EPS Escola Politècnica Superior
UdG

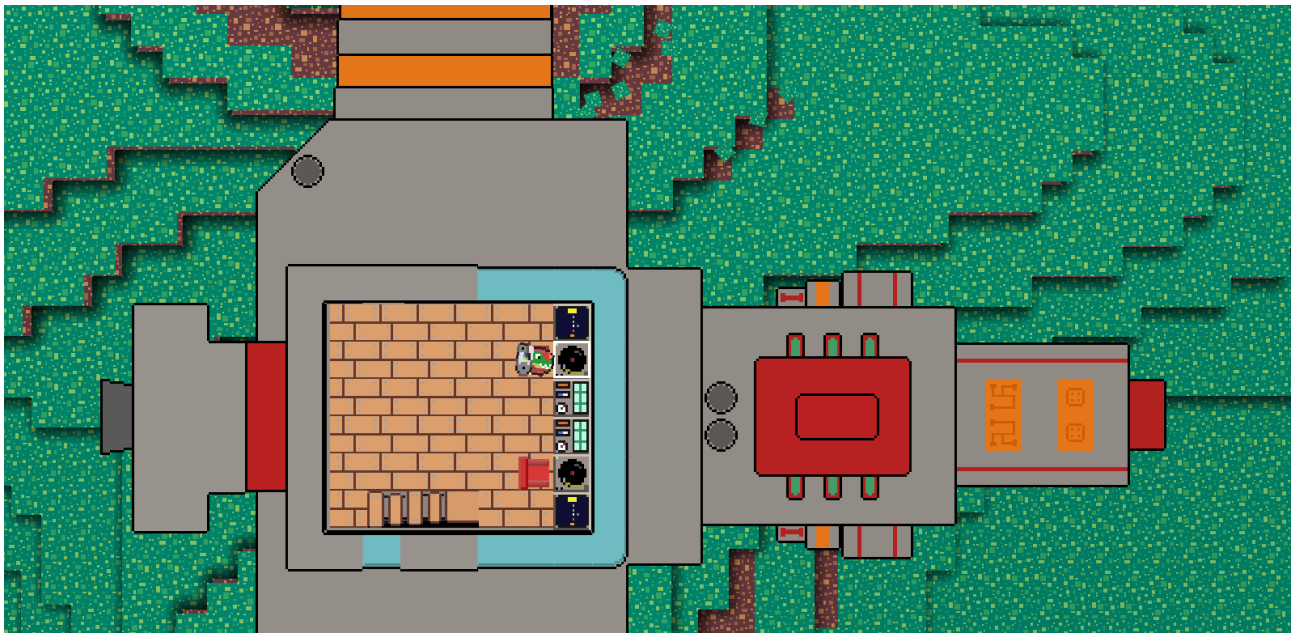
Escola Politècnica Superior

Universitat de Girona

Desenvolupament d'un motor per jocs de rol 2.5D per Nintendo Switch i PC

Projecte/Treball Fi de Carrera

Grau en Enginyeria Informàtica. Pla 2015



Document: Memòria

Autor: Lluís Trilla i Esquinas

Director: Dr. Gustavo Patow

Departament: Informàtica, Matemàtica Aplicada i Estadística

Àrea: LSI

Convocatoria: 9/2020

Índex

Capítol 1

Introducció

Gràcies als grans avenços en computació, a principis dels anys 70 els primers ordinadors personals van començar a aparèixer a les llars. Encara sense Internet i amb unes especificacions molt simples comparades amb la maquinària d'avui en dia, el software capaç de córrer en aquelles màquines era bastant limitat. No obstant, aquestes limitacions no van impedir l'aparició dels primers videojocs i videoconsoles.

Tot i tractar-se de jocs inicialment molt senzills, juntament amb l'evolució dels ordinadors on s'executaven van augmentar de complexitat de forma accelerada. Inicialment eren creats per un sol desenvolupador, programats en llenguatge ensamblador, però l'augment de complexitat va requerir llenguatges de més alt nivell, amb majors i millors abstraccions i equips de desenvolupadors més grans. Gran part de la feina al fer un joc era repetida respecte jocs anteriors, i per tant molts jocs van passar a reaprofitar codi de jocs anteriors. Això va portar a crear una distinció més clara entre joc i motor, per tal de poder reaprofitar tot allò que no calia fer de nou.

En el disseny i arquitectura de videojocs s'anomena motor al conjunt d'eines que faciliten la creació de videojocs. Aquestes eines poden ser tant executables amb interfície gràfica que permetin a un artista crear un nivell, com un conjunt de APIs que treguin càrrega de feina als desenvolupadors i ofereixin un nivell més alt d'abstracció.

1.1 Què és el *homebrew*?

Des dels inicis de les videoconsoles molts grups de desenvolupadors han sentit curiositat per executar codi lliurement en aquests dispositius. Encara que inicialment les consoles no tinguessin cap tipus de protecció, per tal d'evitar la pirateria els fabricants han anat tancant i protegint el hardware per evitar l'execució de codi arbitrari. Des de tècniques bàsiques com signar el codi a executar, fins a protecció contra atacs mitjançant ASLR o pàgines de memòria no executables. Tot i això, sempre que els fabricants han intentat protegir el hardware, els diferents grups de 'hackers' han treballat per vulnerar aquesta seguretat i executar codi no signat. Aquestes aplicacions o codi no signat pel fabricant és el que s'anomena *homebrew*.

1.2 Motivació

Els principals motors per videojocs del mercat actualment són de codi propietari. Tot i que això pot semblar un petit detall, molts principiants es poden veure descoratjats davant els possibles costos i dificultats associats. Si bé és cert que gairebé tots aquests motors tenen opcions gratuïtes per a principiants, la possibilitat de poder modificar el codi lliurement i sense

por a cap tipus de litigi pot fer que les opcions open-source siguin molt més atractives. Godot és un dels motors de codi obert amb més potencial actualment, en constant evolució i cada cop més útil. Tot i així, per tal de poder executar un joc fet en Godot en una videoconsola es requereix accés als Devkits oficials, suposant un gran cost per a desenvolupadors petits. És per aquests motius que he decidit crear un motor de codi lliure amb suport per a homebrew a Nintendo Switch, a part de suport per Linux i Windows.

El món dels videojocs sempre ha estat un tema que m'ha atret, i des de fa uns anys, el desenvolupament d'aquests m'ha semblat fascinant. Crec que és una opció perfecta per posar en pràctica els coneixements adquirits al llarg del grau, aplicant des de conceptes apresos a Enginyeria del Software fins a parts més creatives relacionades amb multimèdia.

1.3 Propòsit i objectius del projecte

L'objectiu d'aquest projecte és crear un motor multiplataforma pensat per a jocs de rol en un món basat en voxels. El motor gestionarà el renderitzat a partir d'una vista d'ocell, dibuixant sprites 2D i aplicant efectes de parallatge segons la profunditat per aconseguir un efecte 3D utilitzant sprites.
Dintre del joc hi haurà suport per a distàncies a escales galàctiques, permetent una simulació a escala real d'òrbites de tot el sistema solar, així com l'òrbita d'aquest respecte al centre de la via làctea, Sagitari A*.
Tanmateix, alhora que permetrà aquesta simulació a gran escala, també tindrà suport per a moviments mil · limètrics quan faci falta, permetent al jugador, enemics i projectils moure's amb gran precisió.
Els diversos planetes i objectes astronòmics tindran un terreny modificable basat en vòxels, el qual serà manipulable en temps real pel jugador o altres entitats dins del joc.

1.4 Estructura de la memòria

Aquest document s'ha organitzat en 14 capítols, que són els següents:

1. **Introducció, què és el *homebrew*?, motivació, propòsit y objectius del projecte.** En aquest capítol s'explica el perquè del desenvolupament d'aquest projecte, què és el *homebrew*, quins són els objectius proposats i com s'ha organitzant el desenvolupament per portar-lo a terme.

2. **Estudi de viabilitat.** En aquest capítol s'exposen els paràmetres que fan possible el desenvolupament del projecte.

3. **Metodologia.** Aquest capítol conté una explicació de la tecnologia emprada i el perquè de l'elecció.

4. **Planificació.** En aquesta etapa es defineix l'estratègia emprada per arribar a complir els objectius plantejats.

5. **Marc de treball y conceptes previs.** En aquest capítol es descriuen els aspectes relacionats amb el desenvolupament general del projecte, que ajudaran a entendre millor els següents capítols. També es tractaran les principals accions desenvolupades durant les primeres etapes de la realització del motor. S'inclouran els passos d'estudi i aprenentatge de conceptes que s'hagin utilitzat per al desenvolupament.

6. **Requisits del sistema.** En aquest capítol es defineixen els requeriments del programari, els quals recullen, a grans trets, els objectius de l'aplicació juntament amb les funcionalitats que es volen obtenir. Aquest document permet entendre els elements que envolten el sistema informàtic que s'intenta construir.

7. **Estudis i decisions.** Aquesta secció conté una descripció de les eines utilitzades, amb les seves característiques i l'ús que se'ls hi ha donat, tant de llibreries i motors com de programari.

8. **Anàlisis i disseny del sistema.** Aquest apartat proporciona una comprensió precisa de les necessitats del sistema. Es a dir, s'encarrega de la investigació del problema a resoldre, però no tracta el trobar una solució. Usant Enginyeria del Software, en aquesta secció es tradueixen els requeriments esmentats en capítols anteriors a un llenguatge més formal. La part de disseny permet augmentar el nivell de especificació, i realitzar un esquema de implementació del sistema mitjançant vàries eines de programació orientades a objectes.

9. **Implementació i proves.** En aquest capítol es donen a conèixer com s'ha construït l'aplicació, les classes i els mètodes implementats que resulten més significatius per la comprensió del funcionament del videojoc.

10. **Resultats.** En aquest capítol es mostren proves d'execució de l'aplicació, es mostren imatges del motor i del videojoc demo, incloent interfícies, imatges de la partida, i tot el que es pot visualitzar del conjunt que ha estat implementat.

11. **Conclusions.** En aquest apartat s'exposen les conclusions extretes una vegada finalitzat el projecte.

12. **Treball futur.** En aquesta secció s'exposa tot allò que es pot millorar en el motor, o ampliar de forma interessant.

13. **Bibliografia.** Aquest capítol conté les referències usades pel desenvolupament del projecte.

14. **Manual d'usuari i instal · lació.** Aquesta secció inclou especificacions del funcionament del producte.

Capítol 2

Estudi de viabilitat

Per tal de desenvolupar el projecte tindrem petits costos d'infraestructura, i no en tindrem cap d'estructura. En aquest cas ja disposàvem del hardware necessari per poder desenvolupar el projecte de forma efectiva.

2.1 Recursos de *hardware*

Per desenvolupar el projecte farem servir un ordinador de sobretaula amb les següents especificacions tècniques:

- **Sistema Operatiu:** Arch Linux
- **Processador:** Intel i5 6600
- **Memòria RAM:** 16 GiB DDR4 a 2666 Mhz
- **Targeta gràfica:** Nvidia GTX 980
- **Emmagatzematge:** SSD NVMe Samsung 960 EVO 250 GB

Tot i que seria possible desenvolupar el projecte amb unes especificacions inferiors, i una targeta gràfica integrada seria suficient tant pel desenvolupament com per fer servir el motor, el processador pot causar llargues durades de compilació, sobretot al fer *builds* optimitzades.
A més del equip per desenvolupar el motor, necessitarem una Nintendo Switch capaç d'executar *homebrew* per poder fer proves i comprovar que el projecte funciona de forma correcte en la versió per la consola.

2.1.1 Cost de *hardware* i materials

Component	Unitats	Preu unitari	Preu total
Ordinador	1	1300€	1300€
Pantalla 24"	2	120€	240€
Pantalla 17"	1	40€	40€
Teclat i ratolí	1	400€	400€
Immobiliari	1	150€	150€
Nintendo Switch	1	320€	320€
Total			2450€

Com ja hem esmentat, ja disposàvem d'aquest *hardware* a l'inici del projecte. En cas de no disposar d'aquest, el pressupost s'ajustaria per invertir menys en teclat i ratolí, i els components del ordinador estarien planejats de forma lleugerament diferent.

2.2 Recursos humans

Per calcular el cost de recursos humans hem creat diversos perfils de treballador amb un salari associat, i hem repartit les tasques i les hores al perfil adequat en cada cas. Els perfils són els següents:

Analista/Dissenyador: 25€/hora.
Programador: 20€/hora.
Dissenyador gràfic: 25€/hora.

Tasca	Perfil	Hores	Cost
Investigació	Analista/Dissenyador	80	2000€
Disseny dels algorismes	Analista/Dissenyador	80	2000€
Implementació dels algorismes	Programador	240	4800€
Proves i optimitzacions	Programador	80	1600€
Disseny conceptual gràfic	Dissenyador gràfic	80	2000€
Creació de gràfics	Dissenyador gràfic	120	3000€
Disseny d'interfície d'usuari	Dissenyador gràfic	40	1000€
Memòria	Analista/Dissenyador	80	2000€
Total		800	18400€

2.3 Recursos de *software*

Tot el *software* que farem servir, incloent el sistema operatiu, serà tant de codi lliure com gratuït. Per tant, el cost en recursos de *software* serà nul. Tot i que farem servir el programa Aseprite comprat a la plataforma Steam, el codi està lliurement disponible a Github.com i existeixen paquets precompilats per al nostre sistema operatiu als repositoris oficials.

Capítol 3

Metodologia

Per a la realització del projecte no s'ha seguit una metodologia de treball estàndard, sinó que s'ha triat i usat una metodologia personalitzada plantejada amb el tutor. Els passos d'aquesta metodologia són els següents:

1. Triar el treball a realitzar.
2. Decidir el llenguatge de programació i eines a utilitzar.
3. Aprendre el llenguatge de programació i les eines escollides.
4. Estructurar el treball en parts segons les funcions que s'han de realitzar.
5. Desenvolupar la part corresponent seguint l'ordre de l'estructura del treball.
6. Fer les comprovacions per confirmar que el funcionament és correcte al acabar cada part.
   (a) Si al fer les comprovacions el resultat no és l'esperat, es tornarà al punt 5 per a realitzar els canvis oportuns a l'última part desenvolupada o en les anteriors, si és convenient.
   (b) Si al fer les comprovacions el resultat és el desitjat, es desenvoluparà la següent part tornant al punt 5. Una vegada s'hagin finalitzat totes les parts amb les respectives comprovacions, s'iniciarà el punt 7.
7. Unir totes les parts desenvolupades i comprovar que el funcionament és correcte.
   (a) Si al fer les comprovacions el resultat no és el desitjat, es tornarà al punt 5 per a realitzar els canvis oportuns a l'última part desenvolupada o en les anteriors, si així és necessari.
   (b) Si al fer les comprovacions el resultat és l'esperat, s'iniciarà el punt 8.
8. Generar diferents models d'exemple per a comprovar que el funcionament és el correcte.
   (a) Si al fer les comprovacions el resultat no és el desitjat, es tornarà al punt 5 per a realitzar els canvis oportuns a l'última part desenvolupada o en les anteriors si així és convenient.
   (b) Si al realitzar les comprovacions el resultat és l'esperat, s'iniciarà el punt 9.
9. Arrodonir la documentació desenvolupada al llarg del projecte.

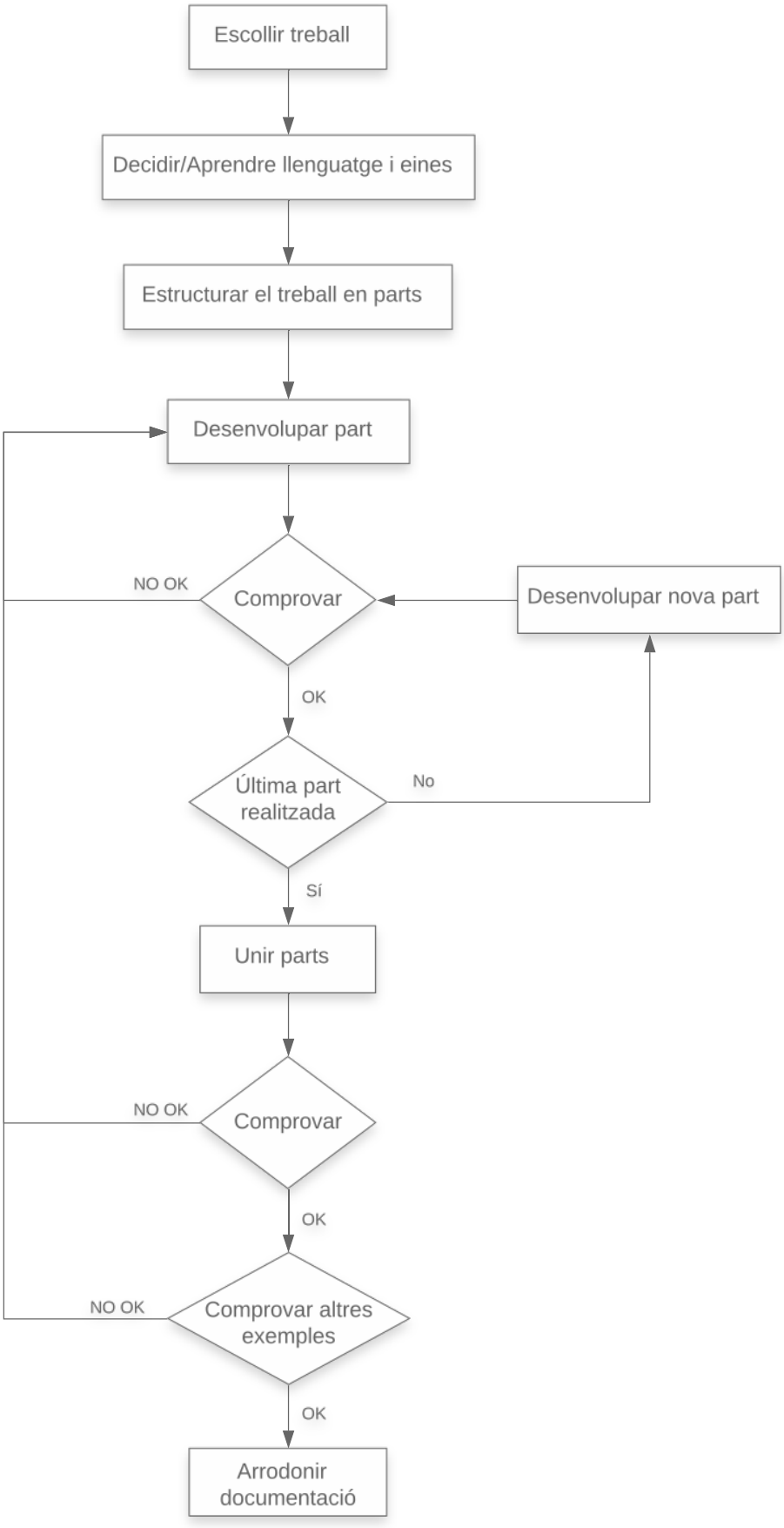


Figura 3.1: Diagrama d'activitat de la metodologia

Capítol 4

Planificació

És important tenir una planificació de tasques en els projectes grans, per tal de poder decidir aproximadament el temps a dedicar a cada secció del projecte. Encara que no es segueixi al peu de la lletra, serveix com a referència per a la direcció del projecte.

L'objectiu inicial del projecte era acabar-lo i presentar-lo en la convocatòria de Septembre del 2019, però a causa de l'incompliment dels requisits pel lliurament i presentació del treball, l'objectiu va passar a ser el Septembre del 2020. És per això que presentem dues planificacions temporals, una feta amb l'objectiu de presentar el Septembre del 2019, i una altre feta al final del projecte reflectint la temporalització real del projecte.

