Universitat de Girona
Escola Politècnica Superior

Treball final de grau

Estudi: Grau en Disseny i Desenvolupament de Videojocs

Títol: Desenvolupament d'un videojoc de supervivència

Document: **Memòria**

Alumne: **Jaume Ramos Moreso**

Tutor: **Gustavo Patow**
Departament: **Informàtica, Matemàtica Aplicada i Estadística**
Àrea: **Llenguatges i Sistemes Informàtics**

Convocatòria (mes/any) Juny/2021

Índex de continguts

Índex de figures

1. Introducció i objectius

1.1. Introducció

Últimament el gènere dels videojocs de supervivència està passant per un molt bon moment, i s'estan llençant molts títols al mercat. Aquest increment de la popularitat d'aquest gènere ha fet que els estudis de videojocs hagin hagut d'innovar en els nous títols per poder oferir experiències diferents als jugadors, fent que el gènere hagi evolucionat des de sobreviure a una illa deserta o a un bosc, fins a poder escollir altres ambientacions com l'oceà (*Subnautica*), l'espai (*No Man's Sky*) o la mitologia nòrdica (*Valheim*), entre moltes altres.

Es va decidir de fer un projecte de final de grau que tingues relació amb la regió de les Terres de l'Ebre, concretament la Ribera d'Ebre, ja que es el lloc de procedència dels participants, i es volia donar visibilitat a aquesta zona de Catalunya que només es coneix per centrals nuclears, abocadors o macroprojectes solars. També es volia mostrar com era la vida en aquesta zona a finals del segle XIX i principis del segle XX, ja que ha canviat molt en pocs anys, i la gent més jove no coneix com van viure els seus iaios.

Per aquests motius es va decidir implementar un videojoc que aprofites algunes de les mecàniques bàsiques dels jocs de supervivència per tal d'explicar als jugadors com era la vida als pobles de la Ribera d'Ebre durant el segle passat, introduint tècniques, recursos o eines tradicionals i explicant el seu funcionament.

1.2. Objectius del projecte

L'objectiu general del projecte és crear una base de mecàniques sòlida i modular a sobre de la qual es puguin afegir, de forma senzilla, els continguts relacionats amb la forma de vida als pobles de la Ribera d'Ebre durant el començament del segle passat. A partir d'aquesta base es desenvoluparà un joc que expliqui algunes d'aquestes tècniques, però posant molt èmfasi en fer un disseny sòlid i flexible on sigui senzill seguir afegint nous contingut.

Altres objectius més concrets són:

- Endinsar-se en el funcionament d'Unreal Engine 4.

- Introduir-se en el desenvolupament de jocs combinant codi C++ i Blueprints (llenguatge visual a partir de nodes), tal i com es fa en projectes professionals.
- Dissenyar el nucli del joc de forma sòlida i modular intentant fer servir bones pràctiques pel que fa a la programació, tant en C++ com en Blueprints.
- Introduir-se en en l'ús d'animacions, sons i shaders avançats amb Unreal Engine 4.
- Documentar-se sobre el context històric i l'estil de vida en les zones rurals de la Ribera d'Ebre durant l'època de finals del segle XIX i començament del segle XX.
- Implementar un prototip del joc on s'hagin implementat algunes de les tècniques estudiades.

1.3. Distribució de tasques

La realització d'un videojoc implica moltes tasques diferents i molt variades que a grans trets es poden agrupar en quatre categories diferents. A la Figura 1.1 es pot veure la distribució del treball entre aquestes categories.

Estètica	15%
Narrativa	10%
Mecàniques	30%
Tecnologia	45%

Figura 1.1: Taula d'autoavaluació

Els motius per haver escollit aquests percentatges s'expliquen a continuació.

- **Estètica:** S'ha escollit una estètica *lowpoly*, i al tractar-se d'un equip de desenvolupament amb un perfil més tècnic que artístic, s'ha decidit adquirir alguns recursos a tercers per tal de poder centrar els esforços en dissenyar una base sòlida per poder afegir nous continguts. Tot i això, s'ha donat un pes a aquest apartat, ja que s'ha treballat bastant per cuidar alguns aspectes visuals del joc, com els *shaders* o les animacions.
- **Narrativa:** Tot i que el videojoc no tindrà una història com a tal, s'ha considerat donar un pes a aquest apartat per representar les tasques de recerca sobre la vida de l'època en la que s'ambienta el joc, i cercar formes per explicar aquests conceptes al llarg del joc.

- **Mecàniques:** S’implementaran una sèrie de mecàniques bàsiques típiques dels jocs de supervivència, com un sistema d’inventari i crafteig, un sistema de recol·lecció o un sistema de construcció, entre altres.
- **Tecnologia:** El projecte s’implementarà combinant codi en C++ i Blueprints, i disposarà d’una base sòlida de funcionalitats programada en C++ que s’ampliarà de forma ràpida i senzilla mitjançant Blueprints. D’aquesta forma es buscarà obtenir les avantatges d’utilitzar un llenguatge de programació escrit per gestionar els aspectes més complexos de la lògica del joc, però sense renunciar a la versatilitat i rapidesa que ofereix un llenguatge de programació visual a l’hora de crear i modificar objectes dintre del món del joc. Es posarà molt èmfasi en dissenyar bones pràctiques per tal d’integrar els dos sistemes de programació que proporciona Unreal Engine 4.

2. Estudi de viabilitat

Abans de començar a treballar en un projecte gran, com és el cas d'un videojoc, cal assegurar-se que aquest sigui viable en tots els sentits, tant des d'un punt de vista econòmic, tecnològic i legal, com també des d'un punt de vista de mercat, realitzant-ne un petit estudi per poder posicionar aquest joc entre altres propostes ja existents.

2.1. Viabilitat del projecte

S'estudiarà la viabilitat del projecte des de diferents punts de vista: El tecnològic, on es considerarà el software i el hardware necessari, l'econòmic, on s'estimarà el cost total del projecte considerant software, hardware i recursos humans; i finalment el punt de vista legal, on s'estudiarà la viabilitat legal del projecte.

2.1.1. Viabilitat tecnològica

Software necessari:

- Documentació: Paquet de LibreOffice
- Planificació: Hack'n Plan
- Implementació:
  - Motor de joc: Unreal Engine 4
  - Entorn de desenvolupament: Visual Studio 2019
- Disseny:
  - Edició d'imatges: GIMP i Inkscape
  - Edició de sons: Audacity
  - Modelat 3D: Blender
- Altre software:
  - Notepad++

Hardware necessari:

- 1 ordinador que pugui treballar amb Unreal Engine 4 de forma fluida

A la Figura 2.1 s'indiquen els requeriments de hardware recomanats extrets de la documentació oficial de Unreal Engine [1], i a la Figura 2.2 s'indiquen les especificacions de l'ordinador que s'ha utilitzat per desenvolupar el projecte.

SO	Windows 10 64-bit
CPU	Quad-core 2.5GHz
RAM	8GB
GPU	DirectX11 o 12

Figura 2.1: Hardware recomanat per treballar amb UE4 (font Documentació UE4)

SO	Windows 10 64-bit
CPU	AMD Ryzen 7 1700 3.00GHz
RAM	32GB
GPU	AMD RX 5700 DX12
Disk	1TB SSD NVMe

Figura 2.2: Hardware de l'ordinador utilitzat per desenvolupar el projecte

2.1.2. Viabilitat econòmica

Software necessari: Tot el software utilitzat és gratuït, i s'ha intentat utilitzar el màxim de software lliure possible. La llicència d'ús d'Unreal Engine 4 permet utilitzar-lo de forma gratuïta mentre els ingressos del producte no superin el milió de dòlars ($1.000.000USD), moment en el què s'ha de pagar un *royalty* del 5%.

Hardware necessari: El cost del hardware recomanat per treballar amb Unreal Engine 4 estaria entre els 1000€ i 1500€, aproximadament.

Recursos estètics:

- Polyperfect Lowpoly Ultimate Pack [2]: 53,60€
- Another Stylized Material Collection 8 [3]: 9,92€
- Gamemaster Audio Prosound Minipack [4]: 18.15$

Tots els recursos estètics utilitzats s'han adquirit a la Humble Store com a part de *bundles* relacionats amb el desenvolupament de videojocs. Els paquets adquirits han estat els següents: *Humble Unity Games and Game Dev Assets Bundle,* per un preu de 25€ i *Humble Unreal Engine Game Development Bundle*, per un preu de 20€.

Recursos humans: En el procés de desenvolupament d'un videojoc participen moltes persones amb rols molt diferents, fent una classificació molt general es poden diferenciar quatre perfils de treballadors: dissenyadors, programadors, artistes visuals i artistes de so. El desenvolupament d'aquest projecte serà realitzat per una persona durant un any, però en el cas hipotètic d'haver de contractar un equip amb un treballador especialitzat per cada rol, i considerant que no tots els treballadors haurien de treballar tots els mesos, el cost de desenvolupament seria aproximadament el següent:

- Dissenyador: 2500€/mes x 10 mesos = 25000€
- Programador: 2900€/mes x 12 mesos = 34800€
- Artista visual: 2100€/mes x 3 mesos = 6300€
- Artista de so: 2500€/any x 2 mesos = 5000€

A la Figura 2.3 es mostra un quadre amb un resum dels costos econòmics hipotètics comparats amb els costos econòmics reals del projecte:

	Hipotètic	Real
Software	0 €	0 €
Hardware	1.500 €	0 €
Estètica	81 €	45 €
RRHH	71.100 €	0 €
TOTAL:	72.681 €	45 €

Figura 2.3: Comparativa entre el cost hipotètic i el cost real del projecte

2.2. Estudi de mercat

Actualment al mercat hi ha un gran nombre de títols de jocs del gènere de supervivència, s'estudiaran alguns d'aquests jocs i es posicionarà aquest projecte respecte al mercat.

2.2.1. Estat de l'art

L'anàlisi fet dels jocs que s'han considerat més rellevants es basarà en les característiques principals, tant des del punt de vista de la jugabilitat com des del punt de vista artístic.

Minecraft

Minecraft és un videojoc de supervivència publicat l'any 2011, i, amb més de 200 milions de còpies venudes, s'ha convertit en el videojoc més venut de la història, essent el videojoc de supervivència més conegut a escala global.

Pel que fa a la jugabilitat, *Minecraft* posa al jugador sense res al centre d'un món generat proceduralment. A partir d'aquí el jugador ha d'anar obtenint recursos que li permetin crear eines més avançades amb les quals poder obtenir nous recursos. L'objectiu del jugador és adaptar-se a l'entorn per poder recol·lectar recursos durant el dia i defensar-se dels atacs dels monstres durant la nit.

Un dels pilars fonamentals d'aquest joc és el sistema de construcció, que permet als jugadors modificar el món de forma lliure per crear les estructures que vulguin, fomentant la imaginació i la creativitat.

Estèticament és un joc molt senzill, ja que només està format per cubs. Aquesta estètica tan única l'hi ha permès diferenciar-se de la resta de jocs del mercat i esdevenir un joc icònic.

Figura 2.4: Captura de pantalla del Minecraft.

Stardew Valley

Stardew Valley és un títol independent publicat l'any 2016, que tot i no ser un joc de supervivència pur, inclou molts elements típics del gènere.

Pel que fa a la jugabilitat, el jugador hereta la granja del seu avi i es muda al poble de Stardew Valley a iniciar una nova vida. El jugador haurà d'anar restaurant aquesta granja mentre explora el poble i els voltants i estableix vincles amb els habitants del municipi.

Aquest joc inclou cinc mecàniques principals implementades mitjançant un sistema d'habilitats tipus RPG, on el jugador pot anar escollint diferents avantatges a mesura que va millorant en cadascuna de les branques. Les mecàniques que implementa són: recol·lecció, agricultura/ramaderia, pesca, mineria i combat.

