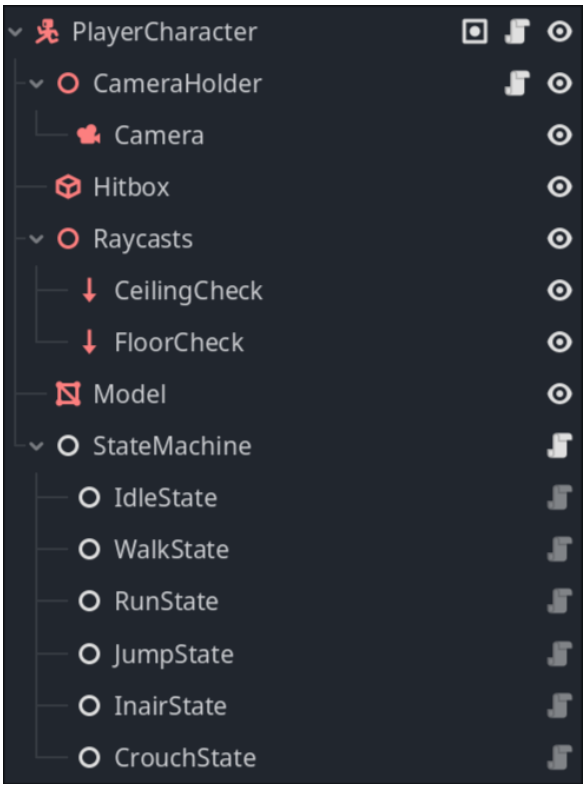

Fig. 2: Escena del jugador amb la jerarquia de nodes

En l'escena que conté el jugador, Fig. 2, tenim com a node pare un "PlayerCharacter" (node propi, hereta de la classe base "CharacterBody3D"), amb 5 fills, el punt d'ancoratge de la càmera ("CameraHolder"), la caixa de col·lisions del personatge ("Hitbox"), els "RayCast3D" ("Raycasts"), el "Model" i la màquina d'estats encarregada del control del personatge. Aquesta màquina d'estats està iniciada amb nodes buits, aquests nodes són la classe base per a tots els objectes de Godot, i no tenen propietats especials. Podem veure que en el lateral tenim una icona amb un pergamí, això ens indica que hi ha un script vinculat al node, si està en gris vol dir que aquest script és heretat, compartit amb altres nodes de la mateixa classe. Si és blanc ens indica que el node té un script únic.

3.1.3 Senyals

Els senyals permeten connectar diferents nodes i escenes sense necessitat de conèixer exactament l'estructura de l'escena ni haver de fer referència al node directament. Aquests senyals són missatges que els nodes poden emetre; els altres nodes es poden connectar a aquest senyal i executar una funció quan reben el missatge. Aquests missatges es poden emetre directament per codi, poden respondre a accions del jugador o a canvis en l'estructura de l'escena. Aquest sistema redueix molt les dependències i fa el codi més modular, ja que s'eviten referències directes entre escenes. Per explicar com utilitzo aquest sistema posaré com a exemple el mecanisme que he desenvolupat per interactuar amb objectes.

En aquest fragment de codi, Fig. 3, es defineixen les funcions utilitzades per emetre els tres senyals d'interacció amb l'entorn: `unfocused`, `focused` i `interacted`. La funció `interact_cast()` s'executa a cada fotograma del joc per comprovar si estem interactuant amb algun objecte. El primer pas és obtenir l'objecte que el jugador està mirant directament. Per fer-ho, emetem un `RayCast3D` des de l'origen de la càmera fins a una certa distància i retornem el primer objecte amb el qual col·lideix mitjançant la funció `make_cast_query()`.

Si l'objecte retornat és el mateix que en el fotograma anterior, no fem res. En cas contrari, verifiquem si en l'objecte anterior teníem definit el senyal `unfocused`. Si no estigués definit, vol dir que amb aquell objecte no es podia interactuar; si hi és, emetem el senyal. Acte seguit, si el nou objecte que estem mirant té el senyal `focused`, l'emetem per indicar que ha rebut el focus.

D'altra banda, la funció `interact()` només s'executa quan el jugador prem el botó d'interactuar. El procediment és similar: comprovem que l'objecte tingui el senyal `interacted` i, en cas afirmatiu, l'emetem.