4.1 Planificació original

En la planificació original hi havia les següents tasques:

- Planificació inicial del projecte: Fase inicial del projecte per fer un estudi general dels objectius, i decidir la direcció en que dur a terme el projecte.
- Estudi de dependències i eines: Fase on decidir les eines i dependències a utilitzar en el projecte, i estudiar i aprendre a utilitzar-les.
- Creació d'una capa d'abstracció de hardware: Fase d'implementació d'una capa d'abstracció de hardware sota la que desenvolupar el motor. Aquesta capa permetrà generalitzar les funcions en que la implementació pot ser dependent de la plataforma.
- Implementació d'una petita demo multiplataforma: En aquesta fase farem una petita demo amb sò, gràfics i interaccions amb els controls per tal de comprovar el correcte funcionament multiplataforma de la capa d'abstracció que hem implementat.
- Implementació del model de dades i la base de la simulació: Fase on implementarem la base del joc. Això inclou un sistema d'estats del joc (menú principal, opcions, joc ...), la simulació del univers i els cossos astronòmics d'aquest, i el sistema de gestió i actualització d'entitats (jugador, enemics ...).
- Implementació del render: Fase on s'implementarà el procés de renderitzat, per tal de poder visualitzar l'estat de la simulació.
- Implementació de la simulació física: Fase on s'implementaran les col·lisions entre les entitats, i entre les entitats i els diferents cossos astronòmics.
- Documentació: Fase on enllestir la documentació del projecte.

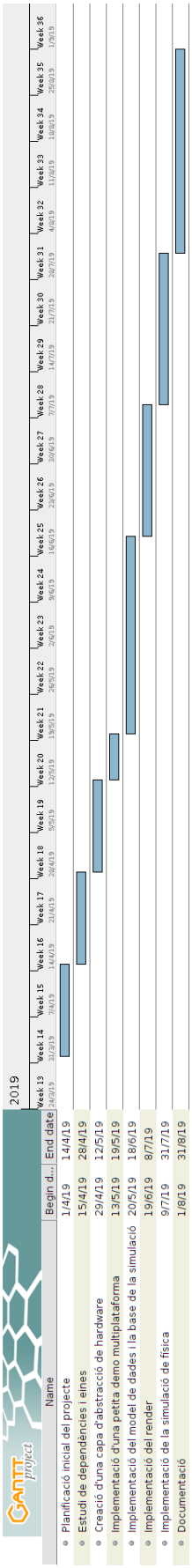


Figura 4.1: Diagrama de la planificació original

4.2 Planificació final

En la planificació final actual es van afegir diverses tasques que es descriuen a continuació. Cal esmentar que s'ha afegit detall a posteriori a la planificació per oferir una millor explicació.

- Refinament i neteja de bugs: Fase on netejar bugs i problemes pendents.
- Refactorització i neteja de codi: Fase per fer canvis no-funcionals al codi per tal de posar-nos al dia amb el deute tècnic. Això inclou simplificacions de codi, millor repartiment del codi en funcions i classes, i possiblement refer des de 0 algunes funcions o classes si és necessari.
- Millores en l'implementació de la física: Fase on acabar de refinar els càlculs de física. Implementació de col·lisions entre nodes i optimització dels càlculs entre entitats.
- Optimitzacions de renderitzat: Fase d'optimització i neteja del procés de renderitzat per tal d'obtenir un millor rendiment.
- Creació d'uin framework per a interfícies d'usuari: Fase per crear el framework que facilitarà fer la interfície d'usuari del joc.
- Disseny i implementació d'una interfície senzilla per la demo: Fase on utilitzarem el framework que hem creat per fer una interfície d'usuari per la demo del joc.
- Implementació d'un sistema per a nodes prefabricats + editor gràfic: Fase on implementarem un sistema per crear i posar en el joc nodes prefabricats. Un node prefabricat pot ser des d'un vehicle terrestre format per un parell de blocs, a una estació espacial de milers de blocs.
- Implementació d'una màquina d'estats pel jugador: Fase on implementarem una senzilla màquina d'estas per controlar al jugador (o altres entitats). Això permetrà que el jugador tingui disponibles diferents accions, com per exemple, a l'estar al aire respecte quan està amb els peus a terra.
- Implementació del sistema de propulsió de nodes: Fase on implementarem un sistema que simularà diversos motors de propulsió juntament amb el combustible. Això inclou la gestió de diversos combustibles, el consum i aprovisionament de combustible, i la generació de forces a partir dels motors.
- Disseny i implementació d'un sistema de blocs interactuables: Fase on crearem un sistema per permetre la interacció entre entitats i blocs, de forma que una entitat pugui seure en una cadira, o pugui fer accions com obrir portes o controlar màquines.
- Creació de contingut per la demo: Fase on crearem contingut per la demo, com per exemple generadors de terreny per als planetes, o naus prefabricades que el jugador pugui pilotar.
- Disseny i creació d'un sistema Observer per a esdeveniments del joc: Fase on implementarem un Observer per poder desacoblar parts del joc que no ho necessitin. Per exemple, enviar esdeveniments a un observador permetria activar una fita quan el jugador fa un salt de més de 100 metres, sense que el sistema de fites i el sistema de física estiguin altament acoblats.
- Implementació de projectils i sistema de salut: Fase on implementarem un senzill sistema per tal que les entitats puguin disparar projectils, i perdin salut o siguin eliminats al rebre dany d'aquests.

- Refinament final: Fase per a fer els últims retocs i netejar bugs pendents.

- Documentació: Fase per finalitzar la documentació del projecte.

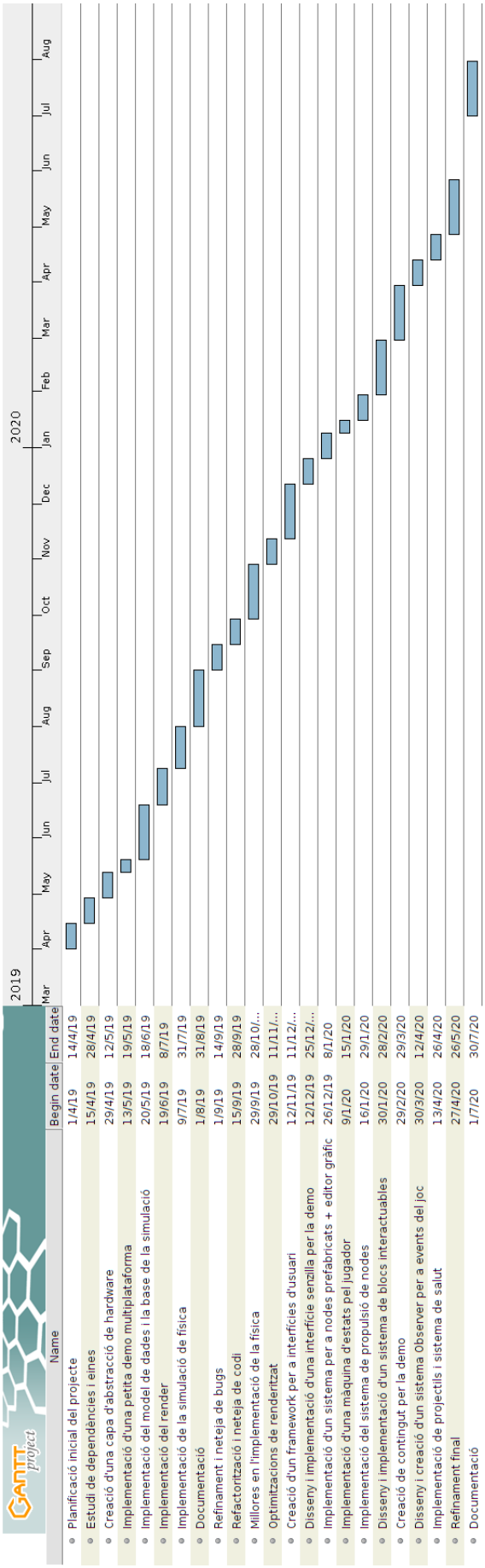


Figura 4.2: Diagrama de la planificació final

Capítol 5

Marc del treball

5.1 Conceptes

5.1.1 Motor de videojocs

Un motor de videojocs és un conjunt d'eines i entorns de desenvolupament que faciliten la creació d'un videojoc. La definició és bastant oberta, ja que podem considerar com a motor des de un conjunt de llibreries independents, com és el cas d'Amethyst, fins a motors amb entorns gràfics d'alta complexitat, com Unreal o Unity.

5.1.2 Vòxels

Podem definir un vòxel com una casella dins d'una malla tridimensional. Seria l'extensió del concepte de píxel si afegissim una nova dimensió a la malla. Podem definir un vòxel com la extensió a l'espai 3D d'un píxel. Dit d'altra manera, un vòxel és una sola casella dins d'una malla tridimensional. Si bé ja es feien servir freqüentment en anàlisi de dades mèdiques i científiques, en l'última dècada s'ha popularitzat l'ús en videojocs, principalment gràcies a Minecraft.
En comparació a mètodes més tradicionals per representar terreny, els vòxels faciliten la subdivisió de l'espai en trossos més petits, així com la manipulació i edició del terreny en temps real. A més, la simplicitat de la representació facilita molt el procés de carregar i desar conjunts de vòxels al emmagatzematge.
En la figura 5.1.2 podem observar les similaritats entre vòxels i píxels. En la figura a) observem un cercle representat per píxels, i en la figura b) observem la representació d'una esfera formada per vòxels.

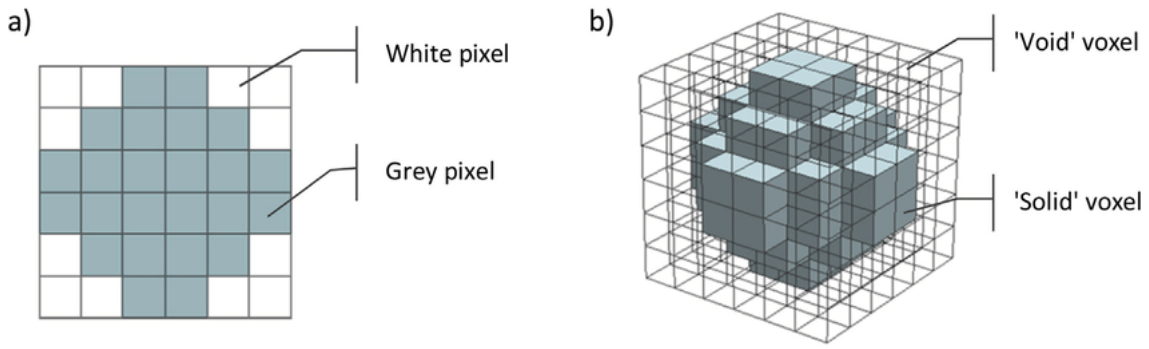


Figura 5.1: Representació gràfica de píxels i vòxels.

5.1.3 Renderització

S'anomena renderització al procés de convertir dades a imatges a través d'un programa informàtic. És un terme molt general, que pot descriure el procés de crear un video a través de software d'edició de video, o la representació en temps real d'un videojoc gràfic.

5.1.4 Astrodinàmica

L'astrodinàmica és el camp de la física que estudia el moviment dels cossos celestes no propulsats. Ja que el nostre objectiu és fer un videojoc en temps real, el que farem per simular les òrbites entre diversos cossos celestials serà integrar les trajectòries a partir de les següents fòrmules:

- L'equació de gravitació universal de Newton, de forma $F = G\frac{m_1 m_2}{r^2}$ on F és la força resultant, G és la constant de gravitació universal, m_1 i m_2 són les masses dels dos cossos, i r és la distància entre ells
- La segona llei de Newton, $F = ma$, on F és la força, m és la massa del cos, i a és l'acceleració del cos.

Per la integració farem servir el mètode d'Euler, que és un mètode d'integració numèrica pertanyent a la familia de mètodes de Runge-Kutta. El mètode consisteix en aproximar punts de la corva a partir de l'equació diferencial i un valor inicial. Tindrem un tamany de passa fix, a partir del qual avançarem en l'equació assumint que la recta tangent a la corva en el punt actual aproxima prou correctament la corva real. Havent obtingut la recta tangent a partir del pendent al punt conegut, avançarem en aquesta recta segons el tamany de passa, i obtindrem el nou punt de l'aproximació. L'error d'aquest mètode és proporcional al quadrat del tamany de passa, així que haurem de triar el tamany de passa amb cura per mantenir l'error sota control.

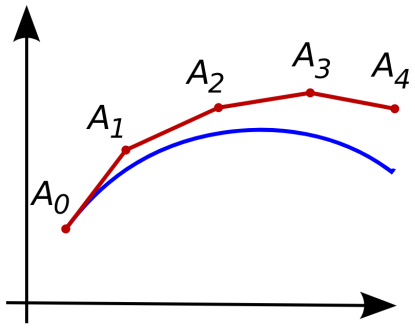


Figura 5.2: Representació del mètode d'Euler. En blau tenim la corva real, i en vermell tenim l'aproximació a partir dels punts obtinguts.

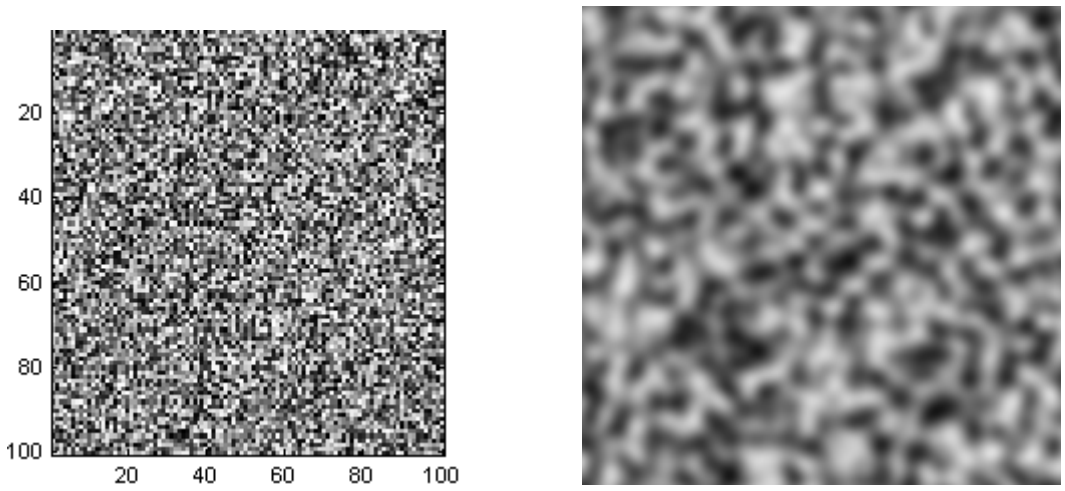
5.1.5 Entity Component System

Entity component system és un patró de disseny de software utilitzat en el desenvolupament de videojocs. El patró especifica un sistema on tots els objectes del joc son una 'entitat'. Fent ús de composició, les entitats no són més que agrupacions de diversos components, i cada component és actualitzat de forma independent pel sistema.
Per exemple, podem imaginar el component Cos. Tota entitat que tingui un cos físic i hagi de poder col·lisionar amb altres entitats i nodes tindrà un component Cos, el qual serà una estructura que emmagatzemarà les propietats del cos. Per altra banda tindrem el sistema corresponent als cossos, el qual a cada actualització s'encarregarà de detectar i solucionar les

possibles col·lisions entre cossos.
Aquest patró facilita la modularitat de les entitats, i ajuda en gran mesura a crear i manipular entitats de forma dinàmica, on un objecte qualsevol podria obtenir la habilitat de, per exemple, volar, tant sols afegint-li el component corresponent.

5.1.6 Gradient de soroll

Un gradient de soroll és una funció matemàtica que genera valors numèrics pseudo-aleatòris dins d'un cert rang a partir d'unes coordenades d'entrada. Aquest soroll no és directament pseudo-aleatòri com podria ser-ho el soroll blanc, sinó que és generat mitjançant interpolació entre valors pseudo-aleatòris. Aquesta interpolació fa que els valors del soroll en una certa àrea estiguin interrelacionats, i resulta en un soroll suau i poc abrupte. En la figura 5.3 podem veure la diferència entre aquests dos tipus de soroll.



(a) Exemple de soroll blanc (b) Exemple de gradient de soroll

Figura 5.3: Comparació de sorolls

5.2 Termes

A part dels conceptes explicats en l'apartat anterior, en aquesta memòria farem servir alguns termes que s'expliquen a continuació.

- Node: Anomenarem nodes als cossos astronòmics de la simulació. Això engloba des del forat negre del centre de la via làctea amb un radi de $22 \cdot 10^9 m$, fins a una petita nau espacial que el jugador hagi construït dins del joc, amb unes dimensions de tant sols 10m^3.
- Entitat: Anomenarem entitats a tots els personatges i objectes que interactuen amb el món. Exemples d'entitats podrien ser el propi jugador, un enemic, una pilota, o la càmera del joc.

Capítol 6

Requisits del sistema

6.1 Requisits funcionals

Els requisits funcionals pel motor i la demo que hem establert com a objectiu d'aquest projecte són els següents:

- Realitzar una simulació en temps real de les òrbites dels planetes i satèl·lits i altres cossos astronòmics del sistema solar.
- Simular col·lisions entre entitats i nodes.
- Oferir una experiència sense pantalles de càrrega dins el joc, tal que el jugador pugui atravessar la galàxia sense aturar el *gameplay* en cap moment.
- Permetre al jugador interactuar amb els nodes de forma directa, ja sigui destruïnt i construïnt terreny, o pilotant-los directament.
- El motor ha de gestionar de manera transparent els canvis d'entitats entre nodes. Dit d'altra manera, quan una entitat sigui transportada a un node diferent, el motor ha de gestionar de forma transparent el canvi de jerarquia de la entitat cap al nou node.
- El motor ha de permetre la creació d'entitats de forma modular a partir de composició.
- Nodes i entitats han de poder moure's en tres grau de llibertat, permetent translacions en els tres eixos. A més, les entitats han de poder rotar al voltant del eix Z.
- El motor ha de permetre a les entitats moure's per nodes d'escala planetària sense problemes de precissió.
- Els planetes i cossos astronòmics del sistema solar seran simulats a escala real, on un bloc del joc equival a $1m^3$, però enlloc de ser esfèrics tindran forma de cilindre.
- Les entitats estaràn sotmeses a forçes de gravetat, fricció i flotabilitat.
- El motor ha d'estar publicat sota una llicència de codi lliure.

6.2 Requisits no funcionals

6.2.1 Requisits de maquinari

Per fer servir el motor del projecte cal disposar d'un ordinador que compleixi els següents requisits:

- 4 GiB de RAM
- 20 GiB d'espai lliure al disc
- Sistema operatiu Linux o Windows 10

A més a més, es recomana tenir una tarjeta gràfica dedicada, i un processador equivalent o superior a un Intel i5-6600. El motor ve preconfigurat amb projectes per a Visual Studio 2019 i Qt Creator, però també es pot compilar amb un senzill Makefile.
Per reproduïr la demo amb un rendiment acceptable cal disposar o bé d'una Nintendo Switch amb accés a *homebrew*, o un ordinador amb les següents característiques:

- nVidia GTX 960
- Intel i5-6600
- 8 GiB de RAM
- 20 GiB d'espai lliure al disc
- Sistema operatiu Linux

Tot i que la demo també pot ser executada sota Windows 10, el rendiment és substancialment inferior i, per tant, requereix millors especificacions per assolir el mateix rendiment que sota Linux.

6.2.2 Requisits de programari

La demo del projecte conté tot el necessari per ser executada, i només cal engegar el binari. Per poder compilar el motor sota un entorn Linux cal tenir instalat el següent *software*:

- SDL2
- GNU GCC i G++
- GNU Coreutils
- GNU Make

A més, per compilar per a Nintendo Switch caldrà instalar el *toolchain* devkitPro devkitA64 junt amb libnx.

Capítol 7

Estudis i decisions

En aquest capítol farem una breu introducció a les llibreries i eines utilitzades per dur a terme el projecte. També veurem alguns dels motors de videojocs més utilitzats, i analitzarem les diferències respecte al nostre.