Un dels seus punts forts és la seva estètica *pixelart* molt polida, amb molts detalls i escenaris diferents molt cuidats.

Figura 2.5: Captura de pantalla de Stardew Valley

Valheim

Valheim també és un títol independent que es va publicar el febrer de 2021, i que en només un mes va aconseguir vendre més de cinc milions de còpies. És un videojoc de supervivència pur que inclou lluites contra *bosses (enemics que van modificant el seu comportament al llarg del joc)*, i està ambientat en un món fantàstic basat en elements de la mitologia nòrdica.

Pel que fa a la jugabilitat, el jugador és transportat a un món de fantasia on tindrà la missió de derrotar una sèrie de *bosses* basats en criatures mitològiques nòrdiques. Per fer-ho el jugador haurà d'explorar els diferents biomes buscant materials que li permetin construir eines més avançades.

El joc implementa mecàniques d'exploració i fa servir múltiples escenaris amb enemics i recursos únics, també implementa mecàniques avançades de construcció, permetent als jugadors crear les seves pròpies bases, i mecàniques de lluita amb diferents tipus d'armes i escuts. A més a més, també disposa d'un sistema de crafteig que permet construir elements que desbloquegen noves receptes.

Figura 2.6: Captura de pantalla de Valhiem

The Long Dark

The *Long Dark* és un altre títol independent publicat l'any 2014, i busca oferir una experiència de supervivència molt més realista que els altres títols. Per fer-ho no introdueix cap tipus d'enemic i ofereix una experiència on el jugador ha d'intentar superar les dificultats de la naturalesa.

Pel que fa a la jugabilitat, el jugador controla a un personatge que ha tingut un accident d'avió i ha caigut a una zona del nord de Canadà, on el seu objectiu serà no morir. El jugador haurà de fer front a les fredes temperatures d'una forma molt realista, havent de considerar paràmetres com la intensitat del vent o la humitat per incrementar les opcions de sobreviure.

Les mecàniques d'exploració i de crafteig que implementa aquest joc tenen un enfocament molt més realista, ja que es poden fabricar poques coses i no gaire grans. A més a més, la gestió de l'inventari resulta fonamental, ja que com més pes porti el jugador, més risc tindrà de patir algun tipus de dany.

Figura 2.7: Captura de pantalla de The Long Dark

Among Trees

Among Trees és un títol independent publicat l'any 2020. Està molt enfocat en oferir una experiència de joc tranquil·la i agradable, amb un estil artístic molt característic i molt polit. No hi ha enemics tot i que alguns animals salvatges es poden atacar.

Pel que fa a la jugabilitat, el jugador controla un personatge que ha de sobreviure en una cabanya al mig d'un bosc. L'objectiu del jugador és explorar el mapa per trobar recursos i receptes que li permetran desbloquejar nous elements.

El joc implementa mecàniques d'exploració i crafteig, però per desbloquejar noves receptes s'han d'aconseguir els plànols, que es troben amagats pel mapa. També implementa un sistema de construcció on el jugador pot anar millorant la cabanya inicial afegint-hi noves àrees que permeten desbloquejar mecàniques noves.

Figura 2.8: Captura de pantalla de Among Trees

2.2.2. Posicionament respecte els jocs analitzats

Per posicionar aquest projecte respecte els jocs analitzats prèviament, es puntuaran diferents aspectes per tal de poder-los comparar. Concretament s'estudiaran els següents paràmetres:

- Estètica: Es tindrà en compte el tipus de gràfics i la originalitat. Es compararà l'estil artístic i s'avaluarà l'originalitat d'aquest valorant-lo entre *poc original* i *molt original*.

	Estètica	
	Estil artístic	Originalitat
Minecraft	3D només cubs	Poc original
Stardrew Valley	2D Pixel Art	Molt original
Valheim	3D estil PS1	Original
The Long Dark	3D Lowpoly dibuixos a mà	Molt original
Among Trees	3D Lowpoly càlid	Original
Projecte	3D Lowpoly	Original

Figura 2.9: Avaluació de l'estètica

- Mecàniques: Es valorarà el nombre de mecàniques implementades i com de polides estan. Es consideraran les mecàniques bàsiques dels jocs de supervivència (recol·lecció R, crafteig CR, construcció CN, pesca P, combat CB i agricultura A) i s'assignarà un ✓ en cas que estiguin ben implementades, un ~ en cas que estiguin

implementades però de forma superficial o un ✗ en cas que no estiguin implementades en absolut.

	Mecàniques					
	R	CR	CN	P	CB	A
Minecraft	✓	✓	✓	✓	✓	~
Stardrew Valley	✓	✓	✓	✓	✓	✓
Valheim	✓	✓	✓	~	✓	~
The Long Dark	✓	✓	✗	~	~	✗
Among Trees	✓	✓	~	~	~	✓
Projecte	✓	✓	✓	✗	✗	✓

Figura 2.10: Avaluació de les mecàniques

- Ambientació: Es valorarà l'originalitat de l'ambientació del joc. Es compararà la ubicació i s'avaluarà l'originalitat d'aquesta valorant-la entre *poc original* i *molt original*.

	Ambientació	
	Ubicació	Originalitat
Minecraft	Món fictici amb diferents biomes	Poc original
Stardrew Valley	Antiga granja en un poblet	Original
Valheim	Món fíctici basat en la mitologia nòrdica	Original
The Long Dark	Zona muntanyosa al nord de Canadà	Original
Among Trees	Bosc	Poc original
Projecte	Poblet de la Ribera d'Ebre	Original

Figura 2.11: Avaluació de l'ambientació

- Tipus de supervivència: Es valorarà el tipus de supervivència (contra l'entorn, contra enemics, realista...). Es compararan els perills a que ha de sobreviure el jugador i s'avaluarà el grau de realisme de la supervivència entre *molt realista* i *poc realista*.

	Supervivència	
	Perills	Realisme
Minecraft	Entorn, diferents monstres	Realista
Stardrew Valley	Enemics només a les mines	Poc realista
Valheim	Entorn, diferents monstres	Poc realista
The Long Dark	Entorn, clima	Molt realista
Among Trees	Entorn	Poc realista
Projecte	Entorn	Poc realista

Figura 2.12: Avaluació del tipus de supervivència

- Preu: Es valorarà el preu de mercat actual (sense ofertes)

	Preu
Minecraft	23,95 €
Stardrew Valley	13,99 €
Valheim	16,79 €
The Long Dark	24,99 €
Among Trees	12,99 €
Projecte	7,99 €

Figura 2.13: Avaluació del preu

2.3. Públic objectiu i perfil de jugador

A l'hora de plantejar el disseny d'algun producte és important definir a qui anirà dirigit, per poder prendre les decisions de disseny que s'adaptin millor al públic objectiu.

En el cas d'aquest projecte s'han tingut en compte diferents paràmetres per definir el públic objectiu:

- Tipus de supervivència: S'ha considerat el nivell de realisme que oferirà la supervivència. S'ha posicionat el públic objectiu del projecte entre un tipus de jugador de *supervivència casual*, com podria ser la representada a *Stardrew Valley*, on el jugador no perd vida al llarg del temps, i un tipus de jugador de *supervivència hardcore,* com la representada a The *Long Dark*, on el jugador necessita menjar, beure, escalfar-se, etc. per no perdre vida. A la Figura 2.14 es pot observar la posició del públic objectiu segons aquests valors mencionats:

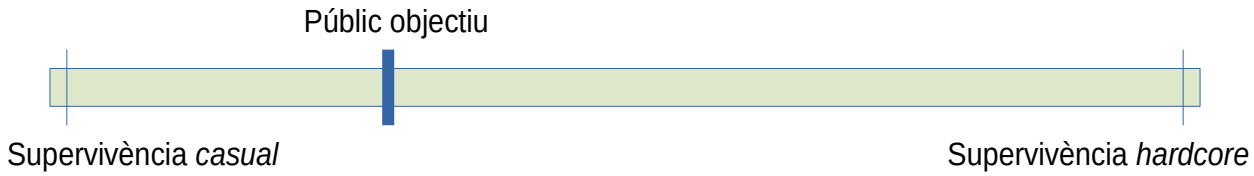


Figura 2.14: Públic objectiu del projecte segons el nivell de realisme de la supervivència

- Profunditat de les tècniques representades: S'ha considerat el nivell de detall en que es representaran les diferents tècniques que s'explicaran al jugador. S'ha posicionat el públic objectiu del projecte entre un públic seriós el qual juga al videojoc per aprendre un concepte, i un públic més casual, el qual juga per divertir-se. A la Figura 2.15 es pot observar la posició del públic objectiu segons aquests valors mencionats:

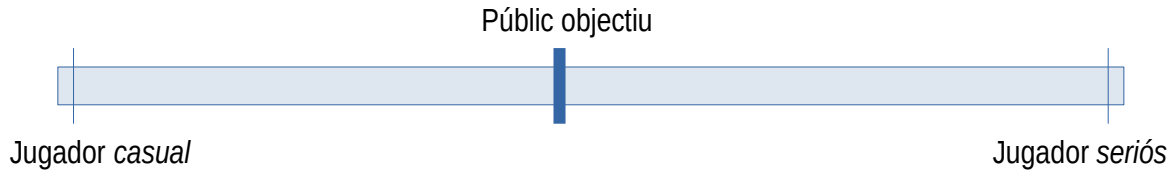


Figura 2.15: Públic objectiu del projecte segons la predisposició a l'aprenentatge

- Estil artístic: S'ha considerat l'estètica del joc en funció a l'edat del públic objectiu. Mentre un joc amb gràfics més realistes donaria més immersió, un joc amb gràfics més *cartoon* donaria un aspecte més agradable per a tots els públics.

A partir d'aquest anàlisi s'ha decidit enfocar el joc a un públic objectiu a partir d'uns 7 anys, ja que els jocs de supervivència tenen un component de gestió i planificació important que pot ser complicat per a jugadors amb poca edat. Pel que fa al nivell de realisme de la supervivència s'ha decidit enfocar el joc als jugadors de supervivència casuals però amb algun toc de realisme, i finalment s'ha decidit enfocar el joc a un públic que busqui divertir-se jugant, però que tingui una certa predisposició a aprendre sobre l'ambientació del joc a mesura que va jugant.

Un cop definit el públic objectiu al qual es dirigirà el projecte s'ha de definir el perfil de jugadors als qual anirà enfocat el joc. Per fer-ho s'utilitzarà la classificació de jugadors plantejada per Richard Bartle, i que defineix quatre tipus diferents de jugadors:

- **Achievers**: Busquen l'acció. L'objectiu és superar els reptes que planteja el joc, ja sigui completant nivells, obtenint millor equipament o intentant obtenir la màxima puntuació possible.
- **Explorers**: Busquen la interacció amb el món. L'objectiu és descobrir al màxim els diferents escenaris plantejats pel joc.
- **Socializers**: Busquen la interacció amb els jugadors. L'objectiu és interactuar amb altres jugadors, i utilitzen el joc com un mitjà per a conèixer altres persones.
- **Killers**: Busquen la acció amb els jugadors. L'objectiu és únicament ser els millors del joc o de la partida.

Segons aquesta classificació s'ha considerat que el perfil de jugadors que més encaixa a la idea del projecte és el tipus **Explorer**, ja que el joc busca que el jugador vagi progressant a mesura que va descobrint nous recursos, noves tècniques o noves ubicacions.

2.4. Conclusions

Com s'ha vist en els apartats anteriors, el projecte és tècnicament i econòmicament viable, atès que ja es disposa dels recursos tècnics i només s'ha de fer una petita inversió per obtenir els recursos visuals, permetent centrar molts més recursos en la implementació del joc i no haver de preocupar-se de realitzar les tasques artístiques.