Per gestionar la resposta a aquests senyals, els objectes interactius disposen d'un node fill anomenat `InteractionComponent`. Aquest conté l'script encarregat de processar els senyals i cridar les funcions adients de l'objecte pare. Aquest component ha de ser fill d'un node `StaticBody3D`, ja que la funció `make_cast_query()` requereix cossos físics per detectar la interacció. Finalment, en l'objecte `InteractionComponent` senzillament busquem el node pare i connectem aquests senyals amb les funcions `in_range`, `not_in_range` i `on_interact` definides en

```gdscript
func interact_cast() -> void:
    var current_cast_result : CollisionObject3D = make_cast_query()
    if current_cast_result != interact_cast_result:
        if interact_cast_result and interact_cast_result.has_user_signal("unfocused"):
            interact_cast_result.emit_signal("unfocused")
        interact_cast_result = current_cast_result
        if interact_cast_result and interact_cast_result.has_user_signal("focused"):
            interact_cast_result.emit_signal("focused")
    pass

func interact() -> void:
    if interact_cast_result and interact_cast_result.has_user_signal("interacted"):
        interact_cast_result.emit_signal("interacted")

    pass
```

Fig. 3: Fragment de codi de l'objecte PlayerCharacter que conté les funcions per emetre senyals

la lògica de l'objecte.

3.2 Lògica de Transmissió Mecànica

El sistema de transmissió mecànica és una adaptació del sistema existent a Minecraft: Create. Per tant, s'ha optat per simularne el comportament, a continuació faré un llistat de les especificacions del sistema, i com s'hauria de comportar sota cada cas específic.

- Tota connexió d'un eix ha de mantenir direcció i velocitat.
- Tota connexió d'un engranatge ha d'invertir la direcció.
- Connectar un engranatge petit a un gran a través de les dents de l'engranatge ha de augmentar la velocitat, a l'invers la velocitat es redueix.
- La xarxa no pot superar el límit d'energia subministrada per tot el conjunt de generadors connectats.
- En cas de superar el límit tot el sistema s'ha d'aturar a l'instant.
- En cas de tornar a estar per sota del límit, el sistema ha d'entrar en funcionament a l'instant.
- En cas que un node tingui incoherències en els seus ports connectats s'ha d'eliminar automàticament.

La classe principal d'aquest sistema és `PowerNode`. D'aquesta classe hereten tots els elements de la xarxa que afegirem. Cada `PowerNode` tindrà mínim un `PowerNodePort`, aquests ports són cubs ubicats allà on es vol que el node tingui un punt de connexió. Aquest port té quatre propietats molt importants: la direcció de gir respecte al node, el modificador de velocitat, el tipus de port que és, i a quins altres ports es pot connectar.

3.2.1 Construcció de les escenes pels `PowerNode`

En la figura 4, podem veure com estan configurats els ports d'un eix bàsic. Podem veure respectivament les propietats: modificador de velocitat ("Ratio Multiplier"),

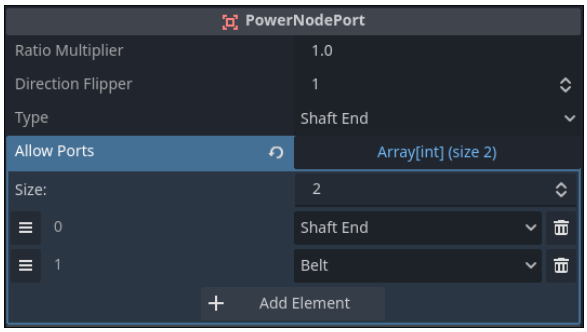

Fig. 4: Exemple de port d'un eix

direcció de gir ("Direction Flipper"), tipus de port ("Type"), i connexions permeses ("Allow Ports"). Els tipus de port implementats són: "SHAFT_END", "COG_SMALL", "COG_BIG" i "BELT". En cas d'implementar un element que ens permeti invertir gir hauríem de tenir dos ports, un com el de la figura 4, i l'altre amb el "Direction Flipper" a -1, si volguéssim implementar un element que ens permetis duplicar la velocitat de gir hauríem de modificar el "Ratio Multiplier" a 2.