7.1 Motors

7.1.1 Unity

Unity és un motor de videojocs multiplataforma publicat per Unity Technologies l'any 2005. Des d'aleshores ha guanyat una gran popularitat, i ha passat a ser un dels gegants de la indústria. Amb l'objectiu d'oferir un motor accessible a més desenvolupadors, Unity va ser un dels promotors de l'explosió de videojocs *indie* al voltant del any 2007. Unity és de codi propietari tot i tenir alguna part publicada amb llicència de només referència, i ofereix subscripcions de pagament així com una versió gratuïta per a petits desenvolupadors.
El motor està construït al voltant d'un editor gràfic 3D, el qual exposa gran part de la funcionalitat de manera visual, evitant la necessitat de modificar codi directament. El llenguatge de programació que suporta actualment és `C#`.
Unity ofereix suport multiplataforma per a totes les plataformes de videojocs modernes. Estàn incloses Windows, Linux, Mac OS, Android, iOS, Nintendo Switch i Xbox One.
La llista de jocs famosos creats en Unity està continuament en expansió, a continuació llistem alguns d'ells:

- Kerbal Space Program
- Hearthstone
- Cuphead
- Cities Skylines
- Hollow Knight

- Enter the Gungeon
- Ori and the Blind Forest

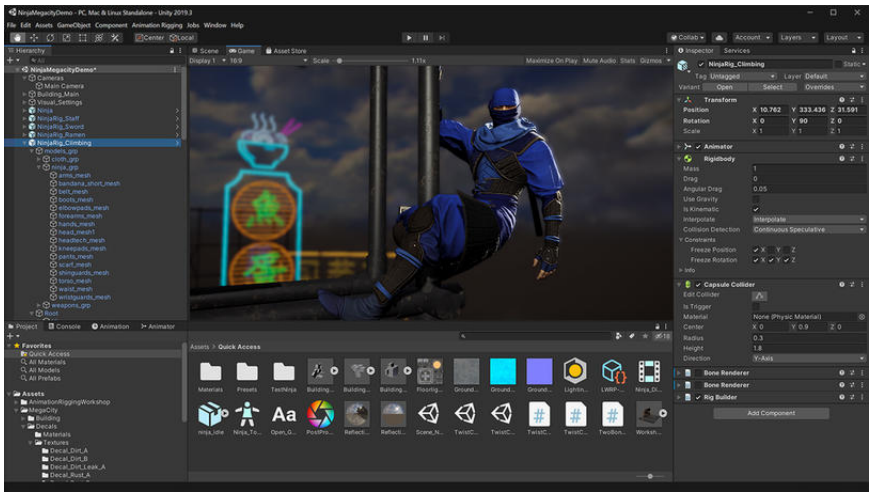


Figura 7.1: Captura de pantalla del editor Unity 2019.3

7.1.2 Unreal Engine

Unreal Engine és un motor de videojocs desenvolupat per Epic Games. Desenvolupat inicialment per Tim Sweeney l'any 1995 com a motor pel joc Unreal, l'Unreal Engine ha passat a ser un dels motors més utilitzats per a jocs AAA. El motor disposa d'una interfície gràfica des d'on es desenvolupen els jocs, i permet als desenvolupadors utilitzar codi C++, o fer servir el seu propi llenguatge de scripting visual anomenat Blueprint.
Unreal Engine ofereix tant una modalitat gratuïta per a petits desenvolupadors, com modalitats de pagament per a estudis més grans. La llista de jocs AAA que utilitzen Unreal Engine és immensa, però aquí llistem alguns dels més importants:

- Rocket League
- Fortnite
- PUBG
- Saga Gears of War
- Saga Bioshock
- Saga Borderlands

Com Unity, Unreal Engine ofereix suport multiplataforma per a Windows, Mac, Linux, Android, iOS, Nintendo Switch, PS4 i Xbox One.

Figura 7.2: Captura de pantalla del editor d'Unreal Engine 4

7.1.3 Godot

Godot és motor de videojocs originalment creat per Juan Linietsky i Ariel Manzur. Tot i ser comparativament molt més petit que Unity i Unreal, recentment està rebent grans actualitzacions que el poden apropar al nivell dels altres dos. A diferència d'Unity i Unreal, Godot és distribuït sota la llicència de codi lliure MIT, i és publicat en un repositori de Github. Ofereix una interfície gràfica per editar projectes, i tot i estar inicialment orientat a videojocs 2D, suporta també projectes en 3D, amb la major concentració del desenvolupament recent centrant-se en aquesta part. Pot ser programat a través d'un editor de scripts visual, C#, el seu propi llenguatge anomenat GDScript, o a través de GDNative, que és una interfície que permet interactuar amb llenguatges compilats tot i que només C++ està suportat de manera oficial. Oficialment suporta compilacions per a Windows, Mac OS, Linux, Android, iOS i web a través de emscripten. Tot i que oficialment no suporta cap consola, hi ha tercers que ofereixen suport per a Nintendo Switch, PS4 i Xbox One sempre que es disposi del SDK oficial.

7.1.4 Decisió

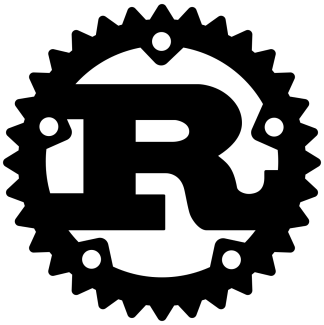
A cause a que cap dels motors de videojocs principals actuals compleix els principals objectius del nostre projecte, creiem que el nostre motor pot ocupar un espai que actualment no està cobert en el sector.

7.2 Llenguatges

7.2.1 C++

Originalment dissenyat com una extensió al llenguatge C, anomenada C amb classes, C++ ha esdevingut un dels llenguatges de programació principals en l'actualitat. Va ser dissenyat i creat per Bjarne Stroustrup, qui el va publicar al 1985 com si d'un preprocessador es tractés, el qual convertia el codi orientat a objectes de C++ en C clàssic. Al llarg dels anys, C++ s'ha convertit en un llenguatge genèric, funcional i orientat a objectes modern.
A part de ser un llenguatge de programació de propòsit general molt apte, C++ ha passat a ser el llenguatge de programació per defecte en l'indústria dels videojocs. Gràcies a la capacitat per barrejar codi a alt nivell amb codi a molt baix nivell i a la naturalesa com a llenguatge compilat, permet als programadors treballar a un alt nivell d'abstracció però sempre tinguent sota control el rendiment.

7.2.2 Rust



Rust és un llenguatge de programació dissenyat per Graydon Hoare a Mozilla, en part amb l'objectiu de presentar una alternativa amb millor gestió de memòria i seguretat respecte C++. A diferència de C++, el fet de no haver de ser compatible amb C suposa una gran llibertat per fer canvis estructurals que a C++ no serien possibles.

7.2.3 Decisió

Tot i que conceptualment Rust sembla una molt bona opció, la falta de maduresa del ecosistema el fa una pitjor elecció per al nostre objectiu. Si bé hi ha *toolchain* per compilar per Nintendo Switch, aquest és molt menys madur que el de C++, i la falta de llibreries orientades a motors de videojocs obligarien a acabar utilitzant llibreries de C++ a través de modes de compatibilitat binària, parcialment perdent les avantatges que Rust aportaria. És per això que es va decidir realitzar el projecte en el llenguatge C++, utilitzant l'últim estàndard disponible, C++17.

7.3 Llibreries

7.3.1 Multimèdia

7.3.1.1 SDL2

Simple DirectMedia Layer 2 és una capa d'abstracció de hardware multiplataforma. Ofereix al programador una API per gestionar audio, video, dispositius d'entrada i finestres entre d'altres. La seva funció és facilitar el desenvolupament de projectes multiplataforma, separant el codi específic de cada dispositiu a dins de la llibreria, i oferint crides homogènies a través d'una API. Tot i que la API és per a C, és perfectament utilitzable des de C++. Es distruibueix amb llicència Zlib, i suporta Linux, Windows, Mac OS, emscripten, iOS i Anrdoid entre d'altres. De forma no oficial també hi ha suport per a Nintendo Switch, desenvolupat per membres de la comunitat.

7.3.1.2 SFML

Simple and Fast Multimedia Library és una altre capa d'abstracció de hardware multiplataforma similar a SDL. Es diferencia respecte SDL en ser lleguerament més orientada a C++. També és publicada sota llicència Zlib, però actualment no existeix suport per a Nintendo Switch.

7.3.1.3 Decisió

A causa de la falta de suport per a Nintendo Swich de SFML, hem decidit triar SDL2 com a capa multimèdia pel projecte. Com que nosaltres també afegirem una capa d'abstracció de hardware pròpia al projecte, podríem haver escollit SFML per la versió de PC i SDL2 per a Nintendo Switch, però hagués suposat un esforç extra que no hauria aportat cap benefici al projecte.

7.3.2 FastNoise

FastNoise és una llibreria de generació de gradients de soroll per C++. Permet de forma senzilla generar sorolls Value, Perlin, Simplex, Celular i Blanc tant en formats 2D com 3D. També permet soroll Simplex en 4D, i habilita la combinació de diferents nivells de soroll amb moviment fraccional Brownià.

7.3.3 EnTT

EnTT és un framework de gestió de Entity Component System per C++. Està publicat sota la llicència MIT, i està en actiu desenvolupament a Github. A part del ECS també inclou diverses altres eines, però nosaltres farem ús exclusiu del ECS. Entre d'altres, és utilitzat per la versió Bedrock de Minecraft.

7.3.4 ReactPhysics3D

ReactPhysics3D és un motor de física per a simulacions i jocs en C++. Està en actiu desenvolupament a Github, i és publicat sota la llicència MIT. Tot i que RP3D ofereix la possibilitat de fer tots els càlculs de física de forma autònoma, nosaltres només l'utilitzarem per a detecció de col·lisions i nosaltres farem la resta de càlculs degut a la naturalesa del motor.

7.3.5 nlohmann/json

JSON for Modern C++, conegut popularment per la seva adreça de Github com a nlohmann/json és una lleugera llibreria que facilita la manipulació d'arxius JSON des de C++. Està en actiu desenvolupament i es troba publicat sota la llicència MIT.

7.3.6 IceCream-Cpp

IceCream-Cpp és una molt petita llibreria que assisteix a l'hora de fer debugging a través de la consola. Alguns errors, per la seva naturalesa, són més fàcils de diagnosticar i arreglar a través de text a la consola en una configuració Release, que no pas amb un debugger. Aquesta llibreria va ser afegida a la meitat del projecte al trobar-la per casualitat, i descobrir que podia ser útil.

7.3.7 DevkitA64

DevkitA64 és un *toolchain* publicat per l'equip de DevkitPro que permet, junt a libnx, compilar projectes C++ per a la Nintendo Switch. Es troba sota la llicència ISC, que és funcionalment equivalent a BSD 2C i MIT.

Capítol 8

Anàlisi i disseny

En aquest capítol explicarem el disseny del projecte. S'ha dividit en dues parts per facilitar l'organització. La primera part explica la interfície, el sistema d'estats del motor, i el editor de prefabs. La segona explica l'estat de joc on el jugador té control del personatge. També hi ha una secció addicional per explicar el funcionament de les llibreries utilitzades, així com altres eines petites que poden ser difícils d'ubicar en cap de les dues parts principals. A la figura 8.1 es mostren la majoria de classes del projecte, i les principals relacions entre elles.

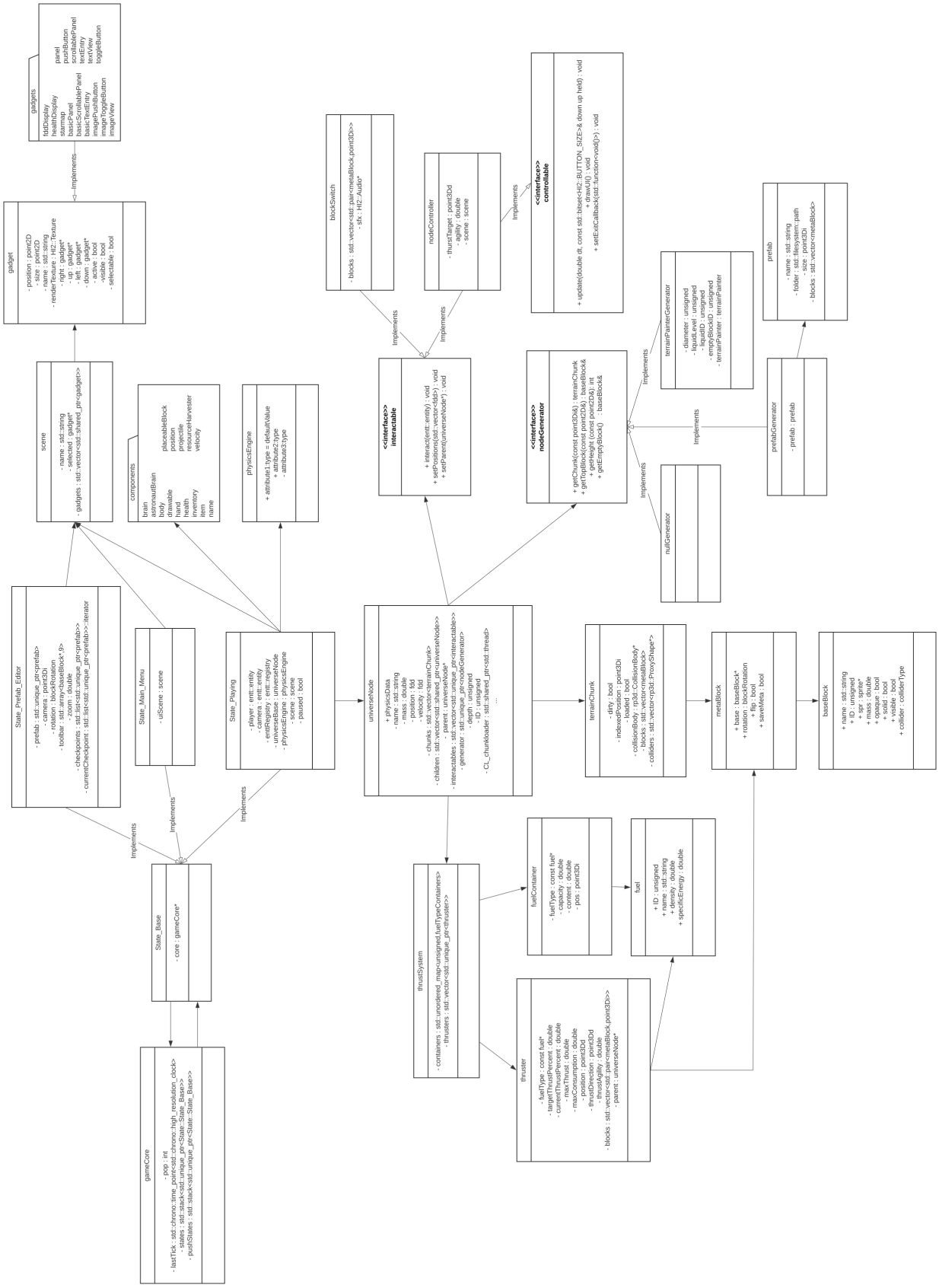


Figura 8.1: Diagrama de classes general

8.1 Interfície i menús

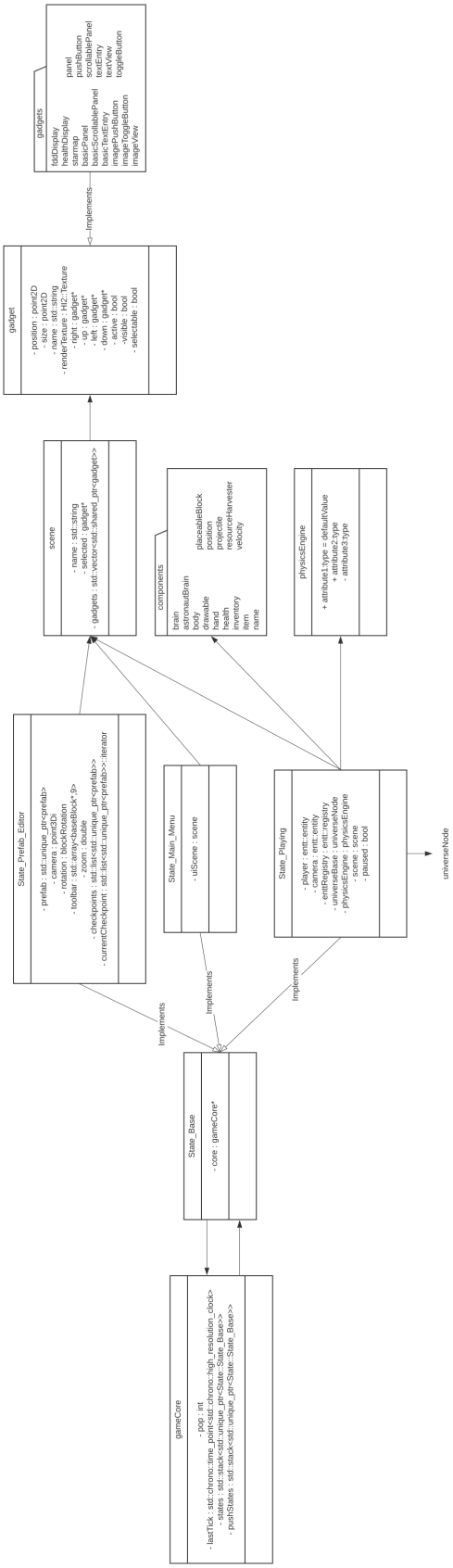


Figura 8.2: Diagrama de classes de l'interfíce i menús

En aquesta part es detallen les classes i el funcionament de la part menys relacionada amb el joc en sí. La separació és complicada de definir, però aqui s'explica el funcionament del bucle principal, els menús, el sistema d'estats i l'editor de 'prefabs'. La funció main del nostre motor tant sols crea una instància d'un objecte gameCore, i acte següit crida la funció gameCore.gameLoop(), així que podríem considerar gameCore.gameLoop() com la nostre funció principal.

8.1.1 gameCore

La classe gameCore és l'encarregada de gestionar els diferents estats del motor. Simplement conté una pila d'estats, i la seva funció que fà de bucle principal s'encarrega de cridar les funcions de input, càlcul i dibuix del estat que estigui al cim de la pila. Els pròpis estats podràn, aleshores, cridar funcions de gameCore per treure o afegir estats a la pila.

8.1.2 States

Els estats, o més concretament, State_Base, és una classe abstracte que representa una escena o estat del motor. Conté mètodes per rebre input, actualitzar l'estat i dibuixar l'estat. Exemples d'estats possibles podrien ser l'estat del menú principal, un estat per modificar les opcions del joc, l'estat de gameplay en sí, o un estat de pausa dins del gameplay. A part dels estats que explicarem a continuació hi ha l'estat del joc (State_Playing) que serà explicat al apartat de simulació del joc.

8.1.2.1 Main Menu

Main_Menu és l'estat que representa el menú principal. Conté molt poca lògica, ja que simplement és interfície gràfica per entrar a altres estats del motor.

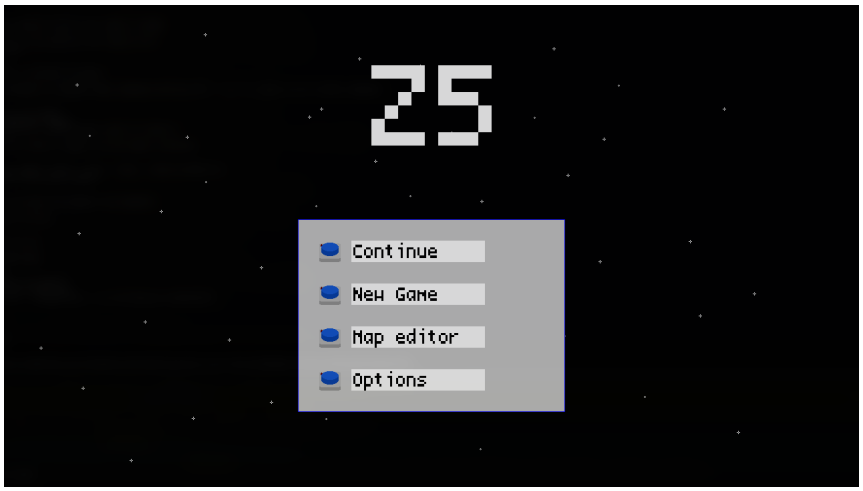


Figura 8.3: Captura de pantalla del menú principal.

8.1.2.2 Prefab Editor

Prefab_Editor és l'estat per l'editor de prefabs del motor. Té un funcionament similar a una èina de dibuix informàtica, on l'usuari pot omplir una graella de diferents colors, en aquest cas blocs, arrastrant el cursor per la pantalla. Disposa d'eines de pinzell, esborrador, cub de pintura i selectors. També incorpora funcionalitats per copiar, tallar, enganxar, desfer i refer canvis.
Inclou una petita interfície d'usuari per sobre el llenç de dibuix, on es mostren els blocs que

tenim disponibles per decorar el prefab. Tot i així, manca molt de feedback visual, ja que la majoria d'interaccions s'expressen a través de la consola mitjançant text, i les eines es seleccionen utilitzant draçeres de teclat.