Pel que fa a l'estudi del mercat s'observa que el projecte busca explotar un gènere del qual hi ha molts títols, però d'una forma diferent, plantejant un repte de supervivència en una època i en una ubicació geogràfica que encara no han estat explorades per altres projectes i que poden resultar interessants tant pel públic que pugui ser més proper a aquesta realitat, com per part del públic que no la conegui en absolut.

3. Planificació

Un cop definides les idees principals del projecte, cal començar a treballar en la planificació. En aquest punt es descriurà tot el procés de planificació del projecte, des de la proposta de la idea al tutor, feta el juny del 2020 fins a l'entrega del projecte, el juny del 2021. Es presentarà la metodologia de treball seguida, els paquets de tasques definits, els objectius escollits i finalment la planificació temporal representada a través d'un diagrama de Gantt.

3.1. Metodologia de treball

Cal diferenciar la metodologia emprada per fer el seguiment del treball per tal d'implementar el projecte.

- **Metodologia de seguiment:** Per gestionar i organitzar el treball es realitzaran reunions cada quinze dies amb el tutor, en les quals es mostraran els avenços realitzats en la implementació i s'organitzaran els següents passos a seguir.
- **Metodologia d'implementació:** Per implementar el joc s'ha decidit utilitzar una estratègia Modular *Top Down*, que consistirà en fer una especificació detallada del què ha de fer el joc i després dividir les diferents funcionalitats en mòduls independents. S'ha decidit seguir aquesta metodologia ja que ofereix molta flexibilitat a l'hora de treballar, i permet la reutilització de mòduls en diferents projectes. Els mòduls en què s'ha organitzat aquest projecte són:
    - Mòdul de vida: S'encarregarà de la gestió de la vida i de la gana del jugador.
    - Mòdul d'inventari: S'encarregarà de gestionar els ítems que té guardats el jugador, permetent d'afegir-ne o treure'n.
    - Mòdul de crafteig: S'encarregarà de convertir ítems de l'inventari en nous ítems mitjançant receptes predefinides.
    - Mòdul d'eines: S'encarregarà de gestionar les diferents eines que pot crear el jugador, i de controlar el seu funcionament.
    - Mòdul de joc: S'encarregarà de gestionar el funcionament del joc controlant els inputs, les interfícies... Serà el mòdul central, encarregat de coordinar tots els altres mòduls mencionats anteriorment.

3.2. Paquets de treball

Els diferents blocs en què s'organitzarà el treball del projecte seran:

- Paquet de documentació:
  - Tasques:
    - Documentar de forma provisional els progressos en la implementació.
    - Documentar la memòria final.
  - Temporització: 11 mesos.
  - Objectiu: Tenir informació bàsica per poder fer la memòria del projecte.
- Paquet de narrativa:
  - Tasques:
    - Recerca sobre el context històric, les tècniques i l'estil de vida de l'època on s'ambienta el joc.
    - Integrar la recerca realitzada al joc.
  - Temporització: 4 mesos.
  - Objectiu: Tenir la informació que permeti definir les tècniques a implementar, així com una història com a fil conductor del joc.
- Paquet d'organització:
  - Tasques:
    - Implementar un nucli del joc sòlid integrant Blueprints i C++.
    - Definir l'organització de classes i d'herència.
  - Temporització: 5 mesos.
  - Objectiu: Estructura que implementi la base del joc i on sigui senzill afegir nous continguts.
- Paquet de mecàniques:
  - Tasques:
    - Definir les principals mecàniques del joc.

        - Implementar aquestes mecàniques.
        - Implementar les interfícies requerides per la interacció del jugador.
    - Temporització: 8 mesos.
    - Objectiu: Tenir un prototipus del joc totalment funcional per tal de poder ser provat per fer una avaluació inicial per part d'altres jugadors.
- Paquet d'estètica:
    - Tasques:
        - Definir l'estil artístic
        - Integrar els *assets (elements del joc)* amb les mecàniques desenvolupades.
        - Definir i implementar els menús necessaris.
    - Temporització: 4 mesos.
    - Objectiu: Joc amb tots els elements visuals implementats.
- Paquet de test:
    - Tasques:
        - Provar el joc cercant errors
        - Preparar una demostració per poder obtenir feedback extern
        - Analitzar el feedback obtingut i corregir els errors/coses millorables
    - Temporització: 8 mesos
    - Objectiu: Joc acabat (es pot jugar el joc complet sense errors)

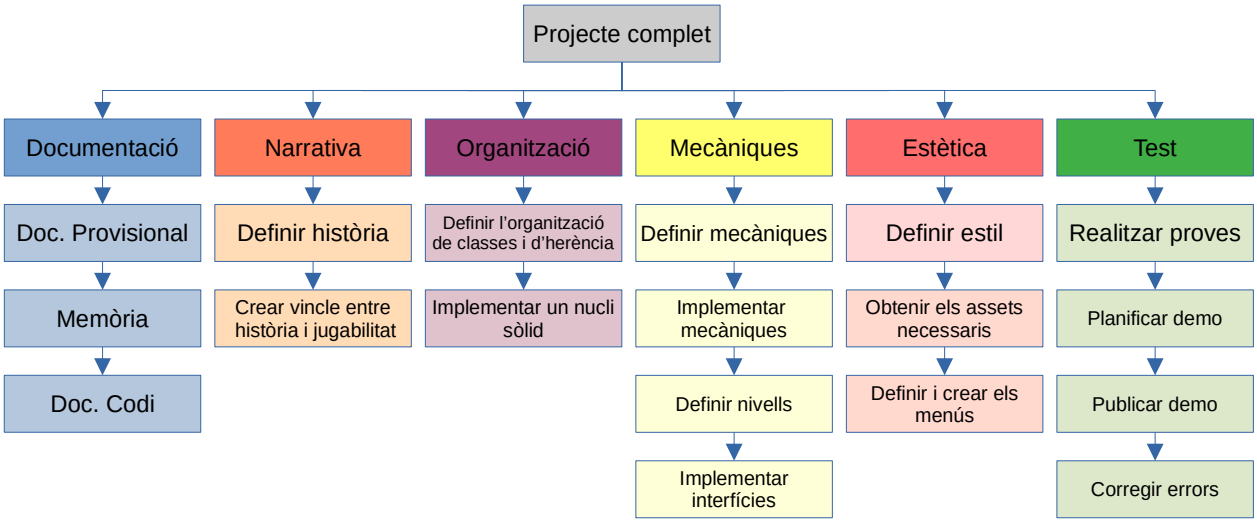


Figura 3.1: Diagrama de paquets de treball

3.3. Diagrama de Gantt

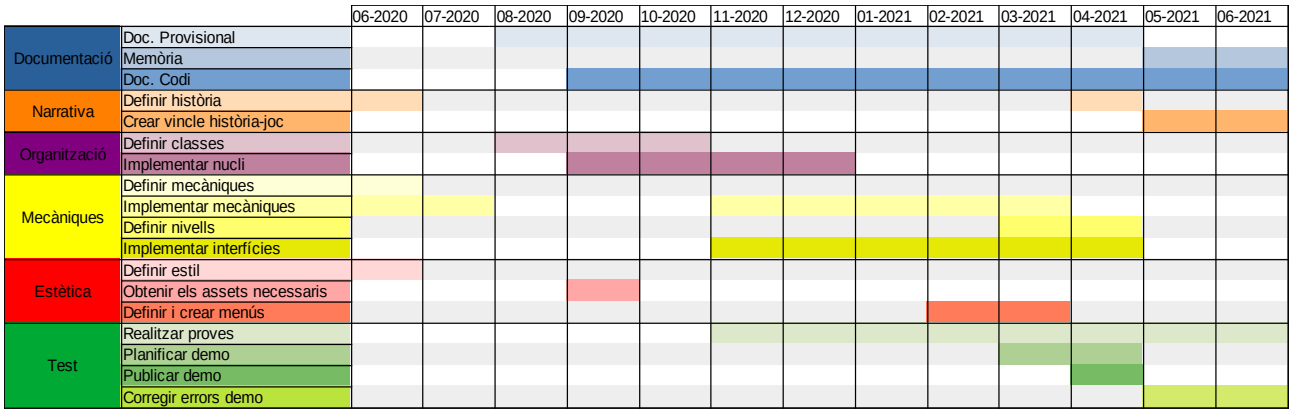


Figura 3.2: Diagrama de Gantt

4. Marc de treball i conceptes previs

Com a punt de partida cal tenir en compte la forma de funcionar d'alguns jocs semblants que s'han pres com a referent per definir el funcionament del joc. Al centrar el projecte molt més en la part tècnica que en la part artística, no s'han tingut en compte els referents artístics.

Per facilitar la comprensió de les posteriors explicacions, es farà una breu introducció a alguns conceptes previs de Unreal Engine 4 que resultaran necessaris.

4.1.1. Referents tècnics

A l'estar l'equip de desenvolupament molt familiaritzat amb el gènere de la supervivència, no s'ha considerat agafar un únic referent a l'hora de definir les mecàniques bàsiques del joc, ja que, com s'ha pogut observar a l'estudi de viabilitat, són implementades per la majoria de jocs del gènere. No obstant, s'adoptaran algunes idees interessants plantejades per altres jocs i es modificaran per tal de poder-les integrar dins el projecte.

- Crafteig: El crafteig s'ha basat en les mecàniques de *Stardrew Valley* i de *Valheim*. *Stardrew Valley* permet construir màquines que converteixen ítems bàsics en ítems d'un valor superior, com per exemple convertir una peça de fruita en melmelada. Aquestes mecàniques tenen un comportament únic: el jugador posa un recurs X i obté un recurs Y, sempre de la mateixa manera. *Valheim* implementa un sistema de crafteig format per diferents zones, ja que considera que no és el mateix crear una eina de metall que crear una peça de roba per al personatge. A partir d'aquestes dues implementacions, s'ha optat per implementar un sistema de crafteig barrejant els dos conceptes exposats: el jugador podrà construir diferents màquines que podran funcionar com a zones de crafteig per a crear nous elements a partir de diferents tipus de recursos.

4.2. Conceptes previs

Per entendre la implementació feta, cal entendre primer el funcionament bàsic d'Unreal Engine 4, i la seva organització, això permetrà explicar de forma més senzilla el funcionament del joc que es farà en els posteriors apartats.

4.2.1. Introducció a Unreal Engine 4

Unreal Engine 4 és un motor de videojocs programat en C++ i de codi obert implementat per l'empresa Epic Games. A diferència d'altres motors de videojocs més centrats en equips petits, Unreal Engine 4 està enfocat a equips de desenvolupament grans, on hi ha treballadors especialitzats en cadascuna de les diferents tasques que comporten desenvolupar un joc. El motor implementa múltiples eines per permetre que cada perfil de treballador pugui centrar-se només en les seves tasques específiques sense necessitat de conèixer com funcionen la resta de parts del joc.

Figura 4.1: Logo d'Unreal Engine 4

4.2.2. Estructura d'Unreal Engine 4

Els principals tipus d'objectes que es troben dintre del motor són els següents:

Object: És la classe base dintre del motor. Tots els subsistemes que formen el motor hereten d'aquesta classe. Implementa característiques com un *garbage collection,* suport per exposar variables des del codi a la interfície del motor, serialització per poder guardar els objectes, etc.

Actor: És un `Object` que pot ser col·locat dintre d'un nivell, com per exemple una càmera, un model 3D, un trigger... Els objectes del tipus `Actor` suporten transformacions en un espai tridimensional, i poden ser creats i destruïts de forma dinàmica a través d'scripts.

Component: És un fragment de funcionalitat que pot ser afegida a un `Object`. Els `Component` no poden existir per ells mateixos, sinó que sempre han d'estar vinculats a un `Object`. Aquests elements permeten implementar funcionalitats de forma modular, i

incorporar-les a diferents objectes. Hi ha diferents tipus de components, però els més utilitzats són els *ActorComponent*, que es poden afegir a objectes del tipus *Actor*.