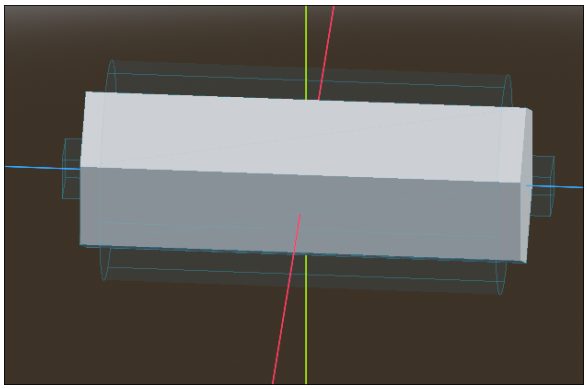

Fig. 5: Exemple de l'escena d'un eix

En la figura 5 veiem l'exemple de l'eix, amb els dos ports col·locats un a cada extrem de l'objecte. Aquests ports hereten de la classe base de Godot "Area3D". Degut a que aquesta classe base té ja té implementats una serie de

mètodes que permeten la detecció d'altres objectes en entrar o sortir dins la seva area, i faciliten cridar altres funcions en el moment que es produeixi un canvi. Aixi sempre en detectar un trespàs cridarem la funció "_on_area_entered" i quan detectem que un objecte surt cridarem la funció "_on_area_exited".

3.2.2 `PowerGridManager` gestor central de transmisió i càlcul de potència

Quan detectem aquesta connexió, comprovem que l'altra àrea sigui un `PowerNodePort` i que la connexió sigui vàlida. En cas de connexió vàlida llancem el senyal "network_changed", que indica al `PowerGridManager` que hi ha hagut un canvi en la xarxa a la qual pertany el node, i per tant s'ha de recalcular la xarxa.

Aquest procés de càlcul comença amb un algoritme "Breadth-first search" ("BFS"), es podria utilitzar també un "Depth-first search" ("DFS") en aquest cas, per tal de trobar tots els nodes connectats. Quan volem trobar tots els nodes connectats no hi ha diferència entre fer servir un "BFS" o un "DFS", però més endavant s'ha d'usar forçosament un "BFS", per tant, ja tenim la funció implementada. Una vegada tenim tots aquests nodes busquem quins són els generadors que hi ha a la xarxa, i és a partir d'aquests generadors que comencem a propagar tots els canvis. Per propagar els canvis apliquem un altre "BFS". En aquest cas ha de ser aquest algoritme, ja que interessa que els canvis es propaguin nivell a nivell, que es calculin abans els nodes més pròxims per així detectar els possibles conflictes en els extrems. Per cada node definim la seva velocitat com la calculada en el moment d'afegir-lo a la llista de "per visitar", i iterem per les seves connexions per tal de calcular la velocitat que haurien de tenir per no generar conflictes. En cas de no tenir la connexió una velocitat ja assignada li assignem la calculada. Si ja tenia una velocitat assignada ens assegurem que no hi hagi conflictes. En cas de conflicte eliminem la peça i tornem a començar el procés de càlcul de la xarxa. Si no hi ha conflicte afegim la connexió al llistat de "per visitar". Finalment, quan hem acabat amb el càlcul de les velocitats recorrem tota la xarxa per comprovar que la potència és prou per mantenir la xarxa activa. Tots els generadors sumen les seves potències i la resta d'elements resten.

3.2.3 Càlcul de velocitats

La part més complexa d'aquest procés és el càlcul de velocitats. Els nodes poden estar situats en qualsevol punt del món i en qualsevol direcció. Per exemple un eix sense girar connectat a un de girat 180 graus presenten una aparienca idèntica, però si es fa una assignació directa de velocitat, el segon eix girarà en sentit contrari. Per solucionar això fem una sèrie de càlculs amb els vectors de direcció de cada element per comprovar com s'ha de comportar aquella connexió. Per fer aquest càlcul seguirem una sèrie de passos, aquests passos no s'executen sempre tots, depenent dels resultats d'operacions anteriors poden canviar els passos que s'executen.

1. $W_{input} = W_{conn} \times R_{conn} \times D_{conn}$
2. $Alg = sgn(\vec{A} \cdot \vec{B})$
3. $Alg = sgn(\vec{S}_{dir} \cdot (1,1,1))$
4. $\vec{V} = (\vec{P}_{conn} + \vec{C}_{conn}) - (\vec{P}_{local} + \vec{C}_{local})$
5. $Alg = \text{sgn}((\vec{A} \times \vec{V}) \cdot (-\vec{B} \times \vec{V}))$
6. $W_{res} = \frac{W_{input} \times D_{local} \times Alg}{R_{local}}$

En la primera fórmula obtenim la velocitat d'entrada aparent, sense tenir en compte l'alineació dels dos nodes. Els paràmetres són els següents: W_{conn} és la velocitat de rotació del node connectat, R_{conn} és el multiplicador de velocitat del port del node connectat i D_{conn} és la direcció del port del node connectat.