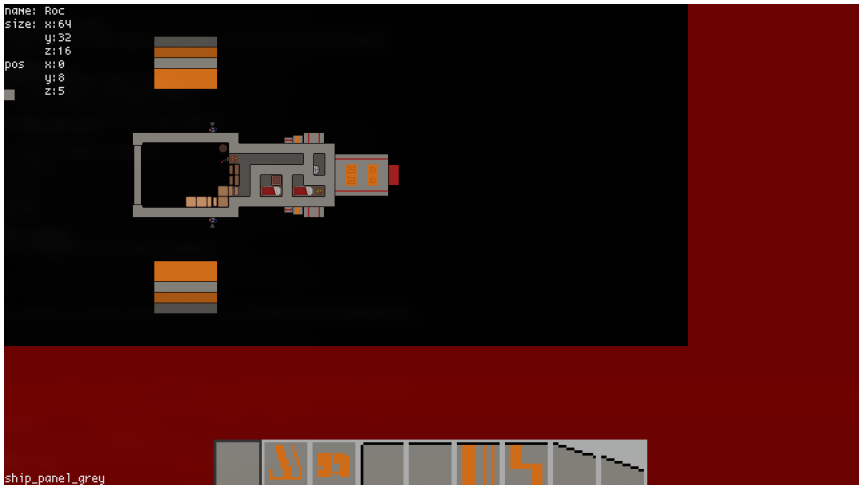


Figura 8.4: Captura de pantalla del editor de prefabs.

8.1.3 UI

Per l'interfície d'usuari s'ha implementat un petit framework fet des de zero prou senzill, amb l'objectiu de permetre al usuari interactuar tant amb teclat i ratolí com amb controls o comandaments de consola.
La interfície d'usuari únicament disposa de dos components diferenciats.

8.1.3.1 gadget

Un gadget és una classe abstracte que representa un element qualsevol de la interfície d'usuari. Els gadgets s'encarreguen de gestionar la seva posició i tamany, la visibilitat, el nom, i mantenen enllaços als gadgets que tenen al voltant per poder-hi accedir-hi direccionalment. Els gadgets que hi ha actualment al motor són:

- panel: Gadget que representa un panell rectangular on es poden afegir altres gadgets.
- pushButton: Gadget que representa un botó, amb el qual es poden prendre accions al ser premut.
- textView: Gadget que representa una caixa de text fixe.
- textEntry: Gadget que representa una caixa d'entrada de text.
- toggleButton: Gadget que representa un interruptor.
- imageView: Gadget que permet mostrar una imatge en pantalla.
- scrollablePanel: Gadget que especialitza un panell, i afegeix funcionalitat per un panell on existeix un 'tamany intern' major al tamany del panell, amb el qual s'afegeix l'habilitat de desplaçar-nos per dins del panell.
- basicPanel: Gadget que especialitza un panell, i afegeix un fons sòlid al panell original.
- basicScrollablePanel: Gadget que especialitza un scrollablePanel, i afegeix un fons sòlid al gadget original.

- basicTextEntry: Gadget que especialitza un textEntry, i afegeix un fons sòlid al gadget original.
- imagePushButton: Gadget que especialitza un pushButton, i afegeix suport per que el gadget es mostri com a una imatge, amb la possibilitat de canviar d'imatge al mantenir-se apretat.
- imageToggleButton: Gadget que especialitza un toggleButton, i afegeix suport per que el gadget es mostri com a una imatge, canviant d'imatge segons l'estat del interruptor.
- fddDisplay: Gadget que representa un fdd de forma gràfica. Principalment utilitzat per propòsits de desenvolupament.
- healthDisplay: Gadget que permet mostrar per pantalla la salut d'una entitat.
- starmap: Gadget que mostra un mapa galàctic al voltant d'un cos astronòmic. Permet visualitzar les òrbites previstes d'un cos i les seves propietats (nom, massa i diàmetre)

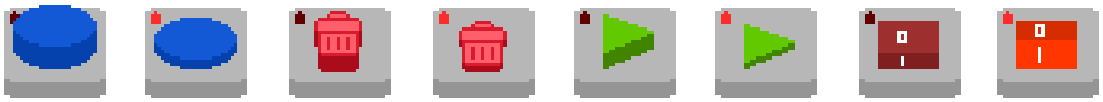


Figura 8.5: Botons i interruptors de la interfície en estats actiu i inactiu.

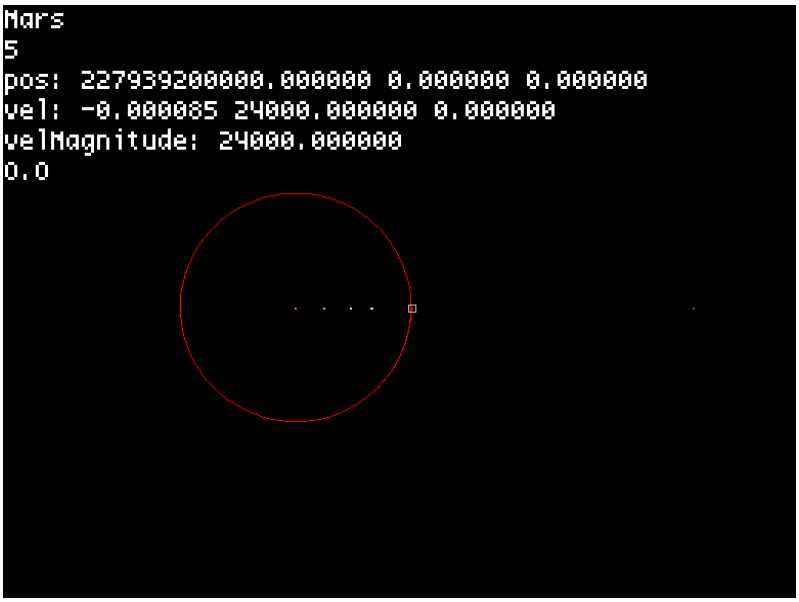


Figura 8.6: Captura d'un starmap mostrant l'òrbita de Mart.

Figura 8.7: Captura d'un healthDisplay amb quatre cors plens de cinc màxims.

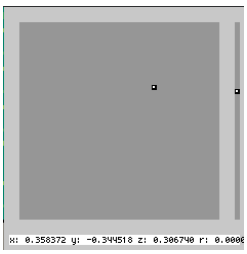


Figura 8.8: Captura d'un fddDisplay representant un vector amb direcció +X +Y +Z.

8.1.3.2 scene

L'escena és poc més que un contenidor de gadgets, s'encarregarà de mantenir un gadget com a seleccionat, i fer les crides recursives per dibuixar i interactuar amb els gadgets que la composen.

8.2 Simulació del joc

En aquesta part es detalla el disseny de la part de gameplay i simulació del motor. A grans trets, podríem dir que la classe principal és State_Playing, que serà l'encarregada de rebre entrada, simular, i mostrar per pantalla l'acció del joc.

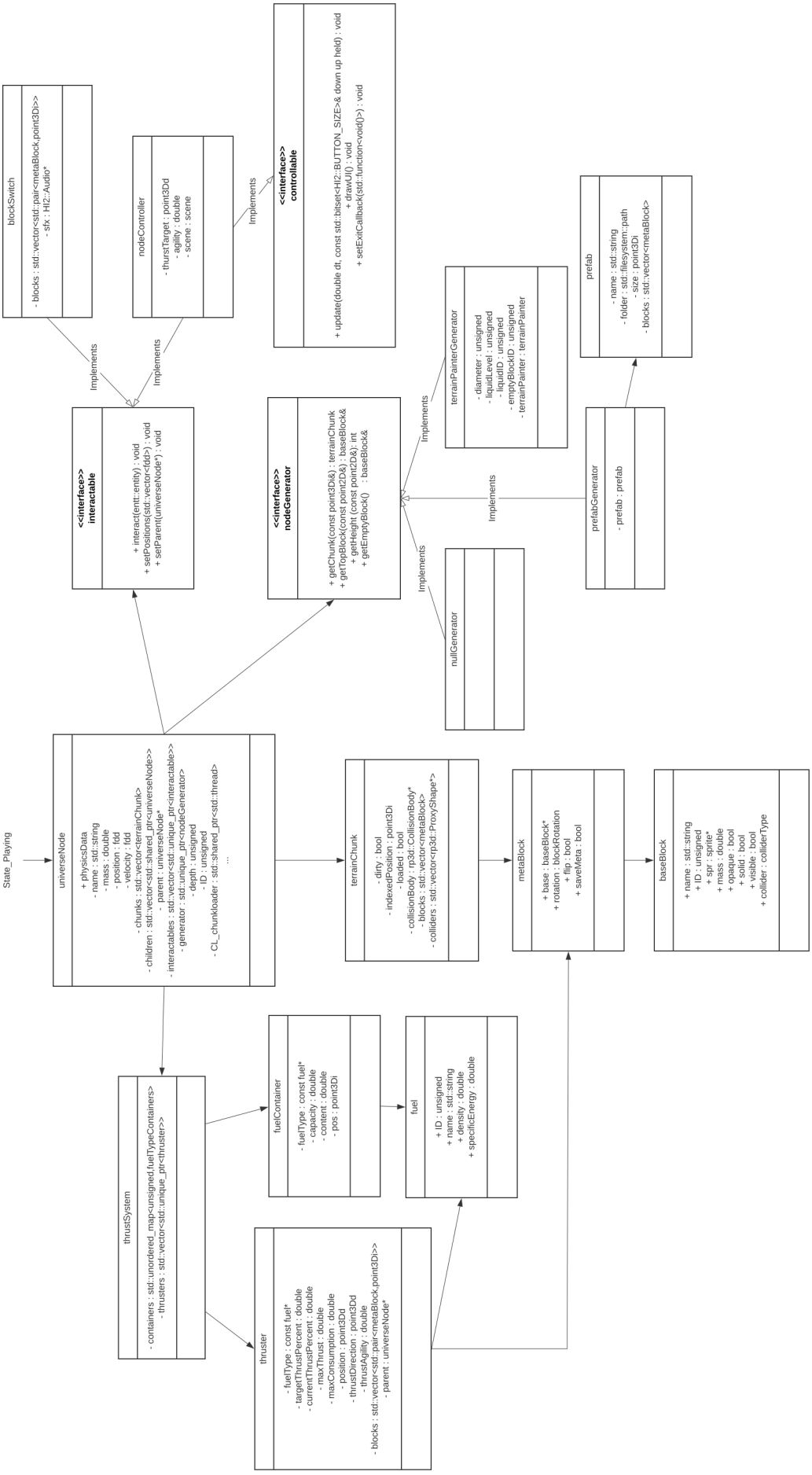


Figura 8.9: Diagrama de classes de la part de simulació.

8.2.1 Escena del joc State Playing

State_Playing és l'escena principal del joc, i és l'encarregada de gestionar els diferents sistemes de la simulació. Conté l'escena de la UI, el registre d'entitats i el node base de la simulació. A continuació es detallen les seves funcions més importants.

8.2.1.1 Renderitzat

Tot i que el nostre motor i simulació es realitzen en un espai tridimensional, el renderitzat es farà utilitzant sprites en 2D. Per tal de donar un efecte de profunditat, el procés de renderitzat consistirà en dibuixar els nodes capa a capa, aplicant parallatge a cada capa segons la seva distància amb la càmera. D'aquesta manera, les capes més profundes i llunyanes a la càmera es dibuixaran abans i amb un menor zoom, tal que al moure la càmera el moviment en píxels d'aquella capa sera menor i donara l'efecte de ser lluny.
Es podran renderitzar tres tipus d'elements, nodes, entitats visibles i requadres. Per gestionar les diferents profunditats es mantindrà una llista de capes a dibuixar, que seran posteriorment ordenades segons la seva profunditat. Els nodes es renderitzaran iterant per les capes del terreny i afegint-les a la llista. Tot node de la simulació serà considerat per renderitzar, però només ho serà en cas d'estar a una distància de la càmera que permeti que aparegui per pantalla. Seguidament s'afegiran a la llista totes les entitats dibuixables que es trobin en una alçada dibuixable per la càmera, i que estiguin a una distància raonable com per ser dibuixades. Finalment, s'afegirà un requadre al voltant del ïnteractable"més proper al jugador, en cas que n'hi hagi cap a l'abast.
Tot seguit s'ordenaràn els elements a dibuixar afegits a la llista segons la seva profunditat respecte la càmera, en ordre de més llunyà a més proper. Finalment començarà el procés de renderitzat pròpiament dit. Entitats i requadre són simples de dibuixar, tant sols cal aplicar una transformació per convertir la seva posició relativa a la càmera en posició respecte l'orígen de la pantalla, i aplicar zoom segons la seva profunditat a l'hora de dibuixar.
El renderitzat de capes de terreny serà lleugerament més complex. El procés consisteix en dibuixar els blocs de la capa un a un, iterant per aquells que es trobin dins el camp de visió de la càmera, aplicant efectes d'ombres i oclusió ambiental en cas de ser necessari. Als blocs de les capes s'apliquen tres tècniques de pre/post-processat. A cada bloc se li és calculada la seva visibilitat, a partir de la qual es decidirà si dibuixar o no el bloc. Aquesta visibilitat és precalculada al carregar-se el chunk que la conté. Es considera un bloc com a visible si qualsevol dels nou blocs situats en un quadrat de 3x3 centrat en el bloc de sobre el bloc actual és visible. També es precalcula un efecte de oclusió ambiental, amb el que s'apliquen diferents ombres als blocs en funció dels blocs que té situats a sobre. Finalment, a cada capa es calcula un ombrejat general segons la seva profunditat, donant tons més foscos a les capes més profundes i ajudant al jugador a entendre la profunditat de les capes.

8.2.1.2 Entrada

Per processar l'entrada simplement es passen les tecles i interaccions rebudes cap a l'escena de la UI, i s'envien també cap al component "brain"del jugador en cas que en tingui.

8.2.1.3 Actualització

En la fase d'actualització s'executen els sistemes del sistema entitat component, s'actualitzen els propulsors dels nodes, es netejen i s'adormen/desperten les entitats que escaiguin, i finalment es fan els càlculs de física i s'actualitza la càmera per evitar col·lisions.

8.2.2 universeNode

La classe universeNode servirà per representar els cossos astronòmics en la simulació (els anomenats nodes). Els nodes tenen una estructura jeràrquica, on cada node conté els seus nodes fills, que són els nodes que l'estan orbitant. Els nodes contenen el terreny del joc, que està compost per multiples agrupacions de blocs anomenades chunks. Cada node conté una matriu 3D de chunks, on els chunks seràn carregats en temps real mitjançant un fil paral·lel d'execució. Els nodes mantenen informació sobre la seva massa, diàmetre, el node pare, la seva posició i velocitat respecte al seu pare, el seu sistema de propulsió, el seu generador i la llista de ïnteractables"que contenen.

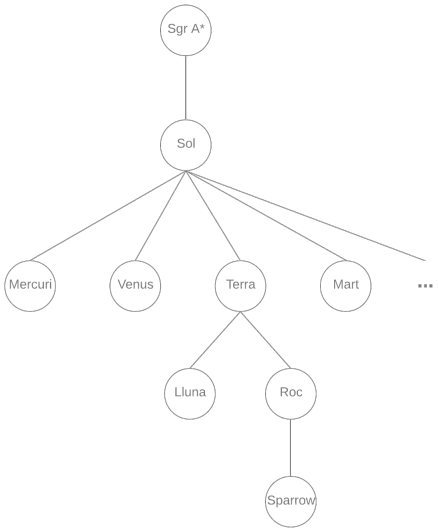


Figura 8.10: Exemple de jerarquía de universeNodes. Sagitari A* és el node arrel, i cada node conté els seus nodes inferiors. La Terra conté la Lluna i la nau grossa (Roc), i dins la nau grossa tenim la petita (Sparrow).

8.2.2.1 Reparentització

Per tal de recalcular la jerarquía dels nodes, els nodes tenen un mètode que els permet calcular el millor pare donada una posició i un pare actual. Aquest càlcul es fa mitjançant esferes d'influència, que són un concepte utilitzat en astrodinàmica per calcular el principal cos al que un cos astronòmic orbita. A part de les esferes d'influència, tenen prioritat els nodes en els quals la posició donada correspongui a un bloc no buit.

8.2.2.2 chunkLoader

Els chunkLoaders són fils d'execució que cada node conté. S'encarreguen de carregar nous chunks al voltant del jugador per tal que el jugador pugui moure's pel terreny sense cap pantalla de càrrega. El funcionament a alt nivell és el següent:

1. Es llegeix la posició del jugador respecte al node, la qual és calculada des de fora el fil i transmesa a través d'una variable compartida.

2. En cas que la posició del jugador no hagi canviat de chunk, el chunkLoader adorm el fil durant un temps.

3. En cas que que sí, els chunks més llunyans al jugador seràn carregats, en cas de ser a disc es carregaran des de disc, en cas contrari seran generats pel nodeGenerator.

Per determinar on posicionar un chunk dins la matriu de chunks s'obté la posició del chunk en el món, es divideix cada eix entre el tamany dels chunks, i finalment s'obté la posició dins de la matriu aplicant el mòdul del tamany dels costats de la matriu 3D. D'aquesta manera s'aconsegueix un efecte similar a un buffer circular en tres dimensions.

8.2.3 terrainChunk

Els terrainChunks són agrupacions cúbiques de blocs. Actualment són cubs de vuit blocs per costat, resultant en 512 blocs per chunk, tot i que el tamany dels chunks és un senzill paràmetre fàcil de canviar. Els blocs, més concretament metaBlocks, es guarden en un vector i s'accedeix als blocs com si estiguessin en una matriu tridimensional. Els chunks poden ser serialitzats i deserialitzats cap a disc, i es poden crear utilitzant el nodeGenerator del node que els contingui.

8.2.4 Blocs

Els blocs seràn el material amb el que el terreny del joc estarà construït. Representen un bloc de un metre cúbic. Dins del motor es fan servir dues classes per representar els blocs.

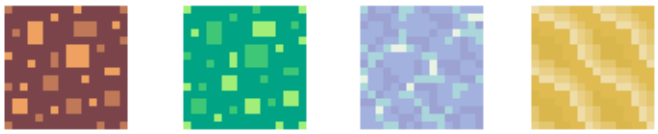


Figura 8.11: Blocs de terra, gespa aigua i sorra.

8.2.4.1 baseBlock

El struct baseBlock representa un bloc imaginari amb unes certes propietats. Les propietats inclouen nom, ID, visibilitat, si és sòlid, opacitat, massa, colisionador i sprite.

8.2.4.2 metaBlock

Els metaBlock en canvi representen un bloc real d'un node. Contenen una referència al baseBlock que representen, pero a més contenen informació sobre la rotació del bloc, visibilitat, i oclusió ambiental. Els chunks de terreny estàn doncs formats per multiples metaBlocks, els quals fan referència a un baseBlock que no es pot modificar. Podríem considerar aquesta separació com a una aplicació del patró *flyweight*, el qual ens permet evitar informació redundant de manera dràstica.

8.2.5 Interactables

Interactable és una classe abstracte per a blocs especials amb els que les entitats hagin de poder interactuar. Són un concepte similar a entitats, però amb la diferència que estan relacionades amb un o múltiples blocs, i estan emmagatzemades en els nodes enlloc de pertànyer a un registre global. Un mateix interactable pot tenir múltiples posicions des de les quals pot ser accionat.

8.2.6 nodeController

Els nodeController són un tipus de interactable que permeten a una entitat controlar el sistema de propulsió d'un node. Permeten a la entitat establir un objectiu de propulsió, i el sistema de propulsió serà dirigit per tal que el node es mogui cap aquell objectiu.

8.2.7 blockSwitch

Els blockSwitch són un interactable que permet a una entitat canviar un grup de blocs per un altre. Aquest interactable serveix principalment per a fer portes i interruptors per a portals.

8.2.8 Entitats i components

Com ja s'ha introduït anteriorment, les entitats son part del patró entity component system que fem servir per modelar els actors i objectes del joc. En la majoria de casos son senzilles estructures que guarden informació, com per exemple la salut del personatge, però no estan limitades a només això.

8.2.8.1 brain

El brain és una classe abstracte que representa un component que controla una entitat. Aquest brain és actualitzat regularment pel sistema, i en cas de pertànyer al jugador s'actualitzarà a partir del botons i tecles que l'usuari hagi apretat. En cas contrari, els brain poden implementar funcionalitat per actuar com a inteligència artificial per les entitats. Actualment tant sols hi ha una classe que implementi brain, la qual explicarem en el següent punt.