Per il·lustrar-ho de forma més clara es mostrarà un sistema de vida implementat mitjançant components. Primer cal definir la classe que implementarà la funcionalitat del component. En aquest cas s'ha definit una classe senzilla anomenada *UHealthComponent* que té únicament dos atributs per saber el nivell de vida per defecte del component i el nivell de vida actual, i dos mètodes que permeten treure o afegir una quantitat donada a la vida actual.

```
class UHealthComponent : public UactorComponent {

public:
	UHealthComponent();

	UFUNCTION()
	void takeDamage(float damage);

	UFUNCTION()
	void gainHealth(float health);

private:
	UPROPERTY()
	float DefaultHealth;

	UPROPERTY()
	float Health;
};
```

Un cop definida i implementada la funcionalitat d'aquest component, es pot afegir a tots els objectes del tipus *Actor* que hi hagi dintre del projecte. A la Figura 4.2 es pot observar com s'ha afegit el component creat anteriorment a l'objecte *Actor* corresponent al jugador per defecte d'Unreal Engine.

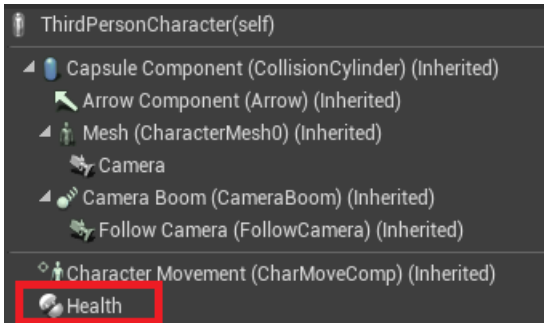


Figura 4.2: Jugador per defecte d'Unreal Engine 4 al qual se l'hi ha afegit el component de vida definit anteriorment

Pawn: És una subclasse d'Actor que serveix per a representar personatges dintre del joc (els personatges no cal que siguin persones per poder ser Pawns). La principal característica dels objectes d'aquesta classe és que poden ser controlats pel jugador o per una intel·ligència artificial. Un *Pawn* controlat per un jugador o una AI es considera que està *possessed,* mentre que si no està controlat per ningú es considera que està *unpossessed.*

Character: És una subclasse de *Pawn* que està pensada per a ser utilitzada per representar el personatge del jugador. Per defecte implementa un personatge humanoide amb un sistema de moviment bàsic (*WASD* per al moviment i el ratolí per moure la càmera).

Controller: És una subclasse d'Actor que pot posseir un objecte del tipus *Pawn* (o derivats) i gestionar-ne el comportament. Per defecte existeix una relació 1:1 entre els *Controller* i els *Pawn*, tot i que pot modificar-se si el joc requereix que un únic *Controller* gestioni diferents *Pawn*. La subclasse *PlayerController* s'utilitza per implementar el control d'un *Pawn* per part d'un jugador humà, i la subclasse *AIController* s'utilitza per implementar el control d'un *Pawn* per part de la intel·ligència artificial.

GameMode: És un tipus d'objecte que determina les regles del joc. Qualsevol objecte instanciat al món té accés al *GameMode*.

Static Mesh: És el tipus de component com s'importen els models 3D dintre del motor.

Skeletal mesh: És un model 3D format per una static mesh i una jerarquia d'ossos interconnectats que es poden fer servir per animar el model.

A la Figura 4.3 es pot observar un diagrama de flux extret de la documentació oficial d'Unreal Engine 4 [5] on s'exposa la relació entre les classes bàsiques descrites anteriorment a l'hora d'organitzar un joc:

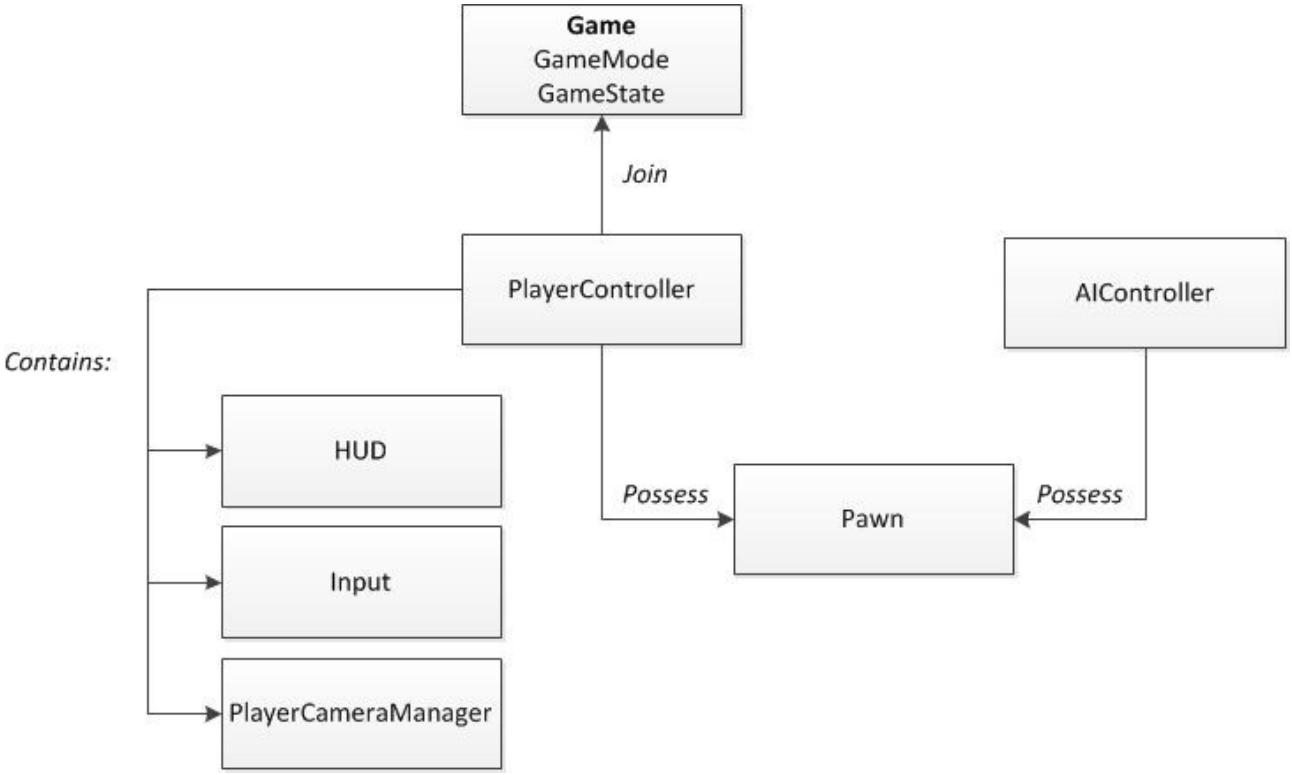


Figura 4.3: Relacions entre les classes bàsiques d'Unreal Engine 4

4.2.3. Scripting

Tot i que el nucli del motor està programat en C++, no és necessari saber programar per poder utilitzar-lo, ja que implementa un llenguatge de programació visual basat en nodes anomenat *Blueprint Visual Scripting*. A partir d'aquest moment es referirà com a *Blueprints* als objectes definits a través del llenguatge Blueprint Visual Scripting.

Aquest sistema permet programar de forma molt senzilla elements de jugabilitat a partir de nodes, sense necessitat d'escriure codi en C++ ni de recompilar el projecte. Les Blueprints estan pensades per treballar utilitzant programació orientada a objectes, de forma que sempre han d'heretar d'una classe pare implementada en C++ o en Blueprints. Aquest punt és clau a l'hora de dissenyar una estructura robusta, com veurem en els següents apartats.

A part d'aquest sistema de programació visual, el motor també permet treballar directament amb programes fets amb el llenguatge C++.

4.2.3.1. Scripting en Blueprints

El sistema de Blueprints es basa en nodes. Els nodes son fragments de codi que realitzen alguna funcionalitat, i estan implementats en C++ dintre del motor. A nivell d'organització les Blueprints són com classes de C++, ja que permeten definir atributs i mètodes tant públics com privats, suporten herència, es poden instanciar a través de codi... La Figura 4.4 mostra l'editor de Blueprints i les diferents parts que el formen.

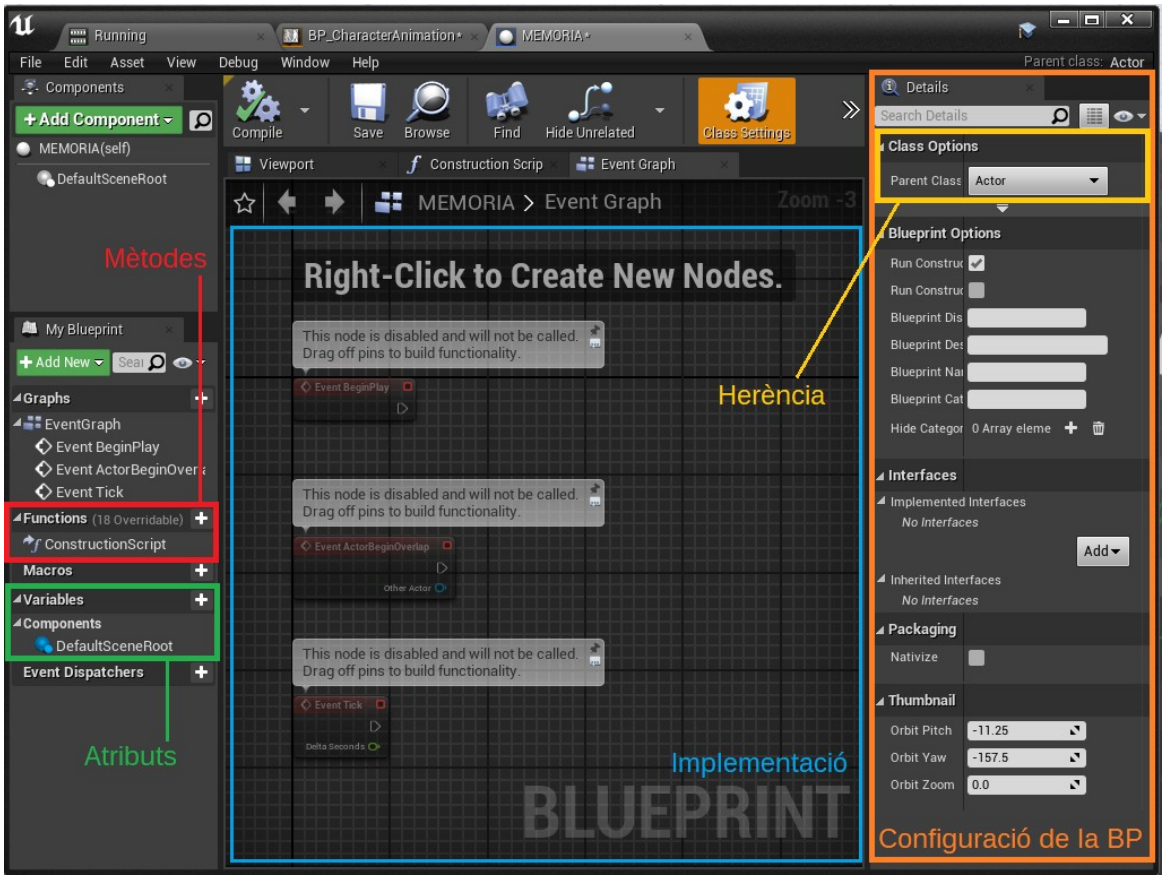


Figura 4.4: Editor de Blueprints

Per explicar com funciona la programació d'scripts a través de Blueprints es farà servir com exemple un script senzill que actualitza un comptador cada cop que es prem la tecla espai, abans però, cal introduir alguns conceptes referents al funcionament dels nodes.