En la segona calculem l'alineació de l'eix de rotació propi, $\vec{A}$, i l'eix de rotació del port connectat, $\vec{B}$. L'alineació es calcula utilitzant el producte escalar entre els dos eixos de rotació. A partir d'aquest punt poden passar dues coses: que el valor Alg sigui 0 o 1/-1.

En cas que Alg sigui 1/-1 significarà que els dos ports són paral·lels. Si els dos ports són del tipus "SHAFT_END" la velocitat del port serà la W_{input} multiplicada pel signe de Alg. En cas que els ports siguin del tipus "COG" la velocitat serà W_{input} multiplicada pel negatiu de signe de Alg, per tal d'invertir la direcció de gir.

En el cas que Alg sigui 0 voldrà dir que els dos ports són perpendiculars, aquest es dona quan connectem dos engranatges grans en perpendicular, o quan un eix està connectat a una cinta mecànica. En el segon cas per obtenir la direcció correcta apliquem la tercera fórmula, on S_{dir} és la direcció de gir de l'eix. En el primer cas, dos engranatges grans en perpendicular, apliquem les fórmules 4 i 5. Amb la fórmula 4 obtenim un vector que va des del centre del node connectat al centre del node actual, P_{conn}/P_{local} són les posicions de l'element en l'escena i C_{conn}/C_{local} és un desplaçament per als casos on el centre de gir del node no correspon en la posició en escena. Finalment calculem l'alineació.

Per acabar, aquesta velocitat W_{input}, que representa la velocitat del port la convertim a la velocitat interna que ha de tenir el node, utilitzant la sisena fórmula desfem el càlcul fet en la primera, però fent servir els valors del nostre port i multipliquem per l'alineació final, Alg. Aquest és el valor que retornem perquè el `PowerGridManager` comprovi si hi ha conflicte.

3.3 Logística i Transport d'Ítems

El sistema de logística és l'encarregat de moure els materials entre les diferents màquines de processament. A diferència de jocs on els inventaris són purament numèrics, en aquest projecte s'ha optat per una representació física de cada ítem en el món.

La peça fonamental d'aquest sistema és la cinta transportadora (`Belt`). A nivell d'arquitectura, la classe `Belt` hereta de `PowerNode`. Això implica que les cintes no funcionen de manera autònoma, sinó que requereixen estar connectades a la xarxa d'energia mecànica per moure's. La seva velocitat de transport és directament proporcional a la velocitat de rotació de la xarxa; si la xarxa s'atura o s'inverteix el gir, els ítems sobre la cinta reaccionen en conseqüència.

3.3.1 Simulació de moviment i Path3D

Per gestionar el moviment dels ítems al llarg de la cinta, s'ha utilitzat el node `Path3D` de Godot. Quan es col·loca una cinta entre dos punts (ja siguin eixos o ports de màquines), es genera una corba de Bezier rectilínia que defineix la trajectòria.

Cada ítem es tracta com un objecte `VisualMaterial`, aquesta classe hereta de la classe base `PathFollow3D`. Aquests nodes han de ser fills de un `Path3D`, i tenen una propietat especial que indica quin es el seu progres al llarg del `Path3D`. En cada cicle de processament (`_process`), s'actualitza la propietat `progress` de cada material segons la velocitat actual de la cinta i el temps delta.

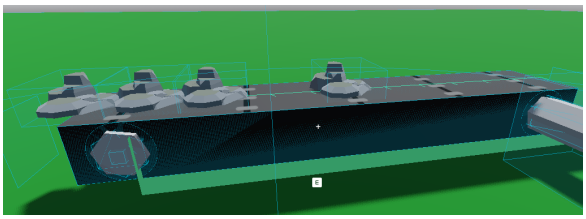

Fig. 6: Escena amb una cinta transportadora amb objectes en moviment i aturats

Tal com es mostra en la figura 6, tenim tres objectes aturats ja que han arribat al final de la cinta, amb un quart objecte que es desplaça cap a l'esquerra. Per aquesta imatge s'ha activat la visualització de caixes de col·lisió, per veure quins són els limits de l'objecte. S'ha implementat un sistema de "contrapressió" (backpressure). Cada ítem disposa d'una `Area3D` que detecta si col·lideix amb un altre ítem situat davant seu. Si un ítem arriba al final de la cinta i no pot saltar al següent node (perquè està ple o no hi ha connexió), s'atura. Aquesta aturada es propaga cap enrere: quan l'ítem posterior detecta la col·lisió amb l'ítem aturat, també deté el seu avanç. Això permet crear cues de materials de manera orgànica sense necessitat d'un gestor centralitzat.