8.2.8.2 astronautBrain

El component astronautBrain és una classe que implementa brain. Conté una màquina d'estats que exposaràn diferent funcionalitat al jugador depenent de cada situació. Per exemple, al estar en contacte amb el terra la entitat es considerarà "grounded", i tindrà accés a caminar i saltar. En saltar, el brain passarà a un estat on està saltant, i comença a incrementar la seva velocitat respecte l'eix vertical. Tot seguit, al perdre contacte amb el terra el brain passarà a estat "airborne"o volant/flotant, i tindrà accés a uns petits propulsors que permetràn al usuari controlar el personatge lleugerament, en cas d'estar volant o flotant a l'espai. Quan el personatge torni a estar en contacte amb el terra tornarà a estar al estat "groundedï tornarà a tenir accés a les opcions d'aquell estat. Aquesta màquina d'estats permet modelar comportaments prou complicats, on cada estat pot portar a diversos altres estats depenent de l'acció que s'hagi près. Actualment aquest component no implementa cap intel·ligència artificial per a entitats que no siguin el jugador.

8.2.8.3 body

Body és una estructura que representa un cos físic esfèric. Conté informació sobre el seu diàmetre, massa, volum i elasticitat. Aquest cos aleshores serà utilitzat pel motor de física per calcular col·lisions i efectes similars.

8.2.8.4 drawable

Drawable és una estructura que representa un sprite dibuixable. Conté a dins una referència a un sprite, el nom del sprite i una variable zoom que permet ampliar o reduïr el tamany del sprite.

8.2.8.5 hand

Hand és una estructura que modela una mà com a recipient per a un objecte. Una mà pot estar agafant un objecte, i en cas de ser accionable, l'objecte podrà ser accionat.

8.2.8.6 health

Health és una classe que representa la salut de la entitat. La entitat que tingui component de health podrà ser danyada i curada, i en cas d'arribar a zero salut, la entitat serà esborrada del joc.

8.2.8.7 inventory

Inventory és una classe que modela un inventari, que és una col·lecció d'objectes. Aquests objectes podràn fer-se servir a través de la mà, en cas que la entitat en tingui.

8.2.8.8 item

Item modela un objecte, o múltiples objectes del mateix tipus. Aquests objectes poden estar caiguts al terra, o bé estar al inventari d'una altre entitat.

8.2.8.9 name

Component que modela el nom d'una entitat. Actualment no té cap ús dins del joc, i només es fa servir en alguna comanda de *debugging* per identificar entitats.

8.2.8.10 placeableBlock

Item interactuable que permet posicionar un bloc cap on la entitat que l'acciona està mirant.

8.2.8.11 resourceHarvester

Item interactuable que permet extreure blocs del terreny, convertint-los en items que les entitats poden recollir i col·locar.

8.2.8.12 position

Estructura que modela la posició d'una entitat. Al igual que amb els nodes, la posició és relativa a un node pare, i la posició en sí és del tipus fdd.

8.2.8.13 projectile

Component que modela un projectil que causa dany a les entitats amb que impacta. Emmagatzema informació sobre el dany del projectil, les vegades que encara pot rebotar en el terreny, els enemics que encara pot atravessar, i la ultiam entitat amb qui ha impactat, per tal de no registrar multiples impactes en la mateixa entitat.

8.2.8.14 velocity

Component que modela la velocitat de la entitat. Tot i que és relativa al node pare, no emmagatzema cap referència al pare, ja que ja hi és en el component posició.

8.2.9 thrustSystem

El thrustSystem és una classe encarregada de modelar tot el sistema de propulsió d'un node. Gestiona els propulsors i contenidors de combustible del node. S'ocupa de gestionar la consumpció i recàrrega de combustible, tinguent en compte els diversos tipus de combustible i els contenidors corresponents a cada tipus. També s'encarrega d'alimentar els propulsors i gestionar la seva potència per tal de dirigir el node cap al objectiu.

8.2.9.1 fuel

El fuel és una senzilla estructura que representa un tipus de combustible. Conté un identificador, un nom, la densitat en kg/m^3 i la energía específica en MJ/kg.

8.2.9.2 fuelContainer

Un fuelContainer representa un contenidor d'un cert tipus de combustible. Manté la seva capacitat màxima, el contingut actual, el tipus de combustible que admet i la posició on es troba.

8.2.9.3 thruster

Un thruster modela un propulsor d'un cert tipus de combustible. Emmagatzema informació sobre el tipus de combustible que admet, el seu objectiu de potència, la seva potència actual, la propulsió que ofereix al estar a màxima potència, i la quantitat de combustible que consumeix a màxima potència, a més de la seva posició, agilitat i orientació. El seu canvi de potència no és instantàni, sinó que es modela a partir de la potència actual i la agilitat del propulsor. D'aquesta manera, al intentar encendre els propulsors a màxima potència aquests han de augmentàr de potència fins arribar a la màxima.

8.2.10 prefab

La classe prefab representa un conjunt de blocs prefabricat. És creat amb l'editor de prefabs, i després utilitzat pels nodes que utilitzin un prefabGenerator. Conté el seu nom, les seves dimensions i el vector de metaBlocks que el composen.

8.2.11 nodeGenerator

NodeGenerator és una classe abstracte que serveix per representar un generador de node. Aquest generador serà cridat pel node quan aquest necessiti carregar un chunk que no hagi trobat a disc. En aquest cas el nodeGenerator fabricarà un chunk i li donarà al node. També ofereix mètodes per obtenir el bloc de la superfície en un punt concret, l'alçada del terreny en un punt, o per saber quin és el bloc que s'utilitza per l'espai buit en aquell node. Actualment hi ha tres tipus de generadors implementats, que es detallen a continuació.

8.2.11.1 nullGenerator

NullGenerator és un nodeGenerator *stub*, que serveix principalment per a propòsits de *debugging* o per aquells nodes que siguin construïts per un jugador.

8.2.11.2 prefabGenerator

PrefabGenerator és un nodeGenerator que genera chunks a partir d'un prefabricat. Això ens serveix per generar estructures fetes a mà com per exemple naus, estacions espacials, ciutats, asteroides o qualsevol altre tipus de node que un dissenyador pugui crear a mà. Tot i qué té gran utilitat per crear estructures estàtiques, no és capaç de generar estructures amb portes o blocs interaccionables, ja que aquests no es troben a nivell de chunk sinó a nivell de node.

8.2.11.3 terrainPainterGenerator

TerrainPainterGenerator és el principal generador per a terreny del motor. Genera cossos astronòmics a escala real però de forma cilíndrica. Funciona a partir de fer un mostreig al gradient de soroll a partir de la posició 2D del bloc a calcular. Un cop es té el nivell de soroll en aquell punt, s'utilitza una llista de seccions de terreny a les quals el soroll està mapejat, per tal d'obtenir el bloc que tocarà a aquella alçada tinguent en compte el soroll de la posició 2D.

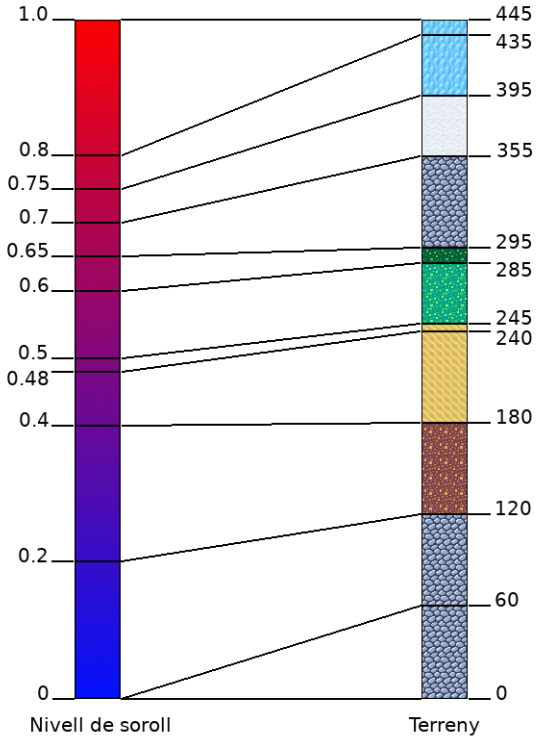


Figura 8.12: Representació del mapeig entre soroll i terreny al planeta Gaia.

8.2.12 physicsEngine

PhysicsEngine és la classe que representa el motor de física, i és la encarregada de fer tots els càlculs de física del motor. És filla de la classe rp3d::CollisionCallback degut a necessitats per utilitzar la llibreria de física. En totes les col·lisions s'aplicarà un petit factor de pèrdua d'energia per simular l'energia convertida en calor, i així evitar sistemes de moviment perpetu.

8.2.12.1 Gravetat

El càlcul de gravetat s'aplica a tot node i entitat de la simulació. La lògica és la següent: Si el node té gravetat artificial, es torna directament com a acceleració resultant la establerta segons la gravetat artifical. En cas contrari, es calculen dues gravetats, la "màgicaï la real. La real és la gravetat normal calculada utilitzant la fòrmula de la gravitació universal de Newton $F = G\frac{m_1 m_2}{r^2}$. A partir de la força i la massa del objecte obtenim una acceleració utilitzant $F = ma$. També és calculada la gravetat "màgica", que és la equivalent a la gravetat real a la superfície del planeta en cas que aquest fós esfèric. La màgica, però, sempre és en direcció cap a la Z negativa, per tal que els planetes cilíndrics no arrastrin els cossos cap al centre, sinó que els atreguin cap al terra (-Z). Tot següit es calcula un factor de "màgia", que decidira quin percentatge de gravetat màgica i quin de real s'aplica al cos. Aquest factor és sempre 1 mentre

el cos estigui en la superfície del node, però ràpidament decau cap a 0 linealment al sortir del perímetre del cos. D'aquesta manera obtenim una gravetat que va cap a -Z al estar a sobre la superfície del planeta o cos, però tenim gravetat real que va cap al centre de massa del cos al sortir de la superfície d'aquest, habilitant la possibilitat d'orbitar qualsevol cos amb suficient massa.

8.2.12.2 Propulsors

Per calcular l'efecte dels propulsors en els nodes simplement hem de sumar l'efecte de tots els propulsors. Per obtenir l'efecte de cada propulsor hem de multiplicar la seva potència màxima en Newtons pel percentatge de potència actual. Amb aquesta força podem aplicar $F = ma$ per obtenir la acceleració resultant.

8.2.12.3 Flotabilitat

Per calcular la flotabilitat de cossos i entitats fem servir la fòrmula $F_b = \rho f V g$, on Fb és la força resultant, ρf és la densitat del fluid, V és el volum de fluid desplaçat i g és la acceleració gravitacional del cos submergit. Per les entitats podem trobar el volum a través de les dimensions del seu body. Per els nodes aproximem el volum utilitzant el seu diàmetre i assumint que té forma d'esfera. Pot semblar una aproximació poc refinada, però els resultats que dóna son suficientment bons. Tot i que va ser esborrada, la flotabilitat va permetre afegir una funció amb la que els personatges podien controlar la seva flotabilitat dins l'aigua a través d'aguantar la respiració, i per tant augmentar o disminuïr el seu volum.

8.2.12.4 Fricció aerodinàmica

Pels càlculs de fricció aerodinàmica fem servir la fòrmula $F = 0.5\rho u^2 C_p A$, on F és la força resultant, ρ és la densitat del fluid, u és la velocitat del flux respecte l'objecte, A és l'àrea on s'aplica la fricció, i C_p és el coefficient de fricció. La velocitat del plux s'assumeix com a igual a la velocitat del objecte submergit, l'àrea és calculada com la meitat d'una esfera, i el coefficient de fricció utilitzat és 0.25, que aproxima un cos suficientment aerodinàmic.

8.2.12.5 Velocitat

Amb les noves velocitats calculades només resta calcular les noves posicions de nodes i entitats a partir de la velocitat del cos i el temps transcorregut. A part d'aplicar aquesta velocitat per trobar la nova posició, de forma desacoblada al motor de física es calculen posicions extrapolades que permeten mostrar moviments suaus en pantalla tot i que la taxa d'actualització de posicions sigui de tant sols 20Hz.

8.2.12.6 Col·lisió entre nodes

Per calcular i resoldre les col·lisions entre nodes el que farem serà iterar per tots els nodes. A cada node comprovarem si és possible que col·lisioni amb el seu node pare, o amb els seus nodes germans, revisant si les seves AABBs es superposen, i en cas afirmatiu comprovarem les col·lisions dels chunks carregats dels nodes. En cas que s'hagi produït una col·lisió, resoldrem les posicions i velocitats a partir de la normal del contacte, la velocitat del objecte petit i la profunditat del contacte. Mourem cap enrere el cos petit en direcció contrària a la seva velocitat fins que estigui just tocant el segon cos, aleshores calcularem la direcció de la nova velocitat reflectint la antiga respecte la normal del contacte. Finalment compensarem el moviment que hem retrocedit aplicant-lo de nou amb la nova velocitat. No farem cap tipus de modificacions en l'objecte gran, ja que en la majoria de casos l'efecte seria negligible, i podria causar problemes al

no simular la gravetat dels cossos fills respecte al pare, ja que un cos petit a la superfície d'un cos amb suficient gravetat acabaria accelerant al propi cos que genera la gravetat cap a la direcció d'aquesta. En una estructura de nodes plana, aquesta fase és $\mathcal{O}(\frac{n^2}{2})$ ja que comprovariem tots els nodes entre sí, però no repetiriem les comprovacions simètriques.

8.2.12.7 Col·lisió entre entitats

En les entitats seguirem un procés similar al dels nodes, però farem la simulació completa d'un xoc elàstic. Això vol dir que es recalcularan les posicions i velocitats de les dues entitats que hi intervinguin. Comprovarem col·lisions entre entitats properes, i com amb els nodes, evitarem comprovar dues vegades la mateixa col·lisió revisant que l'identificador de la primera entitat sigui més gran que la de la segona. Aquesta fase també és $\mathcal{O}(\frac{n^2}{2})$.

8.2.12.8 Col·lisió entre entitat i node

Per les col·lisions entre entitats i nodes comprovarem la col·lisió amb el node pare de la entitat, i tots els fills d'aquest. En cas de que les AABB es superposin aleshores es farà la comprovació més detallada respecte als chunks de terreny. Només modificarem posició i velocitat de la entitat, ja que gairebé sempre tindrà una massa extremadament inferior a la del node.

8.3 Utilitats

En aquest apartat analitzarem certes classes i utilitats senzilles que no poden existir de forma individual, o que es fan servir en tot el motor, i per tant, no són classificables com a *gameplay* o interfície.

8.3.1 fdd

Els fdd són una estructura que representen una posició tridimensional amb una rotació. Utilitzen nombres de coma flotant per representar amb gran precisió les posicions, però també oferint flexibilitat per a grans distàncies. Contenen mètodes per convertir-los a punts 2D, 3D, per importar des de rp3d::Vector3, punts 3D, i diverses funcions per a operacions, com per exmeple el producte escalar, producte vectorial, càlculs d'angles i magnituds, entre d'altres.

8.3.2 services

En el projecte hi ha diversos sistemes als que cal tenir accés des de molts punts del motor. Per tal de no contaminar l'espai de noms global amb variables estàtiques, hi ha certs objectes que posarem a disposició del motor a través de variables estàtiques dins d'un espai de noms anomenat "services". Services conté gestors de audio, gràfics, fonts i col·lisionadors, així com el registre d'entitats i el gestor de memòria de la física.

8.3.3 config

També tenim un espai de noms anomenat çonfig"on tenim diversos paràmetres configurables del motor. Alguns d'ells es poden configurar en temps d'execució, mentre d'altres són constexpr per necessitats de la implementació, o per facilitar optimitzacions al compilador. Els paràmetres ajustables són:

- El tamany del contenidor de chunks que té cada node (chunksContainerSize).

- Un paràmetre per activar o desactivar el renderitzat (render).
- El tamany de l'esfera al voltant de la càmera en la qual el chunkLoader carrega nous chunks (chunkloadSphereRadius).
- El nombre de blocs per costat de cada chunk (chunkSize).
- La profunditat màxima en que el renderitzat pot dibuixar una capa (cameraDepth).
- El tamany dels costats dels sprites (spriteSize).
- El zoom de la càmera (zoom).
- Dos paràmetres que permeten ajustar la profunditat de camp de la càmera (depthScale i minScale).
- L'alçada a la que la càmera es mantindrà sobre el jugador (cameraHeight).
- Un multiplicador de la velocitat de les òrbites (orbitDebugMultiplier).
- La distància en que una entitat pot interactuar amb un interactable (interactableRadius).
- La distància respecte la càmera amb la que les entitats no permanents seràn destruïdes (destroyDistance).
- El freqüència del motor de física (physicsHz).
- El nombre de passes del solucionador de col·lisions entre nodes i entitats (physicsSolverIterations).
- Un paràmetre per activar i desactivar el renderitzat d'ombres segons profunditat de la capa (drawDepthShadows).
- Un paràmetre per activar la extrapolació de posicions per al renderitzat (extrapolateRenderPositions).
- Un paràmetre per ajustar la profunditat en que s'activen les ombres (minShadow).
- Un paràmetre per activar i desactivar la gravetat (gravityEnabled).
- Un paràmetre per activar i desactivar la fricció (dragEnabled).
- Un paràmetre per activar i desactivar la oclusió ambiental (AOEnabled).
- La extensió dels sprites que fem servir (spriteExtension).
- La extensió del audio que fem servir (audioExtension).
- La extensió de les fonts que fem servir (fontExtension).
- El nombre de vegades que es pot fer Ctrl+Z al editor de prefabs (pfbEditorMaxCheckpoints).

8.3.4 Sprites

Els sprites són una classe que representa un sprite animat. Estàn associats a una sola textura, però contenen un vector de quadres que especifiquen diverses regions de la textura. Gestiona una referència al quadre actual, i avançant regularment de quadre dona l'efecte d'animacions.

8.3.5 Gestors de recursos

Les classes audioManager, fontManager, graphicsManager i colliderManager són classes que s'encarreguen de gestionar la creació, accés i destrucció de recursos que es fan servir en la aplicació. Ofereixen funcions per obtenir un recurs a partir del seu nom, i el gestor de forma transparent l'oferirà, carregant-lo de disc si cal. Són tots molt similars, amb la diferència que el de gràfics gestiona Textures i Sprites, i el de col·lisionadors té un funcionament lleugerament diferent degut que ha de gestionar els recursos amb rp3d::.

8.3.6 Observer

L'Observer és una classe que proporciona un sistema d'esdeveniments al motor. Permet definir events associats a un tipus de dades, que podran ser llençats des de qualsevol part del motor. Un cop enviat l'esdeveniment, les dades seràn entregades als subscriptors d'aquell tipus d'esdeveniment.

8.4 Llibreries

A continuació es detalla el funcionament de les llibreries que fem servir pel projecte.

8.4.1 HardwareInterface

HardwareInterface és la llibreria que hem creat al llarg del desenvolupament del projecte per separar el codi dependent de cada plataforma en una capa separada. Ofereix classes per representar textures, àudio, fonts i colors, a més de les estructures per representar punts en 2D i 3D. Ofereix una interfície per dibuixar píxels, rectangles, text i textures, per reproduïr audio, inicialitzar i tancar netament el sistema, i interactuar amb les tecles i joysticks entre d'altres funcions. Si bé les implementacions de Switch i PC estan ambdues fetes amb SDL2, fent que siguin prou similars, hi ha funcions que sí difereixen, com per exemple la d'obtenció de input, o la inicialització i neteja del sistema, que a consola requereix encendre diversos serveis i apagar-los manualment.

8.4.1.1 SDL2

SDL2 és fet servir dins de HardwareInterface per implementar la gestió de finestres, audio, gràfics i entrada. La interfície que ofereix la llibreria és C, així que molta de la interacció amb SDL2 es fà a través de punters, castings, i gestió manual de memòria.

8.4.2 EnTT

EnTT ens presenta dos tipus de dades principals, el registre i les entitats. Les entitats són simplement un identificador per a agrupacions de components del registre. El registre, en canvi, és una classe més completa que farà tota la gestió dels components i entitats del sistema. Ens permet crear i destruïr entitats, afegir i treure components a les entitats, obtenir "vistes"del registre segons filtres de cerca, i manipular els components de les "vistes"obtingudes.