Entrades i sortides: Els nodes poden tenir diferents connexions d'entrada i de sortida, que equivaldrien als paràmetres i als returns en les funcions d'un llenguatge de programació. Cal diferenciar entre les connexions d'execució i les connexions de dades. Les primeres es representen per una línia blanca que es connecta entre fletxes, i les segones es representen per una línia de color (segons el tipus de variable el color canvia) que es connecta entre punts. La Figura 4.5 mostra les connexions d'un node que incrementa el valor enter que li entra com a dada i com a sortida retorna el nou valor.

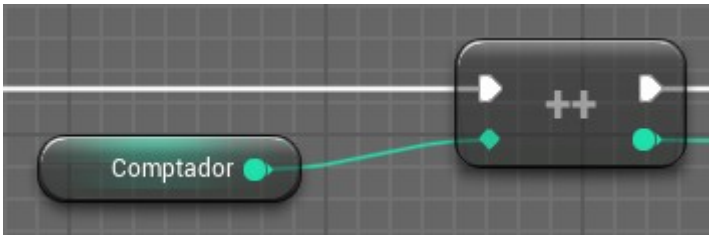


Figura 4.5: Connexions d'un node

Tipus de nodes: Unreal Engine 4 implementa molts nodes amb funcionalitats diferents. Per diferenciar-los de forma ràpida utilitza un codi de color segons el tipus de node:

- Nodes blaus: Implementen una funció que té una entrada i una sortida d'execució. A més, pot tenir o no entrades i sortides de dades.
- Nodes verds: Implementen funcions pures. La diferència entre les funcions normals és que aquestes només tenen connexions de dades, executant-se només quan fa falta saber el resultat.
- Nodes rojos: Representen esdeveniments que poden estar predefinits al motor (`BeginPlay`, inputs...) o definits pel desenvolupador.
- Nodes lila: Representen nodes fixes que no es poden crear ni destruir. Aquest tipus de nodes es troben a l'inici i al punt de sortida de les funcions.
- Nodes grisos: Són estructures de control del codi, com bucles, condicions, etc.

La Figura 4.6 mostra alguns exemples de cada tipus de nodes.

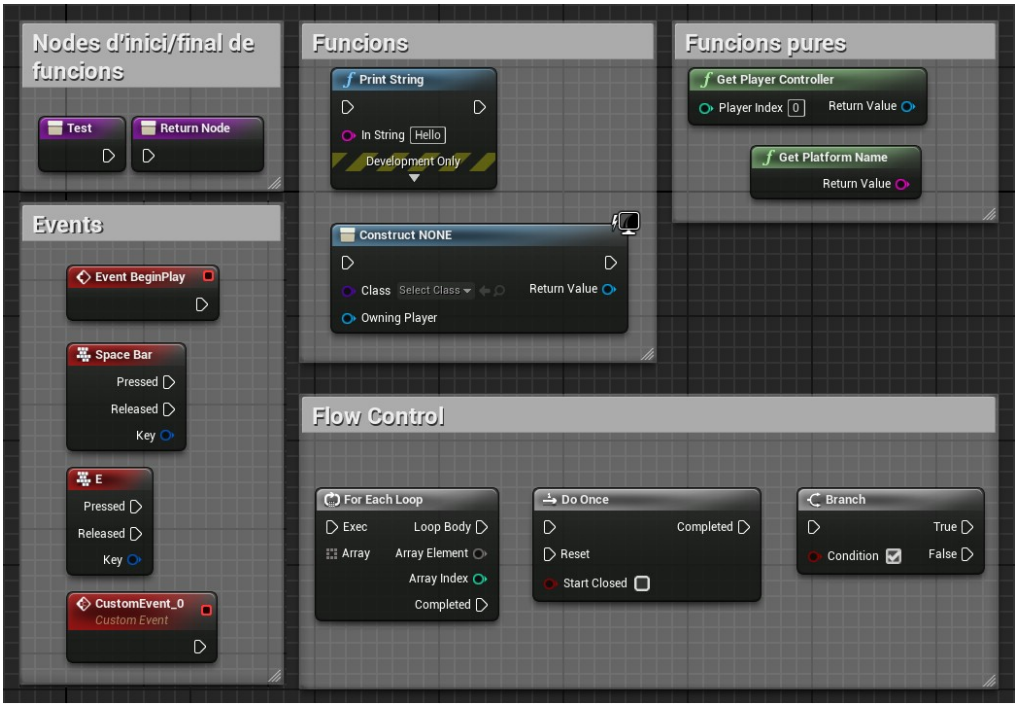


Figura 4.6: Tipus de nodes

Tipus de variables: De forma similar als tipus de nodes les variables també es representen de diferent color segons el tipus. La Figura 4.7 mostra tots els tipus de variables des de l'editor.

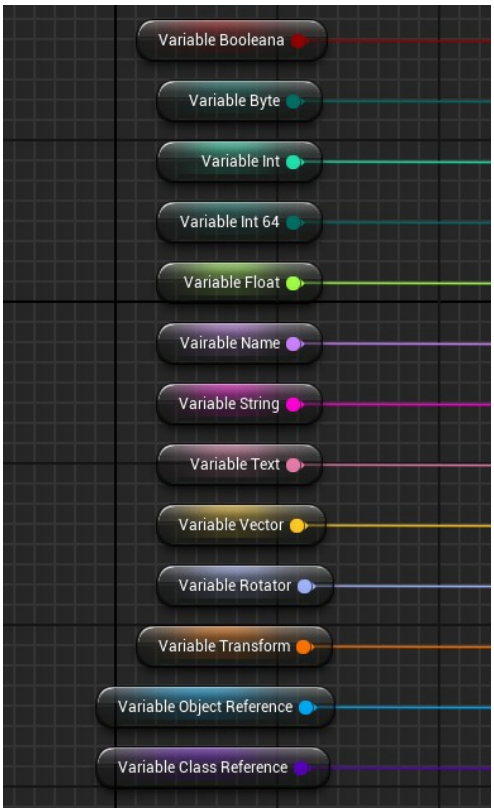


Figura 4.7: Tipus de variables

Ara que sabem com funcionen els nodes, podem analitzar un breu script que incrementarà un comptador al prémer la tecla espai. La Figura 4.8 mostra l'script complet.

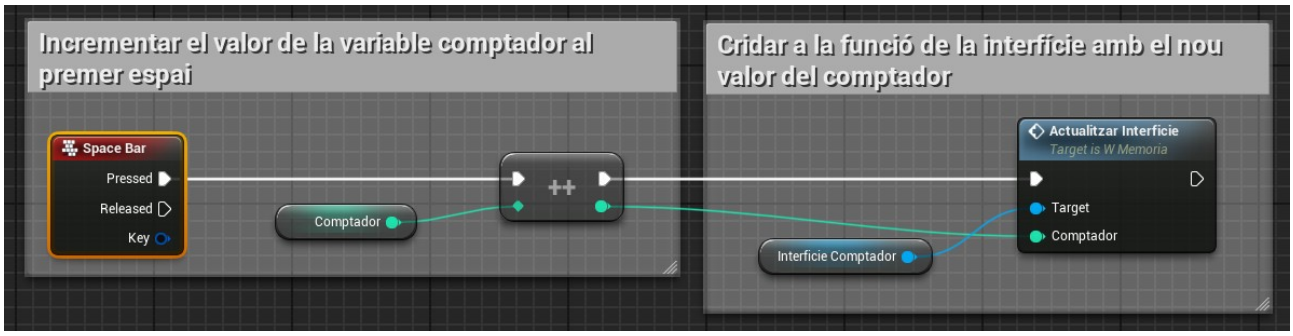


Figura 4.8: Exemple d'script amb Blueprints

Analitzant-lo d'esquerra a dreta s'observa un node d'esdeveniment que s'executa cada cop que el jugador premi l'espai. El flux de l'execució continua cap a un node gris que s'encarrega d'incrementar en 1 el valor de la variable entera que li entra. A l'entrada de dades s'ha connectat un node gris que retorna el valor d'un atribut de la classe. Un cop incrementat el comptador s'executa un node blau que implementa una funció que actualitza una interfície d'usuari (més endavant s'explicarà el funcionament de les interfícies). Aquesta funció rep dos paràmetres, un del tipus enter, on es connecta el resultat del node d'incrementar i un del tipus `Object`, on es connecta una referència a la interfície a actualitzar. L'script s'acaba aquí ja que d'aquest node no surt cap altre flux d'execució.

El llenguatge de Blueprints simplifica molt algunes tasques, però en alguns casos no és la millor alternativa. A mode de resum, es comentaran els principals punts forts i febles de les Blueprints.

Punts forts:

- Prototipatge ràpid: Permeten crear elements jugables funcionals de forma ràpida i senzilla.
- Iteració ràpida: No requereixen recompilar tot el projecte per provar els canvis a la lògica.
- Programació de scripts senzilla: A l'hora d'implementar un script que tingui un comportament predefinit (si el jugador prem aquest botó primer s'obre aquesta porta, després es reprodueix aquest so i finalment es genera una onada d'enemics) es més senzill seguir el flux d'execució a l'estar programat en nodes seqüencials.

- Flexibilitat: Les Blueprints són *assets de Unreal*, i per tant tenen accés a la resta de contingut del joc de forma dinàmica, això simplifica molt les tasques a l'hora de vincular altres *assets* a les Blueprints.

Punts febles:

- Menor eficiència: Les Blueprints s'executen dintre d'una màquina virtual, incrementant el nombre de passos interns previs fins a l'execució del codi a la CPU, especialment quan cal fer operacions matemàtiques o lògiques costoses.
- Dificultat quan s'incrementa la complexitat: A l'implementar lògiques complexes es poden obtenir Blueprints amb moltíssims nodes, dificultant la seva llegibilitat i complicant la depuració dels possibles errors.

4.2.3.2. Scripting en C++

Unreal Engine 4 també ofereix l'opció de treballar directament C++, ja sigui pur o fent servir una estructura de classes creada dintre el motor. Aquesta estructura defineix un conjunt de classes que hereten de la classe base `Object`, i permet als desenvolupadors utilitzar el garbage collector, accedir a una API de funcions, algunes d'elles equivalents als nodes de les Blueprints, o utilitzar estructures de dades optimitzades per al motor com els `TMap`, `TSet` o `TArray`.

A l'hora de definir mètodes o atributs en aquestes classes s'han d'utilitzar les macros `UFunction i UProperty,` que permeten que el *reflection system* d'Unreal Engine reconegui les definicions. A més aquests macros permeten utilitzar *Function Specifiers* i *Property Specifiers* per modificar com el motor interpreta els mètodes o atributs definits.

Els *Function Specifiers* més utilitzats en el projecte són:

- `BlueprintCallable`: Permet que una funció implementada en C++ es pugui cridar des de l'editor de Blueprints, creant el node del tipus corresponent amb les entrades i sortides definides via codi.
- `BlueprintImplementableEvent`: Permet que una funció definida en C++ s'implementi a l'editor de Blueprints com un esdeveniment. Útil per a implementar aspectes visuals en Blueprints i cridar-los des de C++.
- `CallInEditor`: Permet executar una funció definida en C++ des de l'editor del motor sense necessitat que el joc estigui en execució.

- `Category` = «Categoria»: Mostra les funcions dintre de categories personalitzades dintre de l'editor de Blueprints.