3.3.2 Connexions i transferència

La continuïtat del transport es gestiona mitjançant un sistema de ports similar al de la transmissió mecànica, però especialitzat. La classe `Belt` defineix dos ports de detecció: `FrontPort` i `BackPort`.

Quan dues cintes es col·loquen de manera adjacent, o una cinta es connecta a un `MachinePort` (el punt d'entrada/sortida de les màquines), es crea un enllaç lògic anomenat `BeltConnection`.

La transferència d'ítems es produeix quan un material arriba a l'extrem del `Path3D`:

1. El sistema identifica la connexió activa (`ft_conn` o `bk_conn`) segons la direcció del moviment.
2. S'invoca el mètode `try_add_item` del node receptor (una altra cinta o una màquina).
3. Si el receptor accepta l'ítem (té espai lliure), es duplica l'objecte visual al nou contenidor i s'elimina de la cinta original.

Aquest disseny modular permet que les màquines, definides a la classe `Machine`, s'abstraguin del moviment. Elles simplement exposen uns ports d'entrada i sortida, i és la lògica de la cinta l'encarregada d'injectar o extreure els materials quan sigui físicament possible.

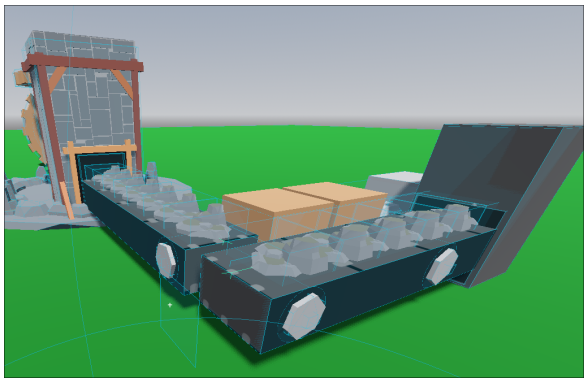

Fig. 7: Mostra de dues cintes connectades entre elles i a dues màquines diferents, un extractor de recursos (esquerra) i una construcció que s'utilitza com a destructor (dreta)

Com es pot observar en la figura 7, aquest protocol garanteix que no es perdin ítems durant la transició i que es respectin les capacitats físiques de cada segment de la xarxa logística. Podem observar una cadena sencera d'esdeveniments: els objectes surten de l'extractor de recursos, quan arriben al final de la primera cinta són transferits a la segona seguint el protocol esmentat anteriorment, i quan arriben al final de la segona entren al destructor per ser eliminats i permetre un flux constant d'objectes.

3.4 Interfícies d'usuari

L'arquitectura de la interfície d'usuari (UI) s'ha dissenyat per ser modular i reactiva, separant clarament la lògica de joc de la presentació visual. Tota la gestió de finestres i menús es centralitza en un únic gestor, el `MenuManager`, que actua com a controlador principal de l'estat de la interfície.

3.4.1 Gestor de Menús i Estats

El script `MenuManager.gd` administra la visibilitat i el focus de les diferents pantalles del joc. Aquest sistema permet alternar entre estats de joc actiu (on el ratolí està capturat per la càmera) i estats de menú (on el ratolí interactua amb la UI). L'estructura jeràrquica de la UI s'organitza sota un node `CanvasLayer`, garantint que els elements gràfics es renderitzin sempre per sobre de l'escena 3D.

El sistema distingeix entre tres tipus principals d'interfície:

- **HUD (Heads-Up Display):** Elements persistents com la barra d'accés ràpid (`BottomMenu`), que permet al jugador seleccionar les estructures a construir.
- **Menús de Sistema:** Finestres modals com el menú de construcció (`BuildingMenu`), que mostra el catàleg complet d'objectes disponibles.
- **Interfícies Contextuals:** Menús específics que depenen de l'objecte amb el qual s'interactua, com ara l'inventari d'un forn o la configuració d'una màquina.