8.4.3 ReactPhysics3D

ReactPhysics3D ens ofereix totes les utilitats relacionades amb els càlculs de física del motor. Principalment ens permet crear cossos, que són agrupacions de simples polígons, els quals podràn col·lionar amb altres cossos. Les col·lisions es comproven manualment entre cossos, i en cas que hi hagi un contacte ens serà informat, obtinguent dades sobre la normal del contacte,

la profunditat i els cossos involucrats. Tot i ser una llibreria de C++, la gestió de memòria és manual, i requereix construïr i destruïr manualment els objectes de la llibreria.

8.4.4 nlohmann/json

Nlohmann/json ens ofereix una senzilla interfície per crear i parsejar arxius JSON. Cada classe a serialitzar ha de implementar les funcions to_json() i from_json, que s'encarregaràn de serialitzar i deserialitzar des d'arxius. Totes les classes de la simulació excepte el terrainChunk i el prefab són serialitzables a JSON, i aquestes dues que no ho són implementen la serialització directament.

8.4.5 FastNoise

FastNoise ens ofereix la classe FastNoise, que és un generador de soroll generalitzat amb el que podem cridar les diferents funcions de soroll. Ens permet canviar freqüència, tipus de soroll, els paràmetres del fractal i la llavor del soroll entre d'altres.

8.4.6 IceCream-Cpp

IceCream-Cpp presenta la macro IC(), que facilita el "print debugging", admetent múltiples paràmetres de tipus variats, i presentant-los per consola de forma entenedora.

Capítol 9

Implementació i proves

A continuació s'expliquen diverses implementacions interessants, o bugs curiosos i la seva subseqüent solució.

9.1 Observer

La implementació del observador és una de les parts que es va fer més tard, i és una implementació on es demostra part de la potència de C++, i en concret, C++20 / Modern C++.

```
using eventArgs = std::variant<
   std::tuple<entt::entity, entt::entity, double>,
   std::tuple<universeNode*, entt::entity, double>,
   std::tuple<universeNode*, universeNode*, double>,
   entt::entity
>;
```

Primer de tot aquí creem un tipus de dades anomenat eventArgs, el qual representarà els arguments que un esdeveniment pot enviar als subscriptors. Std::variant a vegades anomenat "sum type"podriem descriure'l com l'equivalent a un *union* amb *type safety*. Així doncs, una variable del tipus eventArgs podrà contenir una std::tuple d'algun dels tipus especificats, o un entt::entity.

```
enum class eventType{
   COLLISION_EE,
   COLLISION_NE,
   COLLISION_NN,
   PROJECTILEHIT,
   PROJECTILEBOUNCE,
   _SIZE,
};
```

Tot seguit definim el enum class eventType, que especificara els diversos tipus d'esdeveniments que el nostre observador pot gestionar.

```
namespace {
   consteval std::array<size_t,(unsigned)eventType::_SIZE>
    _observertypetable(std::array<std::pair<eventType,eventArgs>,
    (unsigned)eventType::_SIZE> args){
        std::array<size_t,(unsigned)eventType::_SIZE> result;
      for(auto& val : args){
         result[(unsigned)val.first] = val.second.index();
```

```
8       }
9       return result;
10     }
11 }
```

A continuació tenim una funció *consteval* dins un espai de noms anònim, que evita que la funció sigui accessible fora d'aquest arxiu. L'especificador *consteval* avisa al compilador que la funció no té efectes secundaris, i pot ser avaluada en temps de compilació en cas de ser necessari. La funció en sí retorna un array amb tamany igual al nombre de tipus d'esdeveniments diferents que tenim a eventType, i a cada posició conté una variable size_t. Pren per argument un array que conté parelles de eventType i eventArgs, els quals relacionarà per retornar l'array resultant. Per crear aquest resultat primer de tot l'inicialitza. Després itera per cada parella de eventType i eventArgs. Al array resultant coloca a la posició indexada pel valor del eventType un valor de tipus size_t, aquest valor l'obté de cridar index() al variant, que serveix com a una pseudo-reflexió i retorna l'índex del tipus dins del variant. Un cop ha iterat per totes les parelles retorna l'array que ha construït.

```
1  constexpr std::array<size_t,(unsigned)eventType::_SIZE> _eventLUT = _observertypetable(
2    std::array<std::pair<eventType,eventArgs>,(unsigned)eventType::_SIZE>{
3     std::make_pair(eventType::COLLISION_EE,std::tuple<entt::entity,entt::entity, double>()),
4     std::make_pair(eventType::COLLISION_NE,std::tuple<universeNode*,entt::entity, double>()),
5     std::make_pair(eventType::COLLISION_NN,std::tuple<universeNode*,universeNode*, double>()),
6     std::make_pair(eventType::PROJECTILEHIT,std::tuple<entt::entity,entt::entity, double>()),
7     std::make_pair(eventType::PROJECTILEBOUNCE,entt::entity())
8  });
```

Tot seguit creem una LUT que consisteix en l'array obtingut al cridar la anterior funció. La funció la cridem amb un std::array que creem *in-place* que conté parelles de eventType i eventArgs.

```
1  class observer {
2  public:
3    static void registerSubscriber(eventType t, std::function<void(eventArgs args)> f,void* owner){
4      if(!_subscribers[(unsigned)t].contains(owner))
5          _subscribers[(unsigned)t].emplace(owner,f);
6    }
7    static void deleteSubscriber(eventType t, void* owner){
8      if(_subscribers[(unsigned)t].contains(owner))
9        _subscribers[(unsigned)t].erase(owner);
10   }
11
12
13    template <eventType E,typename T>
14    static void sendEvent(T args)
15    {
16       static_assert(eventArgs(T()).index()==_eventLUT[(unsigned)E]);
17       for(auto& sub : _subscribers[(unsigned)E]){
18          sub.second(eventArgs(args));
19       }
20    }
21
22 private:
23    static std::array<std::unordered_map<void*,std::function<void(eventArgs args)>>,
24   (unsigned)eventType::_SIZE> _subscribers;
25 };
```

Finalment tenim el observer en sí. Les funcions registerSubscriber i deleteSubscriber simplement afegeixen un subscriptor al observador o l'esborren. Tenim un std::array que per cada tipus de esdeveniment conté un std::map que relaciona subscriptors amb funcions a cridar. El void* és un handle que l'objecte que s'ha subscrit ha donat al subscriure's, podria interpretar-se com un "reverse handle".
La funció interessant és sendEvent. És una funció genèrica depenent del tipus T i el tipus d'esdeveniment E. Primer de tot es fa una asserció estàtica utilitzant la *lookup table* que abans hem emplenat. Aquesta asserció comprova en temps de compilació que l'esdeveniment que s'envia s'adigui a les dades que s'envien junt amb ell. En temps d'execució el que es farà serà iterar per tots els subscriptors d'aquell tipus d'esdeveniment, i es cridarà la funció associada a cada subscriptor.
D'aquesta manera tenim una implementació d'un observador amb poc codi innecessari, senzilla i amb comprovació de tipus en temps de compilació.

9.2 universeNode::getLocalPos

La funció universeNode::getLocalPos serveix per obtenir la posició local a un node específic, donada una posició qualsevol i el seu node pare.

```
1  fdd universeNode::getLocalPos(fdd f, universeNode* u) const
2  {
3     assert(!std::isnan(_position.x));
4     if (u == this)
5        return f;
6     else
7     {
8        fdd transform = f;
9        const universeNode* transformLocal = this;
10
11       while (transformLocal != u) { // while transformLocal isn't f's parent (u)
12          if (u && u->_depth > transformLocal->_depth) {//should move u
13             transform += u->_position;
14             u = u->_parent;
15          }
16          else {// move transformLocal
17             transform -= transformLocal->_position;
18             transformLocal = transformLocal->_parent;
19          }
20       }
21       return transform;
22    }
23 }
```

Primer de tot creem una fdd anomenada transform, que representarà la transformació que anem acumulant al atravessar l'arbre de nodes, i també crearem una punter al pare actual de la transformació que estem construïnt. Per tal d'evitar la pèrdua de precisió fent transformacions innecessaries, el que farem serà trobar el camí més curt atravessant l'arbre cap amunt des de orígen i destí. Un cop els dos pares coincideixin haurem arriat al objectiu. Sempre mourem primer cap amunt la posició que estigui relativa a un node més profund, i en cas d'empat mourem l'origen. A l'hora de moure una de les posicions el que farem serà restar o sumar la posició del node actual a la transformació, depenent de si estem pujant l'origen o el destí. També actualitzarem la variable transformLocal o 'u' cap al nou pare, depenent de quin haguem mogut. Quan les transformLocal i 'u' siguin el mateix node sabrem que hem aconseguit arribar al objectiu, i podrem retornar la transformació que hem anat construïnt com a resultat.

9.3 terrainPainterGenerator::getChunk(const point3Di p)

```
terrainChunk terrainPainterGenerator::getChunk(const point3Di& p)
{
   if (p.z < 0 || fdd{ 0,0,0,0 }.distance2D(fdd{ (double)(p.x) * config::chunkSize,
    (double)(p.y) * config::chunkSize,0,0 }) > (_diameter / 2)*config::chunkSize)
   {
      return terrainChunk();
   }

   terrainChunk chunk(p);

   for (int x = 0; x < config::chunkSize; ++x) {
      for (int y = 0; y < config::chunkSize; ++y) {
         if (fdd{ 0,0,0,0 }.distance2D(fdd{ (double)(p.x * config::chunkSize) + x,
        (double)(p.y * config::chunkSize) + y,0,0 }) <= _diameter / 2)
         {
            double noise = getNoise({ (p.x * config::chunkSize) + x,
          (p.y * config::chunkSize) + y });
            for (int z = 0; z < config::chunkSize; ++z) {
               unsigned int currentHeight = p.z * config::chunkSize + z;
               chunk.setBlock({&_terrainPainter.getBlock(currentHeight, noise),
            (blockRotation)(rand()%4)}, point3Di{ x,y,z });
            }
         }
      }
   }
   if(_liquidLevel>0)
      fillLiquid(chunk, p, _liquidLevel);
   chunk.setLoaded();
   chunk.clearDirtyFlag();
   return chunk;
}
```

Aquesta és la implementació de la funció que genera els chunks de terreny amb terrainPainterGenerator. El seu funcionament és el següent:

1. Es comprova que el chunk a generar sigui vàlid, i si es troba fora dels límits del terreny es retorna un chunk buit.

2. En cas contrari, creem el chunk buit amb la posició que li correspon.

3. Tot seguit iterem per les columnes de blocs en els eixos X i Y.

4. Comprovem que el bloc a assignar estigui dins dels límits del generador, i en cas contrari ometem la generació d'aquesta columna de blocs.

5. Obtenim el nivell de soroll a partir del generador de gradient de soroll en les coordenades X i Y actuals.

6. Iterem pels blocs de la columna de blocs, i els assignem segons el bloc que el terrainPainter ens indiqui per aquella alçada i nivell de soroll.

7. Un cop assignats els blocs, emplenem de líquid els blocs d'aire en cas que aquells blocs es trobin sota el nivell del mar.

8. Finalment marquem el chunk com a carregat i net i el retornem.

9.4 Bugs

Al llarg del desenvolupament del projecte s'han arreglat gran quantitat de *bugs*, a continuació es detallen alguns dels més destacables.

9.4.1 Pèrdua de precisió al calcular òrbites

Degut a la gran escala del univers, el càlcul d'òrbites d'alguns cossos es feia de forma incorrecte a causa de falta de precisió en els nombres de coma flotant (IEEE 754-2008). El problema originava de la falta de precisió al calcular el moviment en cossos llunyans al seu node pare, en els que l'increment de posició en 33ms era massa petit, i al afegir el desplaçament a la posició la òrbita es desplaçava de manera desproporcionada. El *bug* es va arreglar afegint un acumulador de posició, al qual s'afegeix el desplaçament del cos a cada cicle del motor de física. Un cop el desplaçament és prou significatiu respecte la posició, aquest s'hi afegeix i es neteja per tornar a començar el cicle.

9.4.2 Problemes de precisió al comprovar col·lisions entre nodes i entitats

Als inicis del solucionador de física es van detectar col·lisions incorrectes entre el jugador i el terreny. Va resultar ser un error que només es produïa al estar en posicions llunyanes respecte al pare. Aquests símptomes apuntaven a un altre error de precisió, i efectivament, va resultar ser causat per utilitzar coordenades del node en la resolució de col·lisions. La comprovació del solapament entre chunks i entitats es feia amb coordenades respecte el node, i això causava mal comportament al tenir posicions prou grans. Per arreglar-ho es va canviar la resolució d'aquestes col·lisions a fer-se a partir de coordenades respecte el chunk amb que comprovar, i d'aquesta manera va ser resolt el *bug*.

9.4.3 Renderitzat incorrecte en alçades negatives

Aquest error va ser detectat sorprenentment tard en el desenvolupament del motor, tot i ser-hi des de gairebé els inicis. Les capes de terreny ubicades en una coordenada Z negativa eren dibuixades a una profunditat inferior a la adeqüada. La culpable va resultar ser la funció fmod de C++, que al rebre un nombre negatiu el resultat passava a també ser-ho. Tinguent sospites d'aquesta funció es va fer un petit plot amb un full de càlcul, i es va arreglar fàcilment despres de confirmar que aquesta era la causa.

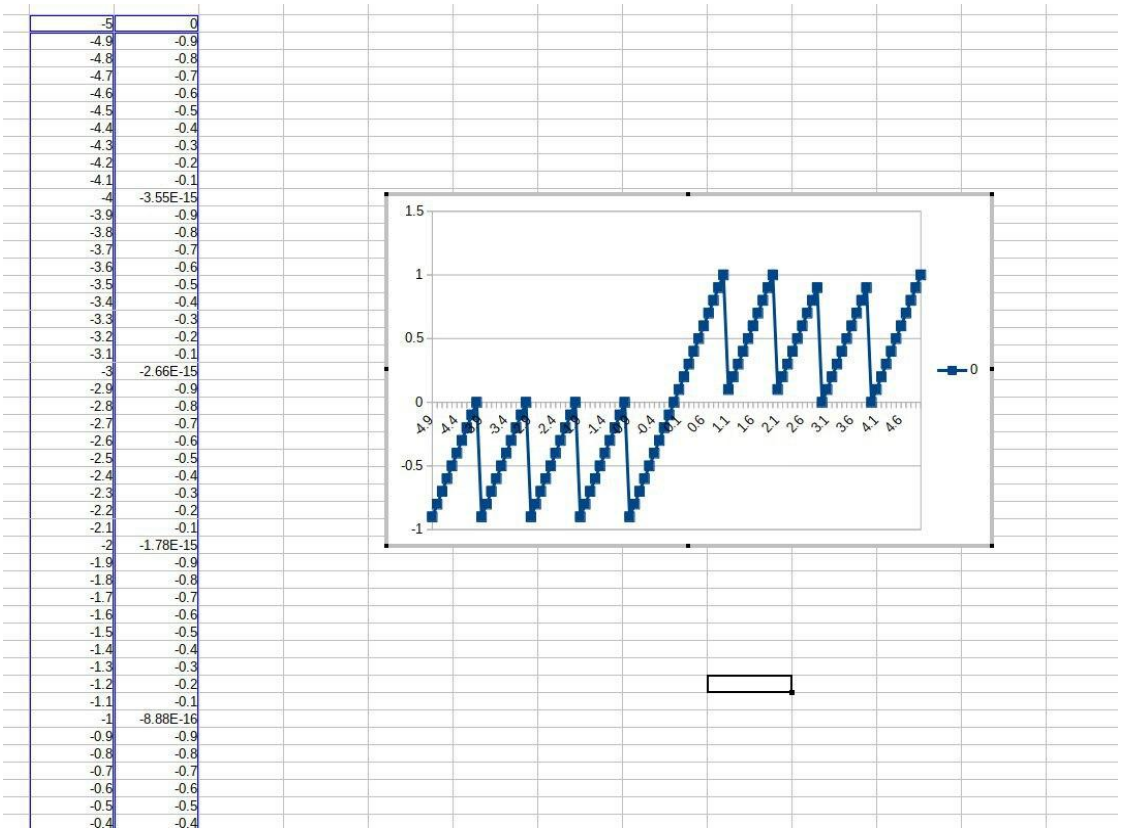


Figura 9.1: Plot fet amb un full de càlcul per comprovar el comportament de la funció fmod amb nombres negatius i positius. Al eix vertical es representa el resultat, i al horitzontal els valors d'entrada.

9.5 Proves

Degut a la naturalesa del projecte és prou complicat dissenyar tests unitaris. El correcte funcionament del motor s'ha comprovat a través del feedback visual de la demo. Alguna de les funcionalitats testejables s'ha comprovat a través dels seus resultats, com és el cas de la simulació d'òrbites. L'escenari del joc conté una simulació del Sistema Solar a escala real, i per tant, de ser correcte la simulació les òrbites haurien de ser estables i similars a la realitat.

En la figura 9.2 podem observar la representació de les òrbites de Mercuri, Venus, Terra, Mart i Júpiter, i s'observen les òrbites estables i amb cap tipus d'eccentricitat. Per la simulació s'han utilitzat les distàncies i velocitats mitjanes dels cossos, així que la eccentricitat real de les òrbites no queda reflectida en la simulació.

Figura 9.2: Captura del mapa estelar.

Capítol 10

Resultats

S'han assolit tots els requisits i tasques establertes pel projecte. El motor resultant actualment té el nom provisional de Z5, en referència a la saga de videojocs 4X espacials anomenats X, publicada pel desenvolupador Egosoft. El motor està publicat sota llicència GPL3 a Github. A més de les plataformes inicials, també es pot compilar per a Windows utilitzant Visual Studio, i també per web utilitzant emscripten. Els binaris estàn disponibles sota la secció de Releases al repositori de Github, i una build experimental de la versió emscripten està disponible a http://z5.ledgedash.com/, però el suport dels navegadors és limitat i pot no funcionar.

10.1 Captures de pantalla

Figura 10.1: Fotografía de la demo executant-se en una Nintendo Switch.

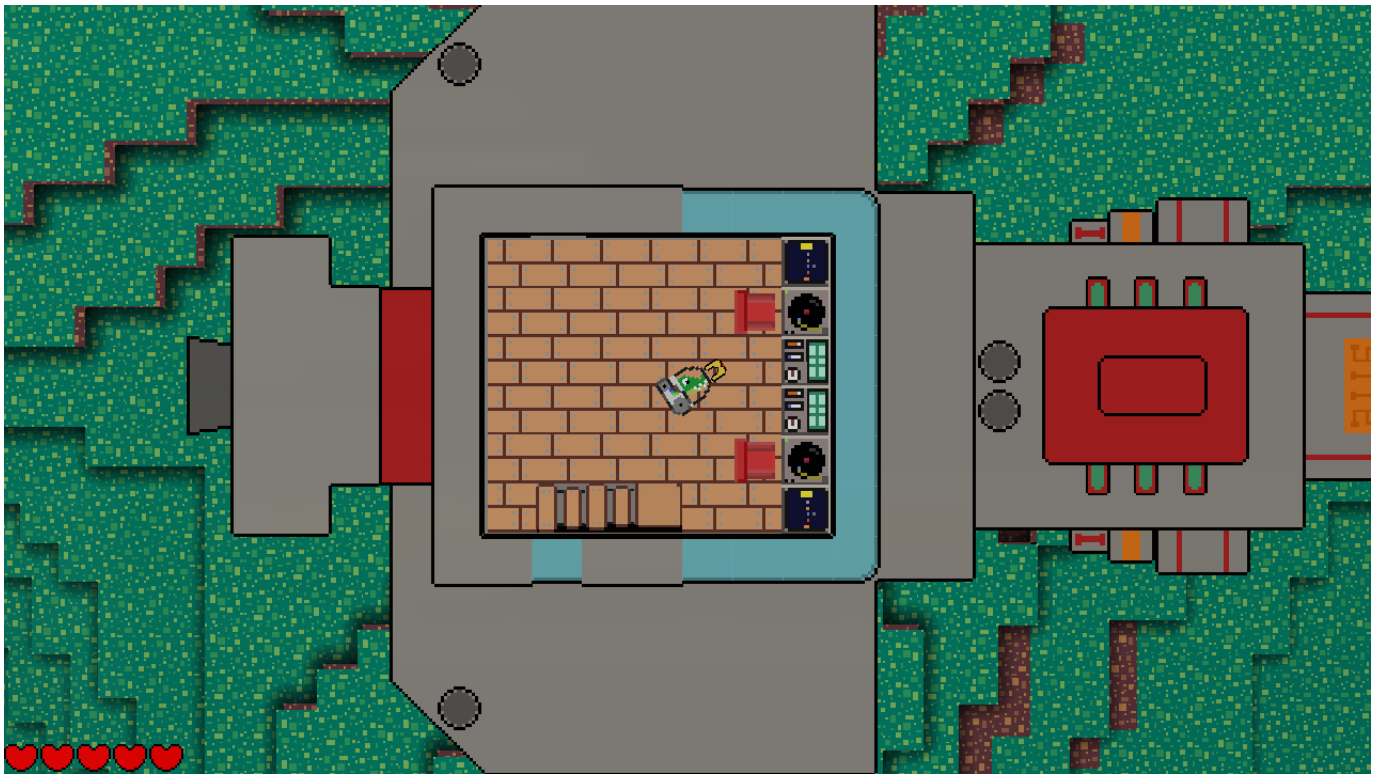


Figura 10.2: Captura de la demo amb el jugador dins la nau Roc.