Els *Property Specifiers* més utilitzats en el projecte són:

- `BlueprintReadOnly`: Permet que una Blueprint pugui accedir al atribut però no permet que el modifiqui.
- `BlueprintReadWrite`: Dona accés per llegir i modificar l'atribut des de les Blueprints.
- `EditAnywhere`: Permet que un atribut pugui ser modificat des de l'editor del motor, des de la configuració d'una instància de l'objecte o des de l'editor de Blueprints.
- `Category` = «Categoria»: Mostra les variables dintre de categories personalitzades dintre de l'editor de Blueprints.

Al igual que les Blueprints, el codi C++ també presenta avantatges i inconvenients. Tot seguit es comentaran els principals punts forts i punts febles:

Punts forts:

- Major eficiència: Al compilar el codi s'obtenen temps d'execució inferiors als de les Blueprints.
- Major control sobre el disseny: És més senzill fer un disseny robust des d'un punt de vista de l'enginyeria del software.
- Major control sobre els permisos: És més senzill per als desenvolupadors determinar quines funcions o variables exposen o protegeixen.
- Accés a més funcionalitats del motor: C++ disposa d'accés a més funcions del motor que les Blueprints.
- Facilitat en els càlculs i la lògica complexa: És més senzill implementar càlculs i lògica complexa que en C++, i també ofereix una avantatge significativa en eficiència.

Punts febles:

- Accés als assets: Per a utilitzar un asset com podria ser un model 3D des de C++ s'ha de definir el path a l'arxiu de forma estàtica dintre del codi, dificultant així les

tasques de prototipatge, ja que qualsevol canvi en l'asset requereix una recopilació del codi.

- Dificultat per a no pogramadors: Al tractar-se d'un llenguatge de programació tradicional pot suposar una barrera d'entrada superior per a dissenyadors i artistes que no tenen experiència prèvia.

4.2.3.3. Integració entre C++ i Blueprints

Quan es comença a buscar informació sobre com introduir-se al funcionament d'Unreal Engine 4 sempre acaba apareixent la següent pregunta: **C++ o Blueprints?** Intentar buscar una resposta pot ser complicat, ja que a efectes pràctics els dos sistemes permeten implementar el mateix, i hi haurà desenvolupadors que defensaran l'ús de C++ i ho justificaran amb arguments vàlids, com que l'execució de codi és més ràpida que si s'utilitzen Blueprints, mentre altres justificaran que la diferència de rendiment és molt petita i que l'ús de Blueprints permet treballar de forma més senzilla al no necessitar coneixements de programació avançats.

Tot i que els dos sistemes permeten implementar tots els aspectes d'un videojoc, a nivell de funcionament intern del motor no han estat pensats com a sistemes independents dels quals escollir-ne un o altre segons les preferències de cada desenvolupador, sinó que han estat pensats per treballar junts, aprofitant les avantatges que ofereix cada sistema, de forma que enlloc de C++ o Blueprints s'hauria de parlar de **C++ i Blueprints**.

Les tasques de programació d'un videojoc es poden dividir en tres categories, ordenades de més baix nivell a més alt nivell:

- Programació del motor: És on s'implementen les funcionalitats de més baix nivell, com per exemple la gestió de la memòria, la representació de la geometria en un espai tridimensional, la gestió de la il·luminació, la connectivitat entre jugadors... Són funcionalitats que no depenen d'un joc en concret, sinó que podrien utilitzar-se en qualsevol projecte.
- Programació del joc: Formen part d'aquesta categoria totes les funcionalitats bàsiques per al funcionament concret del joc que s'està desenvolupant, com per exemple el moviment del jugador, les mecàniques bàsiques o qualsevol altra funcionalitat necessària per la implementació del joc.

- Programació d’scripts: Aquesta categoria utilitza els elements definits a la programació del joc per implementar com es veuen, es comporten i com permeten interactuar els diferents objectes del joc. Ajustar paràmetres com la velocitat del personatge, la vida o quin nivell es carrega a l’obrir una porta, serien exemples d’scripts.

Els motors de videojocs, en aquest cas Unreal Engine 4, implementen la part de programació del motor, i ofereixen als desenvolupadors una base provada i robusta que integra les principals característiques necessàries per a treballar en un videojoc. A partir d’aquesta base els desenvolupadors només s’han de preocupar per implementar la programació del joc i la programació d’scripts.

Unreal Engine 4 està pensat per a que la programació del joc es realitzi amb C++, i la programació d’scripts es realitzi amb Blueprints. L’objectiu al darrere d’aquesta metodologia és poder tenir un nucli del joc programat amb un llenguatge de programació compilat, que és més ràpid que el sistema de Blueprints, però sense renunciar a la facilitat que ofereixen les Blueprints a l’hora de poder ajustar paràmetres del joc, ja que no necessiten recompilar el projecte, i inclús es poden modificar en temps d’execució.

No existeix una línia fixa que determini quan s’ha d’utilitzar C++ o quan s’han d’utilitzar Blueprints, i cada equip de desenvolupament pot plantejar el disseny del videojoc de forma molt diferent. Una metodologia molt típica al treballar amb UE4 és realitzar un primer prototipatge ràpid amb Blueprints per després anar convertint les diferents funcionalitats a codi C++, actualitzant la línia que separa els dos sistemes de forma dinàmica a mesura que avança el desenvolupament del videojoc. A continuació es poden observar tres maneres diferents de dissenyar un projecte amb Unreal Engine 4.

La Figura 4.9 representa un projecte on es distribueix de forma equilibrada l’ús de C++ i Blueprints, implementant les funcionalitats referents a la programació del joc en C++ però utilitzant els elements que proporcionen les Blueprints.

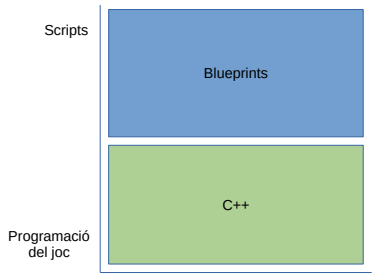


Figura 4.9: Implementació utilitzant de forma equilibrada Blueprints i C++

La Figura 4.10 representa un projecte basat principalment en Blueprints, on s'implementen tant les funcionalitats referents a la programació del joc com els aspectes específics amb Blueprints.

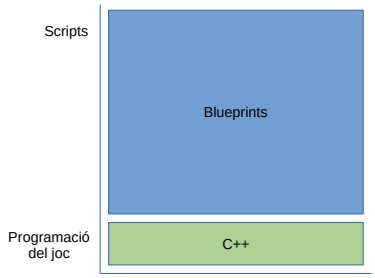


Figura 4.10: Implementació on s'utilitzen de forma més extensa les Blueprints

Finalment, la Figura 4.11 representa un projecte basat en C++, on s'implementen tant les funcionalitats referents a la programació del joc com els aspectes concrets dels diferents objectes amb C++.

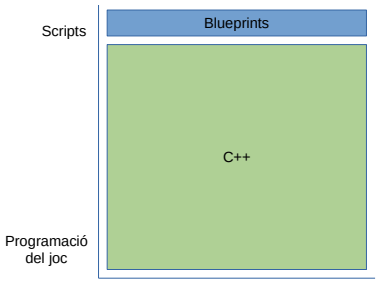


Figura 4.11: Implementació on s'utilitza de forma més extensa C++

Cal destacar que un mateix projecte pot estar format per diferents mòduls, i cadascun d'aquests mòduls pot tenir la seva pròpia distribució entre C++ i Blueprints. Per exemple, un mòdul de mecàniques pot tenir una distribució equilibrada entre C++ i Blueprints, un mòdul de funcions auxiliars pot estar implementat únicament en C++, o un mòdul on es

gestionin les interfícies d'usuari pot estar implementat totalment en Blueprints. La Figura 4.12 representa la distribució entre C++ i Blueprints de diferents mòduls que formen part d'un únic projecte.

Un cop feta aquesta breu introducció sobre la integració entre C++ i Blueprints des d'un punt de vista més teòric, es presentaran una serie d'exemples per il·lustrar de forma més pràctica els conceptes presentats, relacionant-los amb els avantatges i inconvenients de

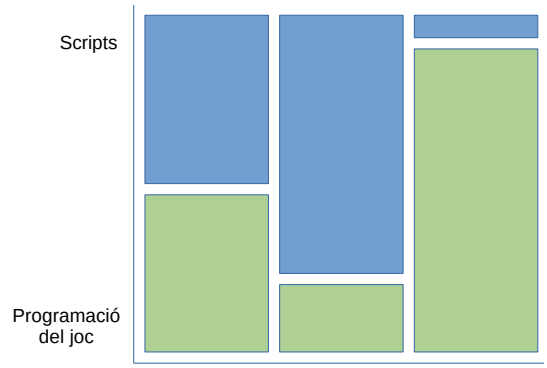


Figura 4.12: Distribució entre C++ i Blueprints en diferents mòduls d'un projecte

cada sistema i que s'han explicat als apartats anteriors.

Com exemple de integració de codi C++ amb Blueprints s'exposarà un sistema de gestió d'ítems que el jugador pot agafar i utilitzar. A la Figura 4.13 es pot observar la definició de la classe `Item`, amb un atribut que guarda el nom, un altre que guarda el model 3D de l'ítem i un mètode `utilitzar()` que implementa el que passa quan el jugador utilitza l'ítem.

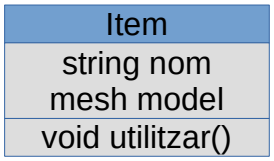


Figura 4.13: Definició de la classe Item

El següent fragment de codi implementa aquesta classe `Item` per tal de poder-la utilitzar des de les Blueprints.

```
class AItem : public AActor {
	GENERATED_BODY()

public:
	Aitem();

	UFUNCTION()
	void use();

private:
	UPROPERTY()
	FString name;

	UPROPERTY()
	UstaticMesh * model;
}
```

La Figura 4.4 integra la classe Item definida anteriorment a l'editor de Blueprints.

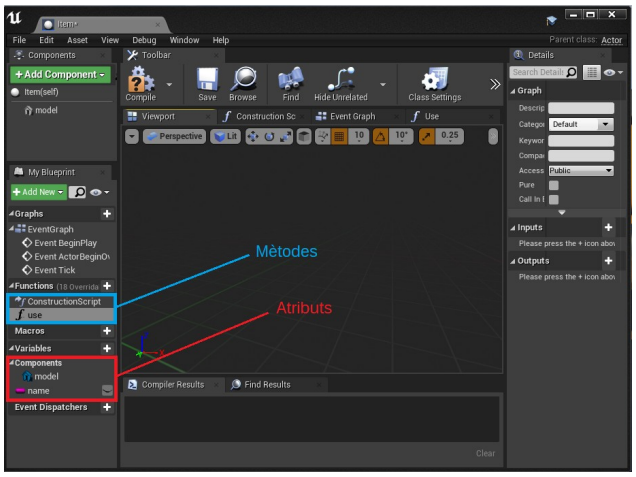


Figura 4.14: Implementació de la classe Item amb Blueprints

En cas de voler tenir diferents tipus d'ítems s'utilitzaria la herència per crear subclasses a partir de *`Item`*. La Figura 4.9 representa una relació d'herència entre la classe base *`Item`* i dos tipus diferents d'ítems.

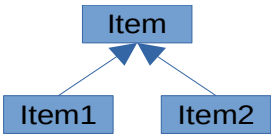


Figura 4.15: Subitems d'Item

S'ha mostrat com es pot implementar el mateix tant amb C++ com amb Blueprints, però aquesta solució encara no aprofita la idea d'integrar el C++ i les Blueprints. En el cas d'aquests atributs senzills com una cadena o un component del tipus *`StaticMesh`*, no hi ha cap motiu que justifiqui la utilització d'un dels dos sistemes per sobre de l'altre, però a l'hora de implementar el mètode *`utilitzar()`*, el C++ guanya pes respecte les

Blueprints, ja que si cal programar alguna lògica complexa seria més senzill i intuïtiu, a part de més eficient, fer-ho amb C++ que no pas amb els nodes de les Blueprints. Per altra banda, a l'hora d'assignar un model 3D a l'atribut que guarda la `StaticMesh`. Aquí hi ha un clar avantatge per les Blueprints, ja que com són *assets* poden agafar referències del contingut del projecte de forma dinàmica, sense necessitat de recompilar el projecte, mentre que en C++ caldria recompilar el projecte en cas d'haver de fer alguna modificació al model.