3.4.2 Components Contextuals

Per vincular la lògica dels objectes 3D amb la interfície 2D sense crear dependències rígides, s'ha implementat el node `ContextComponent`. Aquest component s'afegeix com a node fill a qualsevol entitat del món (màquines, cofres, etc.) que requereixi una interfície pròpia.

Quan el jugador interactua amb un objecte, el sistema comprova si aquest disposa d'un `ContextComponent`. En cas afirmatiu, el component instancia l'escena de la interfície corresponent (per exemple, `furnace_gui.tscn`) i la injecta al `MenuManager`. Això permet que cada objecte gestioni la seva pròpia lògica visual de forma encapsulada.

3.5 Construcció i interacció amb objectes

El nucli del joc resideix en la capacitat del jugador per modificar l'entorn, col·locar estructures i connectar-les. Aquest sistema es divideix en la lògica de detecció (interacció) i la lògica de col·locació (construcció).

3.5.1 Sistema d'Interacció

La interacció es basa en el llançament de raigs (*Raycasting*) des de la càmera del jugador. Com s'ha detallat en la secció de senyals, s'utilitza un `InteractionComponent` per estandarditzar el comportament de tots els objectes interactuables. Aquest disseny permet diferenciar clarament entre dos tipus d'accions:

1. **Interacció directa:** Accions com obrir un menú o activar un interruptor, gestionades pel senyal `interacted`.
2. **Connexió de ports:** En el cas de les cintes transportadores i eixos, la interacció detecta automàticament els `PowerNodePort` o `MachinePort` propers per facilitar la connexió entre màquines.

3.5.2 Lògica de Construcció

El sistema de construcció funciona mitjançant una màquina d'estats implícita gestionada per l'objecte que s'està intentant col·locar. El procés segueix el següent flux:

1. Selecció i Instanciació "Fantasma": Quan el jugador selecciona un objecte del menú, es crea una instància temporal de l'objecte (coneguda com a *ghost* o fantasma). Aquesta instància no té col·lisions físiques actives però segueix la posició del cursor del jugador, projectada sobre el món 3D.

2. Validació de la posició: Abans de confirmar la construcció, l'objecte executa una funció de validació (visible en scripts com `belt.gd`). Aquesta funció comprova:

- Que no hi hagi col·lisions amb altres objectes existents.
- Que l'objecte estigui recolzat sobre una superfície vàlida (el terra o una altra màquina).
- En el cas d'elements direccionals com les cintes, es calcula la longitud i la rotació necessària entre el punt d'inici i el punt final.

Visualment, aquesta validació es comunica al jugador mitjançant *shaders* que tenyeixen l'objecte de verd (posició vàlida) o vermell (posició invàlida).

3. Col·locació i Actualització: En confirmar l'acció (clic esquerre), l'objecte passa a ser part permanent de l'escena. En aquest moment:

1. S'activen les seves col·lisions (`CollisionShape3D`).
2. Es registra l'objecte al node pare `Buildings` per garantir-ne la persistència en el guardat.
3. En el cas de nodes de potència (`PowerNode`), es detecten els veïns i es dispara el recàlcul de la xarxa de transmissió mecànica per integrar la nova peça al sistema de grafs.

3.5.3 Construcció Dinàmica: El cas de les Cintes

Un cas especial d'interès és la construcció de cintes transportadores (`Belt`). A diferència dels edificis estàtics, les cintes es construeixen definint dos punts. El script `belt.gd` gestiona aquest comportament calculant dinàmicament la distància entre els nodes connectats i ajustant l'escala del model 3D i la corba del `Path3D` per adaptar-se a l'espai disponible. Això permet una construcció fluida i flexible, adaptant-se automàticament a la distància entre màquines sense necessitat de col·locar trams individuals peça a peça.

3.6 Guardar i carregar escenaris

Guardar i carregar l'estat del joc és una funcionalitat molt important, rarament es completa un videojoc en una sola sessió, o el jugador no té interes en guardar el seu progrés, fets que fan indispensables un sistema per tal de recuperar l'estat previ. A més per tal de crear nivells de prova i fer la demo jugable necessitava un sistema que permetis guardar i carregar escenaris.

3.6.1 Guardar

L'incorporació d'aquest sistema ha requerit la modificació de l'estructura de persistència de dades dels objectes, canvis al codi executat en el moment d'entrar en escena i en cas de tenir variables que depenen d'altres nodes modificar-les, ja que un node no pot fer referència a un altre en el moment de guardar-lo, això genera un error de recursivitat infinita degut a dependències cícliques.