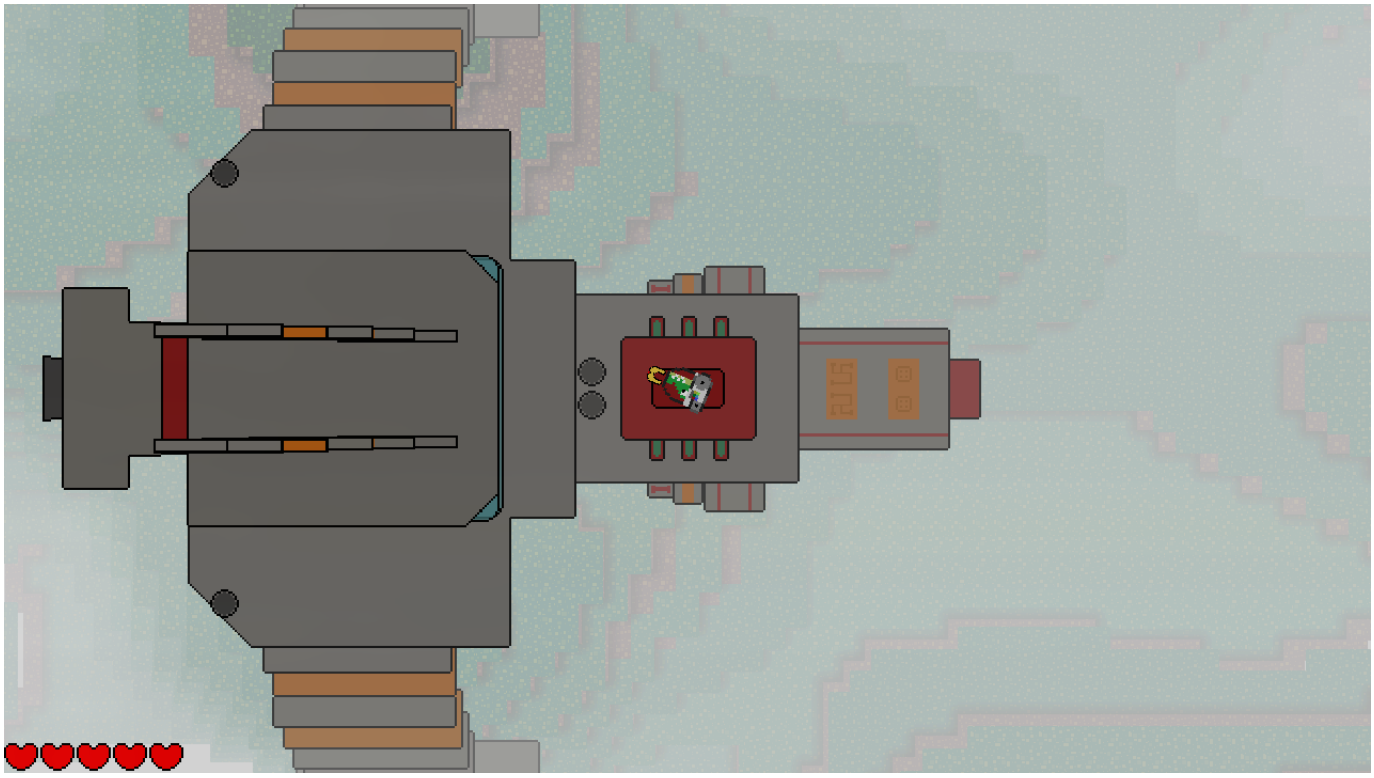


Figura 10.3: Captura de la demo amb el jugador fora la nau.

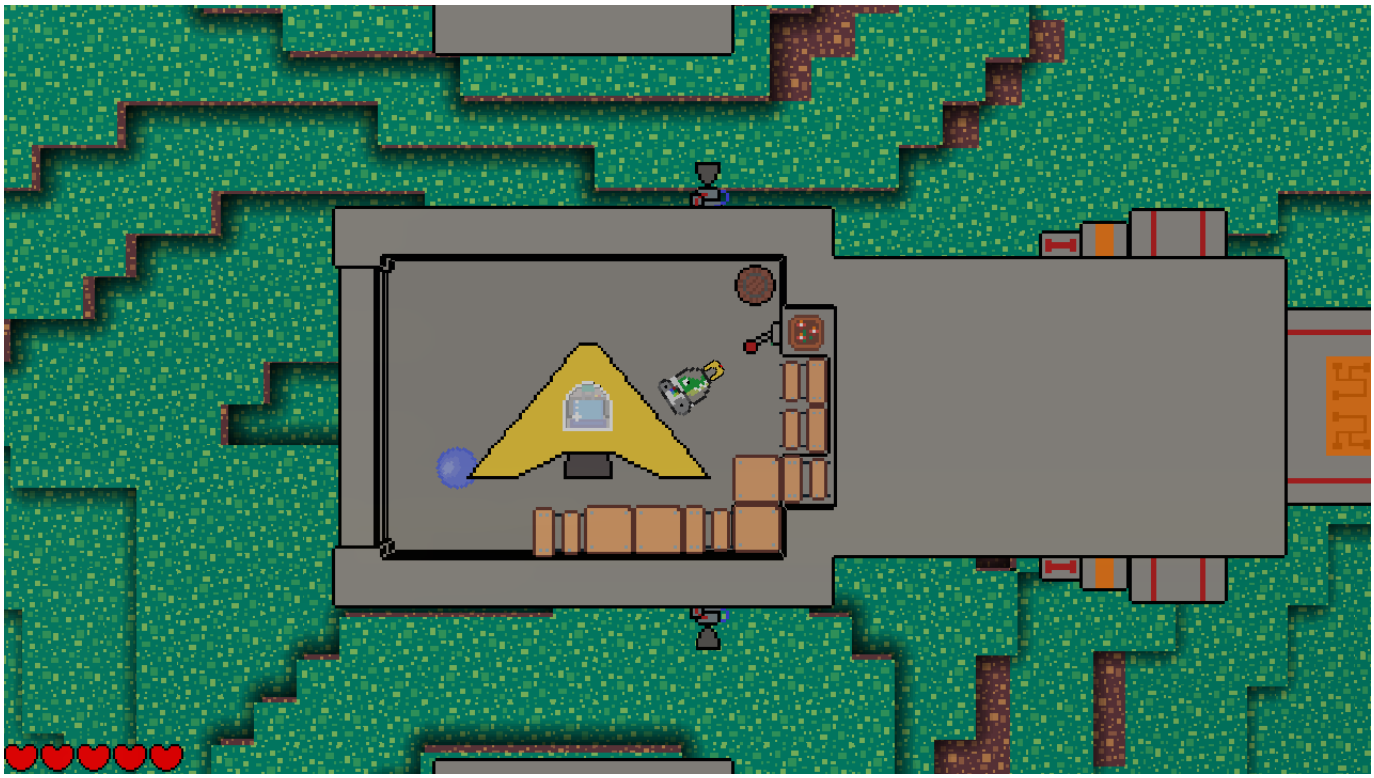


Figura 10.4: Captura de la demo amb el jugador fins l'hangar de la nau.

Figura 10.5: Captura de la demo amb el jugador dins una habitació.

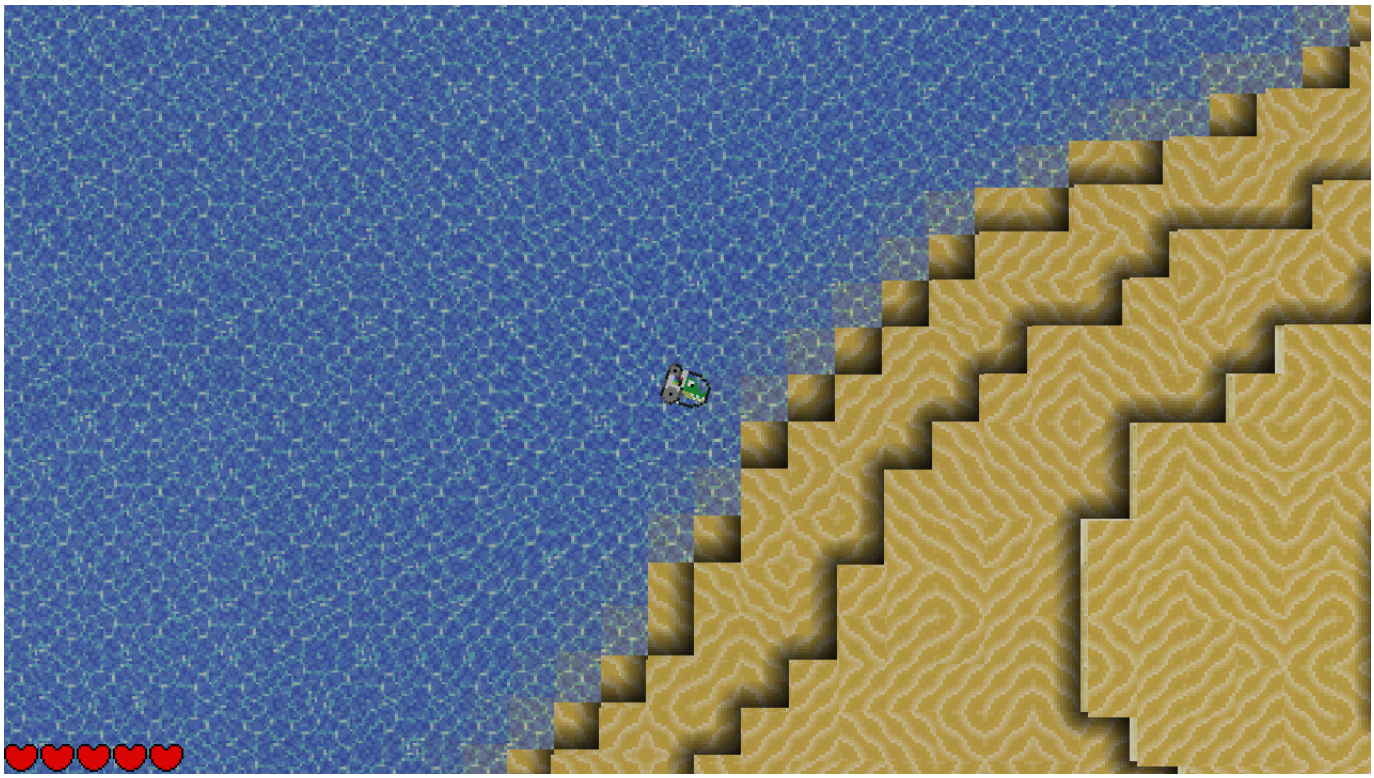


Figura 10.6: Captura de la demo amb el jugador nedant a la costa.

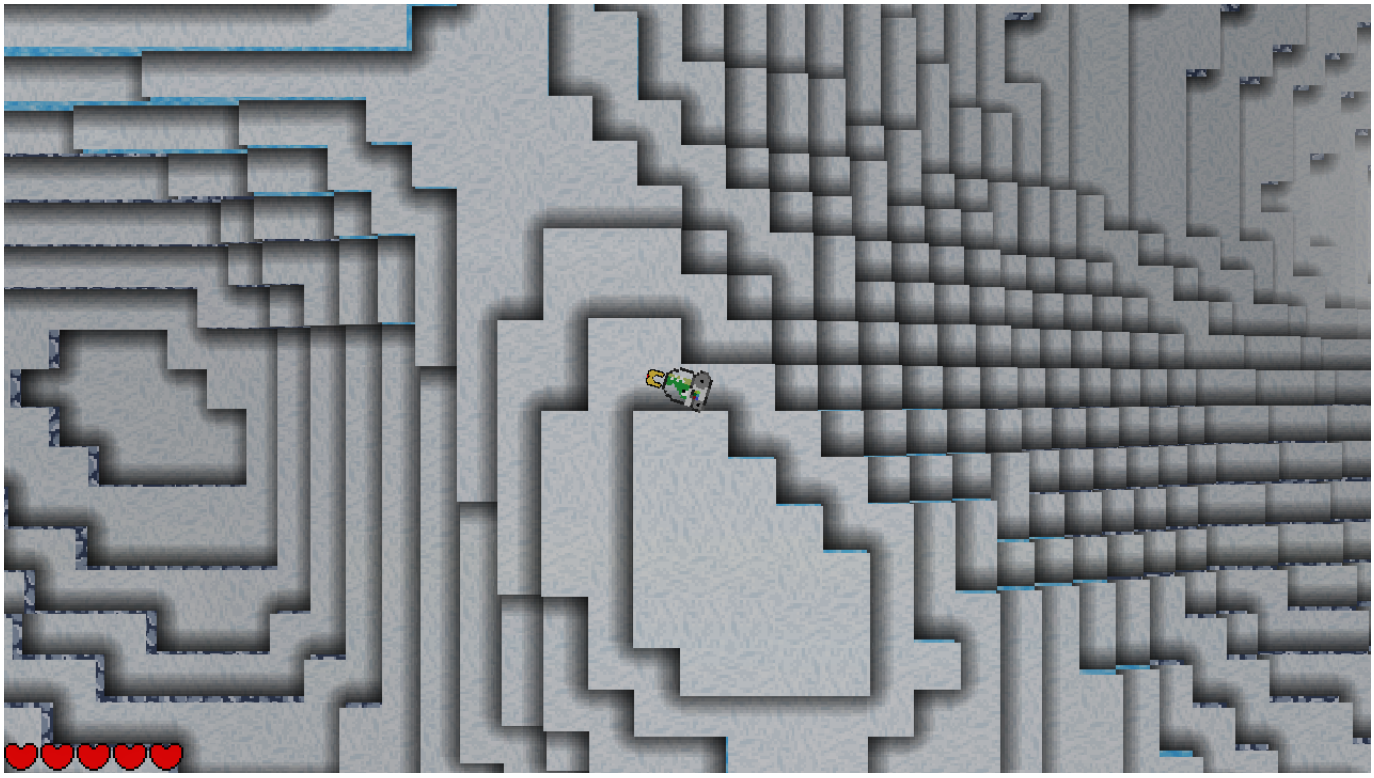


Figura 10.7: Captura de la demo amb el jugador en una regió muntanyosa nevada.

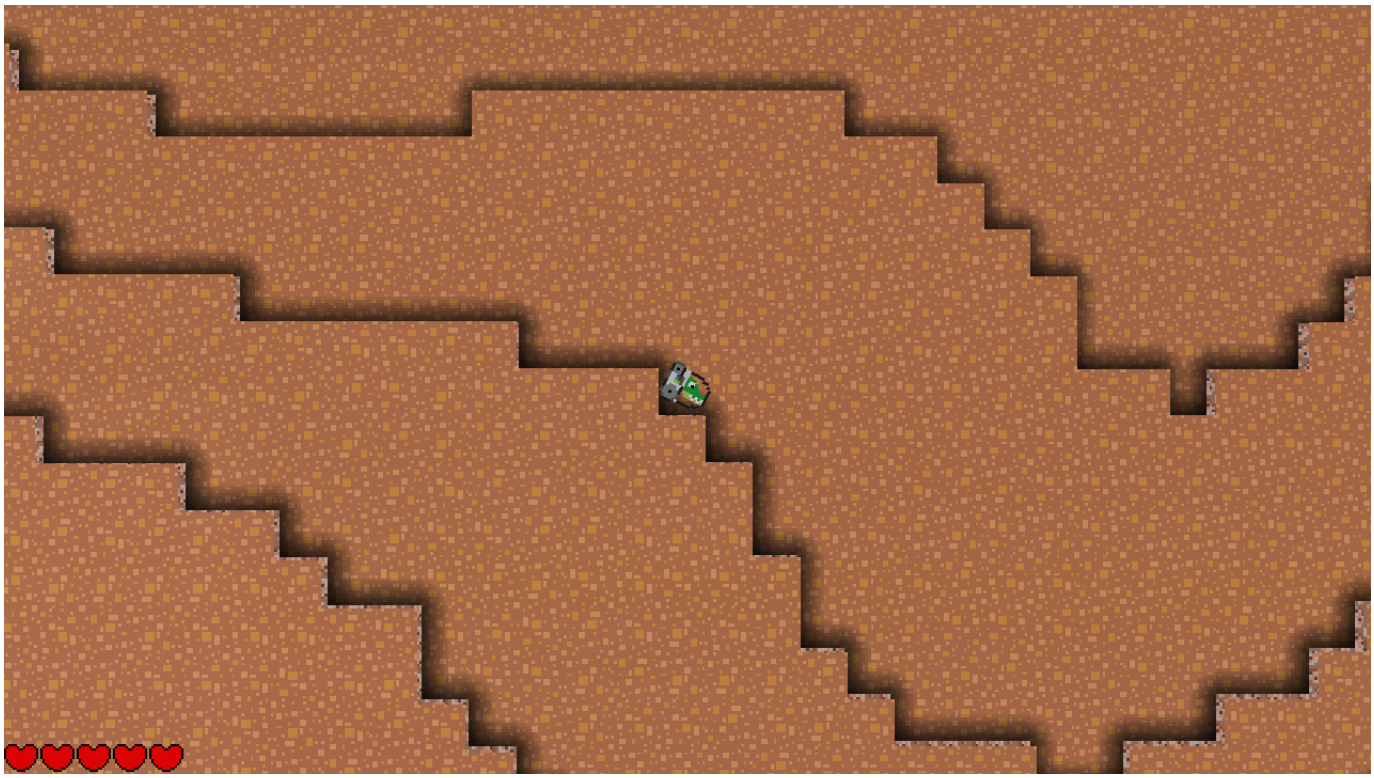


Figura 10.8: Captura de la demo amb el jugador a la superfície de Mart.