La solució òptima passaria per aprofitar els avantatges de cada un dels dos sistemes. La Figura 4.16 representa un diagrama de classes on la lògica de comportament dels ítems es defineix en C++ i els atributs específics de cada ítem, com pot ser el nom o el model, s'assignen des de les Blueprints. De forma que es podrà fer servir C++ per implementar la lògica complexa del mètode `utilitzar()` mantenint la flexibilitat que proporcionen les Blueprints a l'hora de gestionar els atributs.

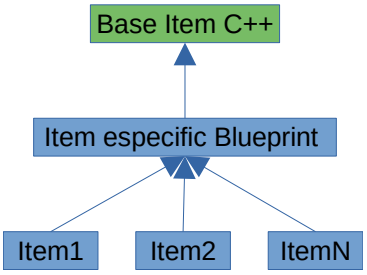


Figura 4.16: Diagrama de classes integrant C++ i Blueprints

4.2.4. Inputs

Unreal Engine 4 gestiona els inputs del jugador com a events que es disparen a mesura que el jugador realitza accions sobre els controls de qualsevol dispositiu. Tant des de C++ com des de les Blueprints es pot accedir directament als inputs dels dispositius més utilitzats, com el teclat, el ratolí o un gamepad, però això suposa tenir els controls vinculats a la implementació, fent més difícil que el jugador pugui personalitzar les tecles. Per solucionar-ho, el motor ofereix l'opció de mapejar els controls com a events personalitzats, diferenciant entre **accions (action mapping)**, que són inputs amb un valor digital (on – off) i **eixos (axis mapping)**, que són inputs amb un valor analògic.

Els mappings es defineixen a les opcions del projecte, a l'apartat d'Input. La Figura 4.17 mostra el menú on es creen i configuren aquest events personalitzats. Per cada input creat s'ha de definir el nom i quina tecla (o tecles) el dispararan.

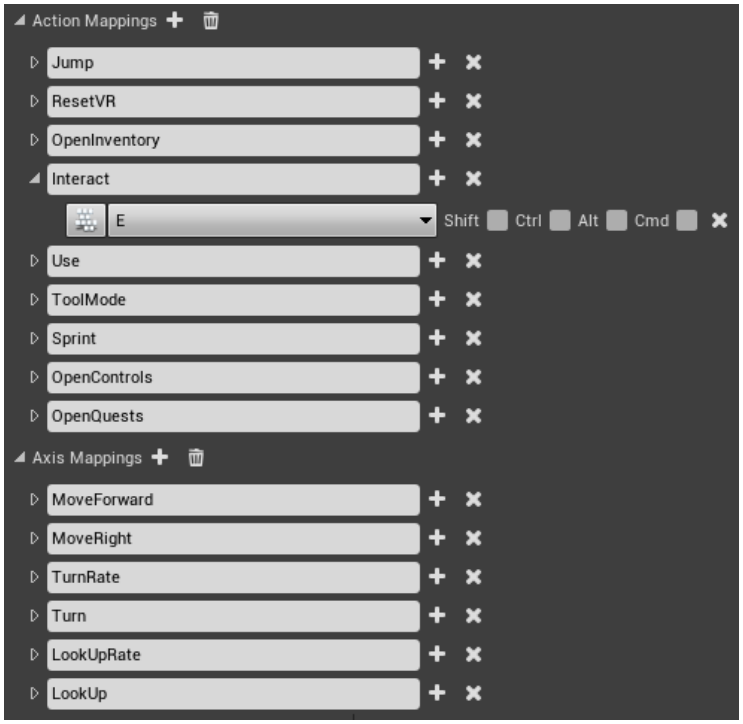


Figura 4.17: Mapping d'inputs

Implementar els controls a partir de mappings també permet fer jocs compatibles amb diferents plataformes de forma molt més ràpida, ja que la implementació és totalment independent de la gestió dels controls.

4.2.5. Programació d'UI

Unreal Engine 4 implementa una eina per treballar amb interfícies d'usuari anomenada UMG (Unreal Motion Graphics UI Designer), i que permet crear tant HUDs (Heads Up Display), que serien les interfícies que disposa el jugador mentre està jugant, com menús, pantalles de carrega o qualsevol altre element d'interfície que es vulgui presentar als jugadors.

Aquest sistema està basat en *Widgets,* que són una serie d'elements predefinits com botons, *checkboxes*, desplegables, barres de progrés, etc a partir de les quals es poden construir les interfícies d'usuari. Tots aquests widgets es poden gestionar a través d'un editor d'interfícies anomenat *Widget Blueprint*, que permet implementar tant la part visual

com la part funcional d'aquestes interfícies. Aquest editor disposa de dos modes de treball diferents:

- Designer Mode: Permet definir visualment les interfícies. La Figura 4.8 mostra els diferents elements d'aquest mode.

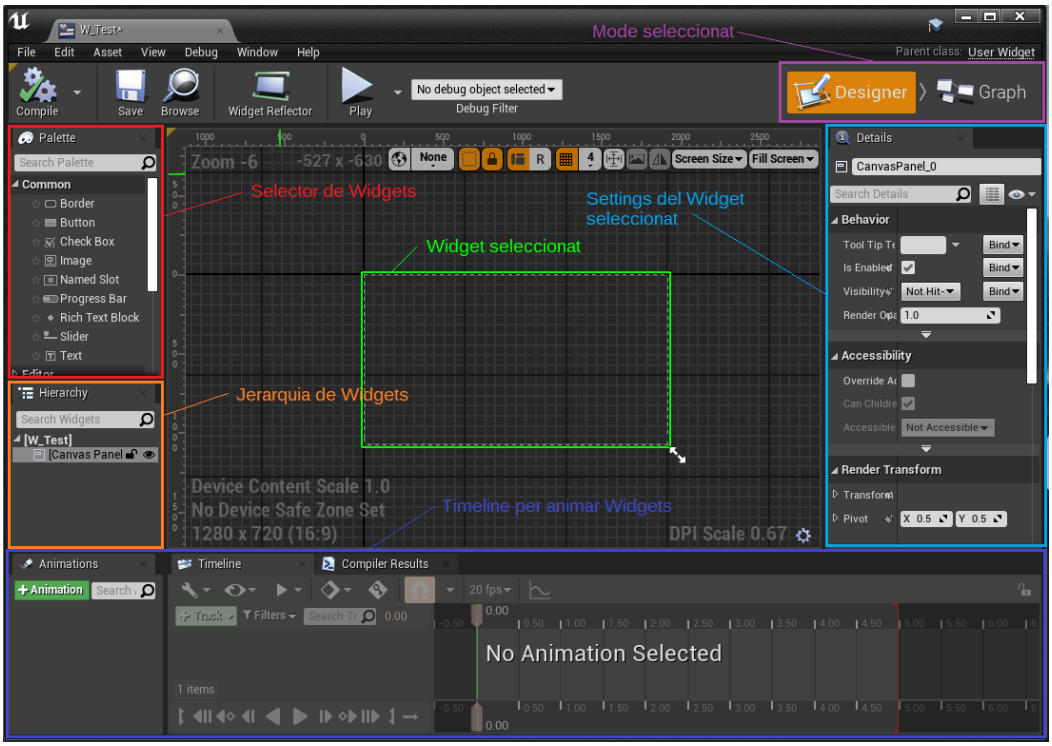


Figura 4.18: Interfície del mode dissenyador d'una Widget Blueprint

- Graph Mode: Permet definir la funcionalitat de les interfícies. Està basat en llenguatge visual de les Blueprints, tot i que inclou nodes addicionals per realitzar tasques específiques de les interfícies. La Figura 4.19 il·lustra els elements que es poden gestionar en aquest mode.

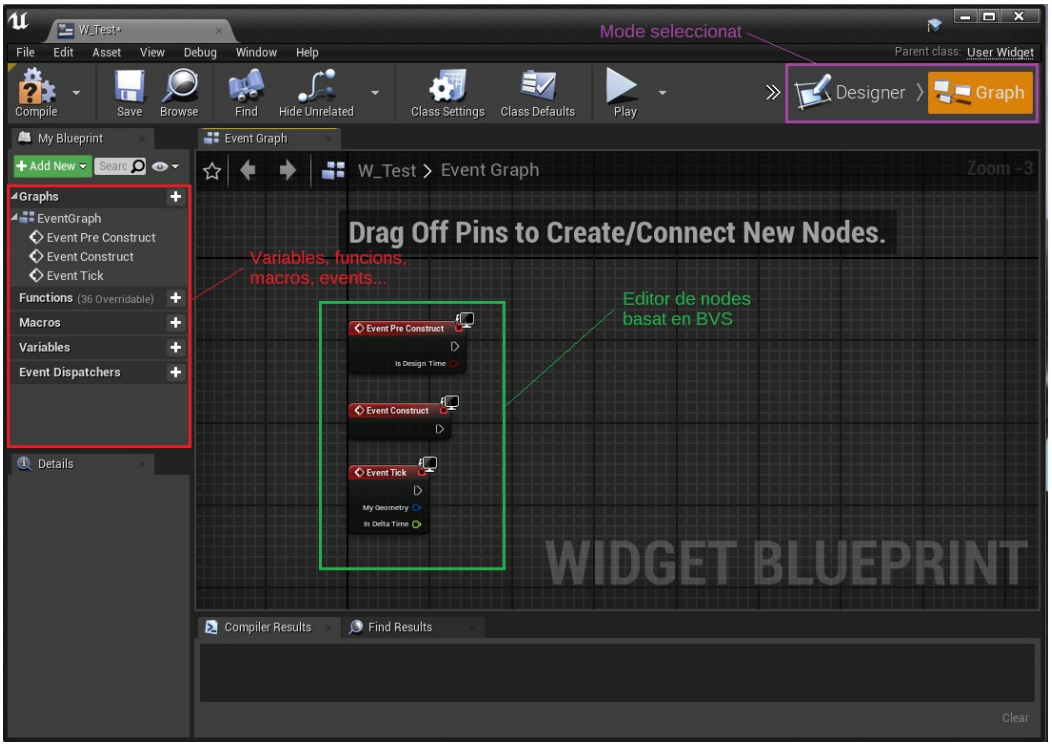


Figura 4.19: Interfície del mode graph d'una Widget Blueprint

La Figura 4.20 permet veure aquests dos modes en una interfície amb diferents widgets i múltiples funcions que s'ha implementat en aquest projecte.