Tot i tots aquests canvis que s'han hagut de fer el motor Godot proporciona eines molt senzilles i directes per tal de guardar escenes en disc. La primera modificació ha consistit en garantir que tots els objectes que tenia intenció de guardar estiguessin en una ubicació coneguda, en el meu cas he creat un node anomenat `Building`, aquest node serveix per a dos propòsits, fa de pare per a tots els nodes del tipus `Building` i manté una llista actualitzada de tots els nodes `PowerNode`.

En prémer el botó de guardar executarem una funció recursiva per assegurar-se que tots els nodes (fills, nets, etc.) tinguin la propietat "owner", definida al node `Building`. Aquest pas és imperatiu perquè la funció integrada de Godot "ResourceSaver.save(node, path)", només guardarà

aquells nodes que el seu ”owner” sigui el node passat com a paràmetre.

Aquesta funció per guardar ens permetrà guardar en dos tipus de fitxers diferents, fitxers ”tscn” i ”scn”. El primer és un fitxer de text, similar a ”json”, que permet una bona llegibilitat i fàcil edició del contingut, és el mateix format que utilitzen les escenes de Godot per defecte, no és un format massa eficient per a l'ordinador però senzill de depurar per al desenvolupador. El format ”scn” és un format binari, aconsegueix temps de guardat i càrrega superiors i una menor mida de fitxer de guardat, per contra, com és un binari és molt complicat de depurar i modificar manualment sense eines externes que l'interpretin.

3.6.2 Càrrega

Per carregar el fitxer de guardat Godot ens ho fa senzill també, necessitem carregar aquest fitxer com una variable del tipus ”PackedScene”, aquesta variable guarda una versió comprimida d'una escena, una vegada carregat creem una nova instància de l'escena i obtindrem un node `Building` amb tots els elements que contingués en el moment de guardat. Finalment, substituïm el node anterior pel qual acabem de carregar.

3.6.3 Canvis i problemes a l'hora d'implementar el sistema

Explicat el procés de guardat i carrega faré menció a totes aquelles parts que s'han hagut de modificar per adaptar-les a aquest sistema.

El primer és marcar amb la propietat ”@export”/”@export_storage” totes les variables que es volen guardar, tota variable que no tingui aquesta propietat serà ignorada per la funció de guardat.

```
@export_group("Meshes, areas and collistions of the building")
@export var meshes_path : Array[NodePath]
@export var areas_path : Array[NodePath]
@export var collisions_path : Array[NodePath]

var meshes: Array[MeshInstance3D]
var areas : Array[Area3D]
var collisions: Array [StaticBody3D]
```

Fig. 8: Codi actual per guardar les referències a nodes

A continuació s'han de modificar totes les variables que guardaven referències directes a nodes, i en el seu lloc guardar un ”NodePath” en l'exemple de la Fig. 8 podem veure com mantenim els ”@exports” per a modificar aquestes variables des de l'editor i que aquests ”paths” es guardin, però en el moment de carregar l'escena hem de convertir aquests ”paths” en les referències directes que guardaran les variables ”meshes”, ”areas” i ”collisions”.

El codi per executar aquesta conversió s'executarà quan es creï l'objecte i en tot moment que aquest objecte es carregui des de la memòria.

Un altre component important que necessita ser executat en tots aquests moments és el codi encarregat de connectar els diferents senyals utilitzats. En el moment de guardar i carregar aquests senyals es desconnecten i han de ser reconnectades.

Finalment, tots els `PowerNode` hauran de tornar-se a connectar amb els nodes adjacents i es farà un recàlcul sencer de la xarxa de potència, per tal d'evitar possibles errors que es podrien produir si no comprovem que les velocitats de rotació dels nodes no són les correctes. Durant el procés de càrrega cal considerar que la detecció de col·lisions de les ”Area3D” no depenen del fotograma lògic, sinó del cicle de física, per tant, per evitar possibles condicions de carrera, i comportaments no deterministes necessitem implementar una espera explícita mitjançant l'instrucció ”await get_tree().physics_frame”. Això endarrerirà l'execució de les físiques un fotograma sencer, i ens assegurem que el fotograma lógic s'hagi completat.

4 Conclusions

Agraïments

References

[1] Godot

Apèndix