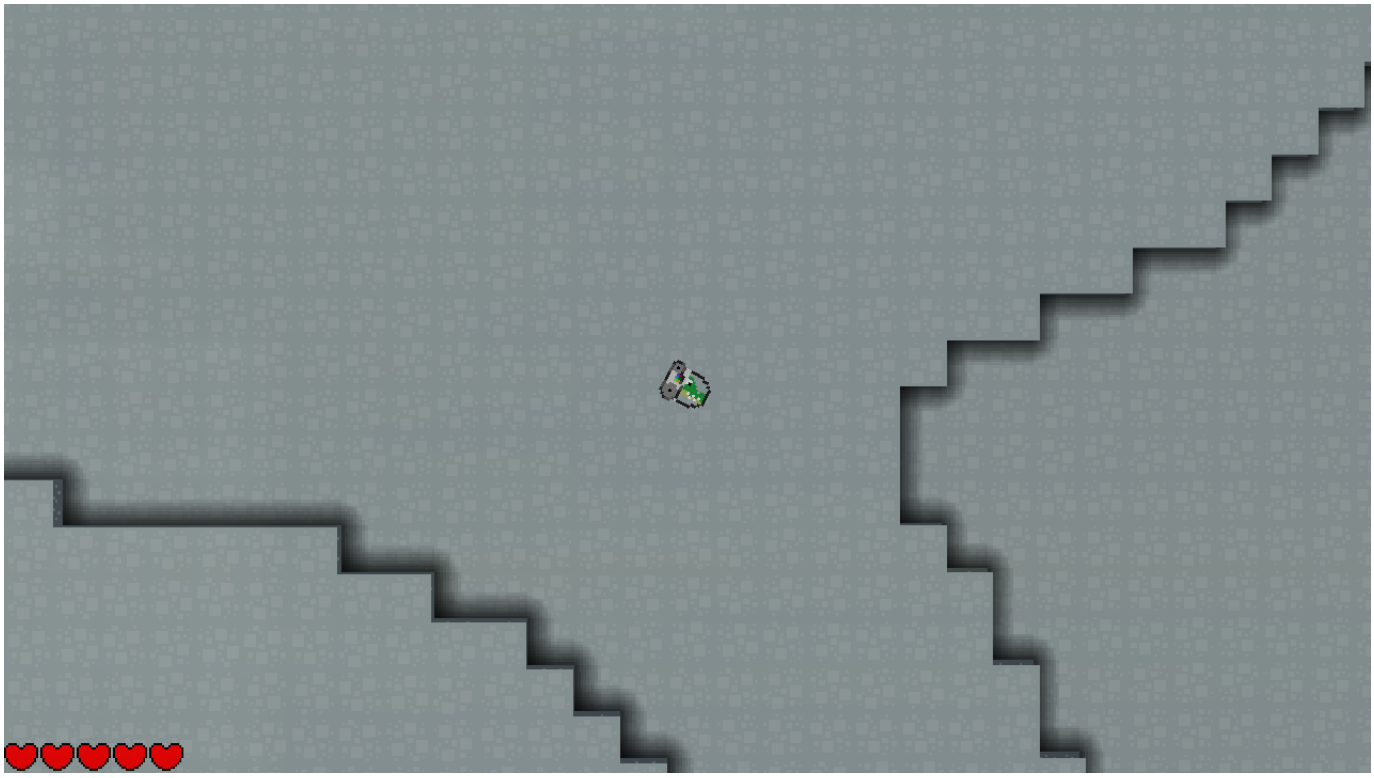


Figura 10.9: Captura de la demo amb el jugador a la superfície de la Lluna.

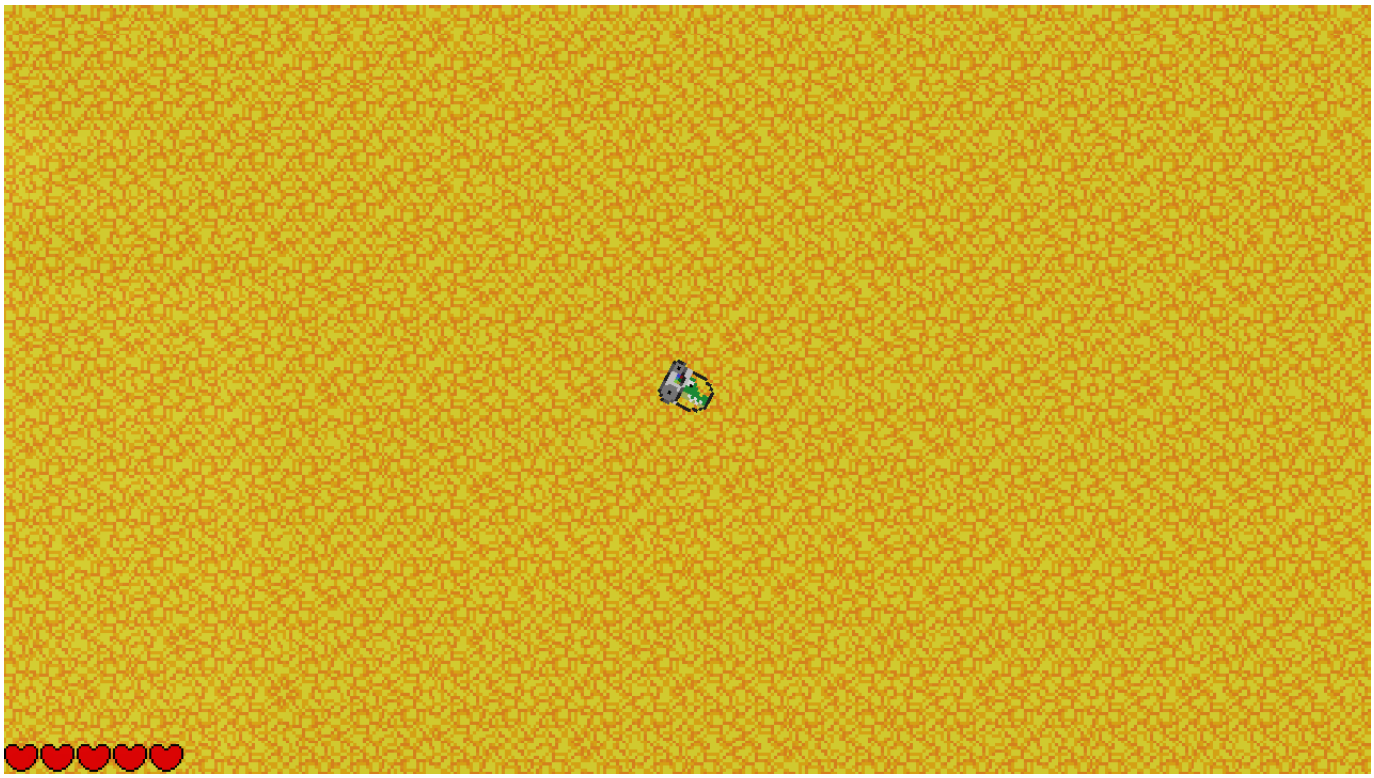


Figura 10.10: Captura de la demo amb el jugador dins del Sol.

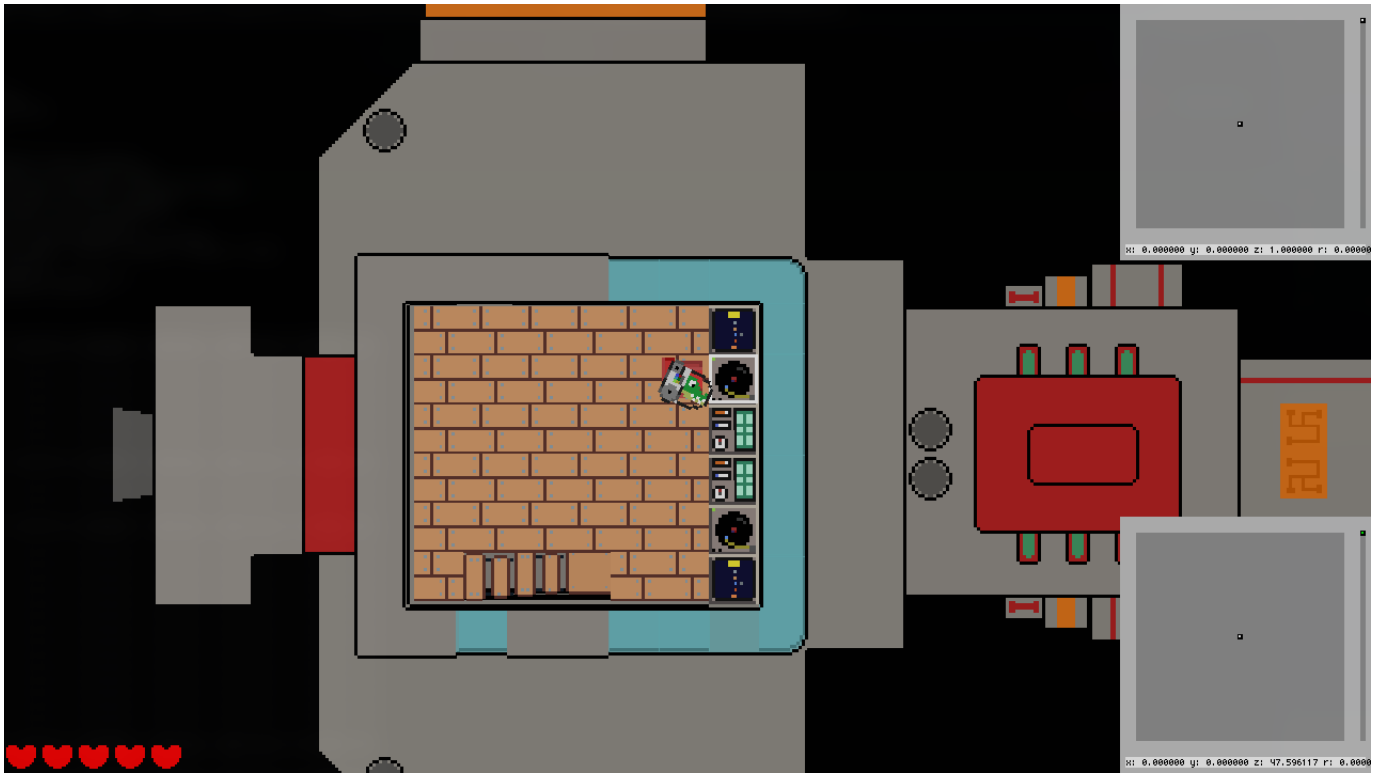


Figura 10.11: Captura de la demo amb el jugador pilotant la nau per l'espai.

Figura 10.12: Captura de la demo amb el mapa estelar desplegat, mostrant l'òrbita de la Terra al voltant del Sol.

Figura 10.13: Captura del menú de creació de partida nova.

Figura 10.14: Captura del menú de selecció de partida.

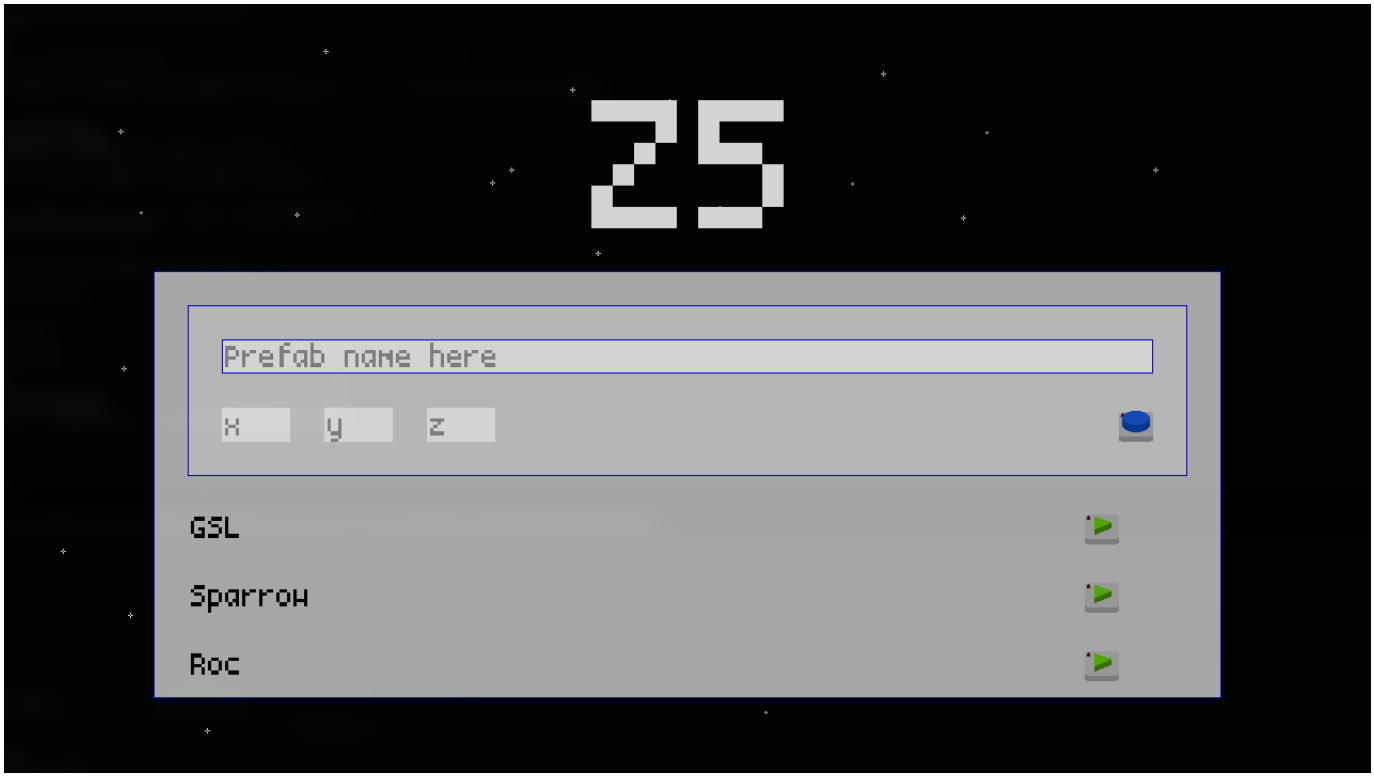


Figura 10.15: Captura del menú d'edicio de prefabs.

Capítol 11

Conclusions

Els objectius del projecte han estat assolits, i tot i no competir al mateix nivell que els motors principals, compleix unes funcionalitats que actualment no estan cobertes per cap altre motor, i pot resultar molt útil en cas de voler desenvolupar el tipus de jocs pels que ha estat dissenyat. El rendiment a Linux és bastant acceptable, però sobretot per a Switch queden optimitzacions per fer per tal que el motor sigui comercialment viable. Tot i haver assolit els objectius inicials, espero continuar treballant en aquest projecte al llarg del temps, i acabar publicant un joc fet a partir d'una versió millorada d'aquest motor.

Al llarg del projecte s'han posat a prova els coneixements adquirits al llarg del grau. S'han aplicat idees apreses des d'Enginyeria del Software fins a Arquitectura de Computadors. He aprofundit molt les habilitats de programació, i he après en major profunditat les eines que ofereix C++. També he après a estructurar i organitzar la feina en un projecte de tamany prou considerable. He fet gran ús de Git, amb més de 300 commits multiples branques. El projecte també m'ha ajudat a aprendre a escollir i utilitzar llibreries amb major facilitat i agilitat. Crec que tot el projecte m'ha aportat coneixement molt valuós, i estic content i orgullós d'haver-lo dut a terme.

Capítol 12

Treball Futur

Com s'ha esmentat a la conclusió, el motor encara té camí per recórrer si es vol desenvolupar un videojoc comercial fent-lo servir.
A continuació es fa un recull de punts a desenvolupar en el futur:

12.1 Refactorització i neteja de codi

Actualment creiem que el motor està ben dissenyat, però hi ha una certa quantitat de deute tècnic acumulat que ha de ser adreçat si es vol continuar el desenvolupament del projecte. Alguns dels punts a netejar són els següents:

- Separar algunes de les responsabilitats de universeNode en altres classes més petites.
- Separar la renderització del State_Playing en una classe separada.
- Reimplementar els blocs interactuables com a entitats lligades a un registre específic a cada node.
- Desacoblar algunes de les classes, fent servir el Observer com a mitjà de comunicació.

12.2 Optimitzacions

Tot i que el rendiment és normalment molt jugable, en certes ocasions la taxa de quadres per segon pot baixar a un sol dígit. Optimitzacions en el càlcul de col · lisions podria ajudar molt a evitar aquestes caigudes, ja que sembla ser la part que més càrrega dóna a la CPU.

12.3 Renderització

Per aconseguir un gran salt de rendiment i evitar molts dels maldecaps que la renderització en 2D d'un món 3D, passar el procés de renderitzat a 3D fent servir la GPU donaria molta més flexibilitat al motor, junt amb un gran salt de rendiment al renderitzar. Tot i passar a 3D, la idea seria seguir amb la mateixa càmera en vista d'ocell per mantenir el mateix estil visual.

12.4 Rotacions

Tot i que les entitats poden rotar sobre sí mateixes, actualment els nodes no tenen aquesta habilitat. Encara que aquesta limitació dóna lloc a opcions de gameplay bastant interessants, podria ser una gran opció almenys explorar la possibilitat d'afegir rotacions al voltant del eix

Z als nodes. En cas de haver canviat a renderització 3D també, podríem inclús implementar rotacions al voltant de tots els eixos, oferint sis graus de llibertat tant a nodes com a entitats. Aquest canvi però requeriria d'una gran reestructuració del motor, i no entra en els plans immediats de futur.

Bibliografia

[1] Orbital mechanics. URL https://en.wikipedia.org/wiki/Orbital_mechanics.

[2] Runge-kutta methods. URL https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods.

[3] Michele Caini. Documentació de entt, 2020. URL https://github.com/skypjack/entt/wiki.

[4] Daniel Chappuis et al. Documentació de reactphysics3d, 2020. URL https://www.reactphysics3d.com/usermanual.html.

[5] Niels Lohmann et al. Documentació de nlohmann/json, 2020. URL https://nlohmann.github.io/json/.

[6] Stefan Gustavson. Simplex noise demystified. 01 2005.

[7] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014.

[8] Amit Patel. Making maps with noise functions, jul 2015. URL https://www.redblobgames.com/maps/terrain-from-noise/.

Capítol 13

Manual d'usuari

13.1 Instalació de la demo a PC

Per la instalació de la demo a PC cal seguir les següents passes:

1. Descarregar els binaris adeqüats per la nostre plataforma a través de la secció Releases de Github.
2. A Linux, cal també instal·lar SDL2 des del nostre gestor de paquets. A windows les llibreries necessàries estàn distribuïdes junt amb els binaris.
3. Descomprimir el zip a una carpeta qualsevol.
4. Executar el binari.

13.2 Instalació de la demo a Nintendo Switch

Assumint que la Nintendo Switch ja disposa de *homebrew launcher*, només cal descomprimir la carpeta Z5 del zip distribuït a través de Github a dins la carpeta /switch de la tarjeta microSD. Un cop fet això podrem executar la demo accedint al *homebrew launcher* i prement sobre la icona del Z5.

13.3 Instruccions per la demo

Des del menú principal tenim accés a les partides guardades, la opció de crear una nova partida i el menú del editor de prefabs. Dins el *gameplay* de la demo es poden realitzar les següents accions:

- Podem controlar el personatge amb les tecles WASD.
- Podem saltar si estem en contacte amb el terra utilitzant la tecla d'espai.
- Podem interactuar amb objectes ressaltats utilitzant la tecla intro.
- En cas d'estar interactuant amb els controls d'una nau, podem dirigir-la amb WASD, RF i X. També podem sortir dels controls utilitzant la tecla retrocés.
- En cas d'estar al aire podem maniobrar lleugerament utilitzant WASD, i podem controlar la nostra altitud amb R i F.

- Si el mode Debug està activat, podem obrir una consola de *debugging* amb la tecla de sobre el tabulador (º), i es pot interactuar com amb una consola normal. La comanda 'help' llistarà totes les comandes disponibles.
- Podem obrir i tancar el mapa estalar amb la tecla M.
- Podem canviar l'objecte equipat fent servir la rodeta del ratolí.
- Amb el clic esquerre del ratolí accionarem l'objecte que tinguem a la mà. Això ens permetrà excavar el terreny amb l'eina per defecte.

13.4 Instruccions d'ús del editor de prefabs

Un cop dins l'editor podem prémer la tecla H per veure les accións que fa cada tecla. Els controls bàsics són:

- WASDRF per moure la càmera.
- Clic esquerre per colocar el bloc seleccionat a sota la posició del cursor.
- Clic dret per esborrar el bloc de sota el cursor.
- Rodeta del ratolí o fletxes esquerra i dreta per seleccionar un altre bloc de la barra.
- Fletxes amunt i avall per canviar el bloc de la posició actual de la barra.
- Esc per sortir i guardar, Shift+Esc per sortir sense guardar els canvis.

13.5 Instal·lació del motor

Per crear un videojoc fent servir el nostre motor seguirem les següents passes. S'assumeix que es fa el desenvolupament en una màquina Linux.

1. Instal·lar les llibreries SDL2 a través del nostre gestor de paquets.
2. Clonar recursivament el repositori fent servir la comanda 'git clone –recursive https://github.com/thel
3. Podem obrir el projecte amb Qt Creator seleccionant la carpeta del repositori des de la interfície. També podem treballar manualment amb un editor de text.
4. Un cop fetes les modificacions necessàries al motor i joc, podem compilar el projecte amb la comanda 'make -f linux.mk'
5. En cas de voler compilar per Nintendo Switch haurem d'instal·lar les llibreries de desenvolupament de *homebrew*. Per compilar per a Switch farem servir la comanda 'make -f switch.mk'