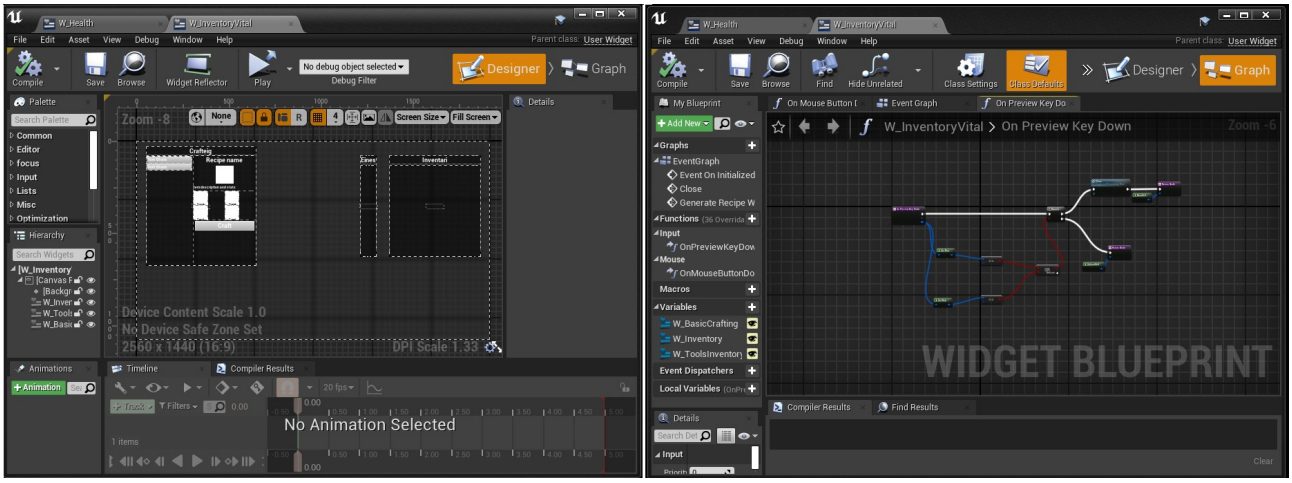


Figura 4.20: Comparativa entre el mode Designer i el mode Graph d'una interfície complexa

Per dissenyar interfícies de forma robusta i eficient s'ha de tenir en compte com s'actualitza la informació de la interfície i com es gestionen els inputs del jugador mentre la interfície es troba activada. Unreal Engine 4 disposa d'un sistema de *bindings* que permet definir de forma ràpida funcions que enllacen els widgets amb variables del joc.

Aquesta solució resulta molt senzilla, ja que automatitza la representació de valors interns del joc. El problema que té aquesta aproximació és que la forma en que el motor de UE4 la implementa no és gens eficient, ja que actualitza el valor l'interfície a cada frame. Quan aquestes variables canvien poc, el motor consulta moltes vegades el valor d'una variable per obtenir gairebé sempre el mateix resultat. Una millor pràctica seria implementar una funció que actualitzi el valor mostrat pel widget a partir d'uns paràmetres d'entrada, i cridar aquesta funció només quan s'hagi modificat el valor.

Per il·lustrar aquesta idea s'implementarà, de les dues formes presentades, una interfície senzilla per a visualitzar el valor d'una variable del jugador que s'actualitzarà cada vegada que aquest premi l'espai. La Figura 4.21 mostra la variable del jugador i com s'actualitza el valor, i a la Figura 4.22 es pot veure la interfície que s'ha creat.

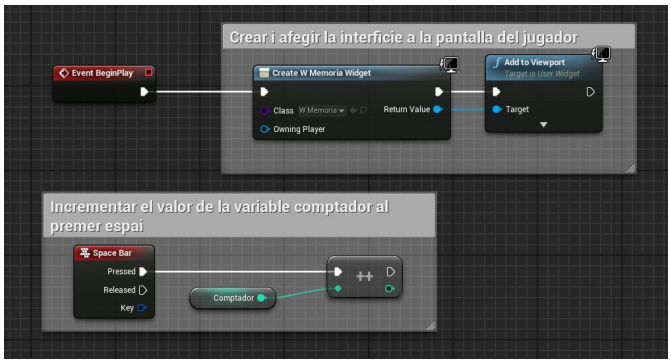


Figura 4.21: Codi del jugador que inicialitza la interfície i incrementa el comptador al premer espai

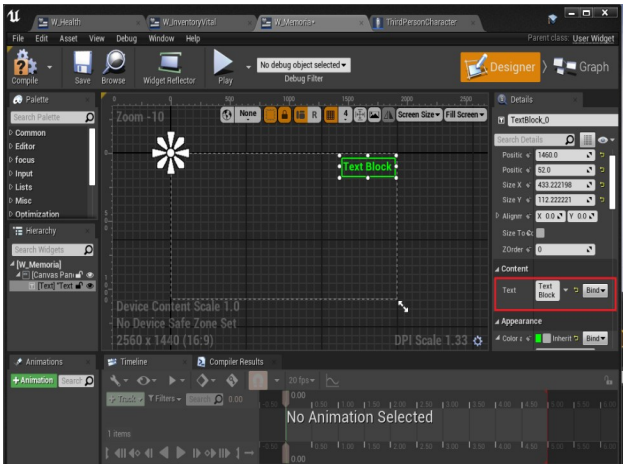


Figura 4.22: Interfície d'exemple

Per definir un bind s'ha de seleccionar el text i assignar-li la funció que s'encarregarà d'actualitzar el valor del widget. Al clicar al botó de bind marcat a la Figura 4.22 automàticament es crea una funció anomenada `Get_Text_0()` que retorna un valor del

tipus text. S'ha modificat aquesta funció per tal que abans de retornar el valor de la variable comptador mostri per pantalla un missatge indicant que s'ha actualitzat el widget. La Figura 4.23 mostra la implementació en Blueprints d'aquesta funció.

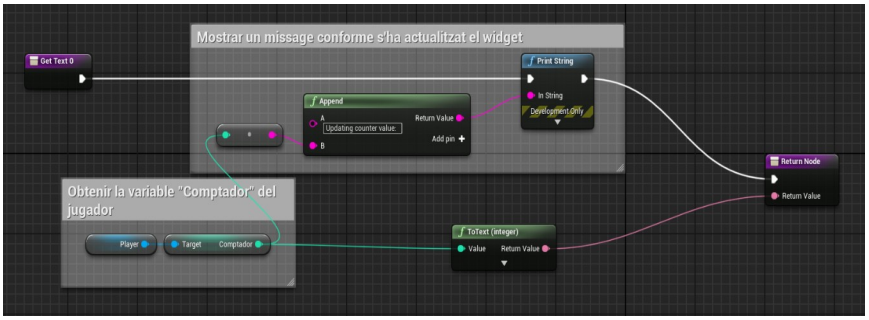


Figura 4.23: Implementació de la funció Get_Text_0

A l'iniciar el joc la interfície preguntarà al jugador pel valor del comptador una vegada cada frame, si el joc funciona, aproximadament, a 120FPS, el codi anterior s'executarà 120 vegades cada segon, encara que el valor del comptador sigui el mateix que al frame anterior. La Figura 4.24 mostra una captura de l'execució del joc on es pot veure a l'esquerra com no para de mostrar-se per pantalla el missatge que s'ha programat a la funció *`Get_Text_0()`*.

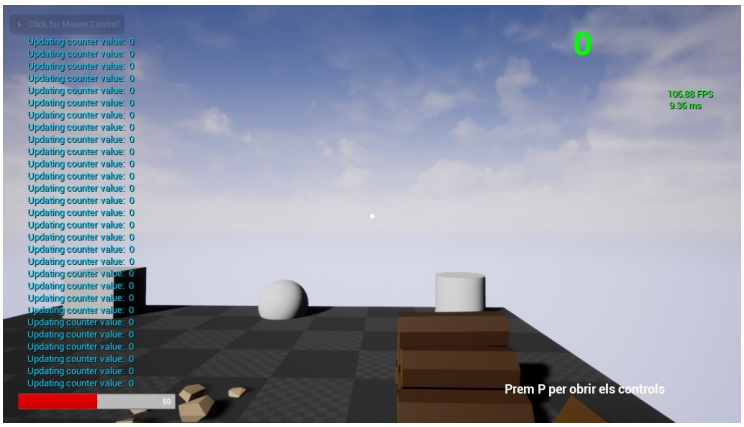


Figura 4.24: Execució del joc utilitzant bind

Per fer-ho de la forma correcta caldrà implementar una funció que actualitzi el valor de la interfície a partir d'un paràmetre amb el nou valor. Aquesta funció la cridarà el jugador cada cop que es modifiqui el valor del comptador, de manera que la interfície només s'actualitzarà quan realment hi hagi hagut un canvi. La Figura 4.25 mostra la funció *`actualitzarInterficie()`* implementada, i la Figura 4.26 mostra els canvis fets al jugador per a cridar aquesta funció al incrementar el comptador.

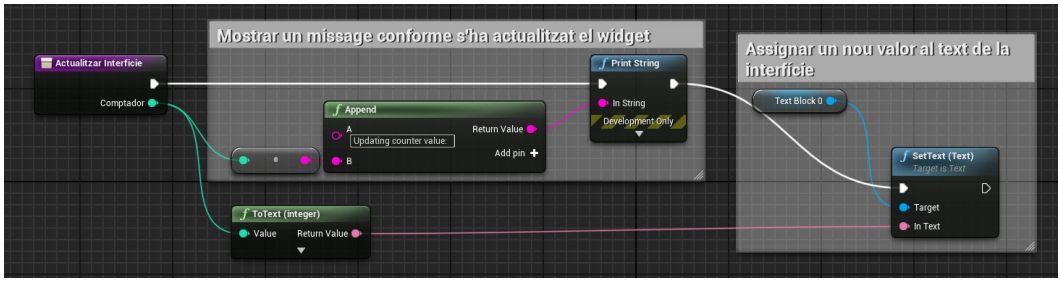


Figura 4.25: Implementació de la funció actualitzarInterficie

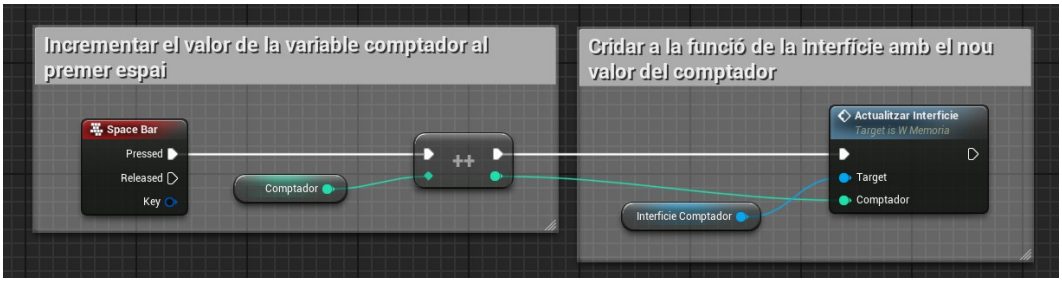


Figura 4.26: Modificacions al jugador per a cridar la funció actualitzarInterficie després de modificar el comptador

Ara, si s'executa el joc, el widget tindrà el valor inicial fins que el jugador premi l'espai, moment en què s'actualitzarà el comptador i s'executarà la funció per modificar la interfície. La Figura 4.27 mostra una captura de l'execució del joc on es pot veure com el nombre de missatges mostrats per la funció `actualitzarInterficie()` coincideix amb el valor actual del comptador.

Figura 4.27: Execució del joc utilitzant la funció update

4.2.6. Materials

Els materials són un tipus d'asset que es pot aplicar a un model 3D per controlar l'aspecte visual. A molt alt nivell es podrien considerar com la pintura que s'aplica als objectes, però en realitat ofereixen moltes més possibilitats que simplement modificar el color d'un

objecte, ja que a nivell intern del motor s'encarreguen de calcular com interacciona la il·luminació de l'escena amb els diferents objectes. Aquests càlculs es realitzen a partir de textures i d'expressions matemàtiques que es poden definir a través del Material Editor.

El *Material Editor* d'Unreal Engine 4 és una eina de programació de shaders basada en nodes, similars als de les Blueprints, però basats en el llenguatge de programació HLSL (High-Level Shading Language), que és un llenguatge pensat específicament per a implementar shaders. Els nodes disponibles en aquest editor s'anomenen *Material Expressions*, i en realitat són fragment de codi HLSL programats per a realitzar una funció específica, permetent als desenvolupadors implementar materials complexos de forma visual. El codi generat ha de ser compilat per poder visualitzar el material, i aquesta compilació s'ha de fer abans de començar el joc. La Figura 4.28 mostra algunes *Material Expressions* dintre de l'editor.

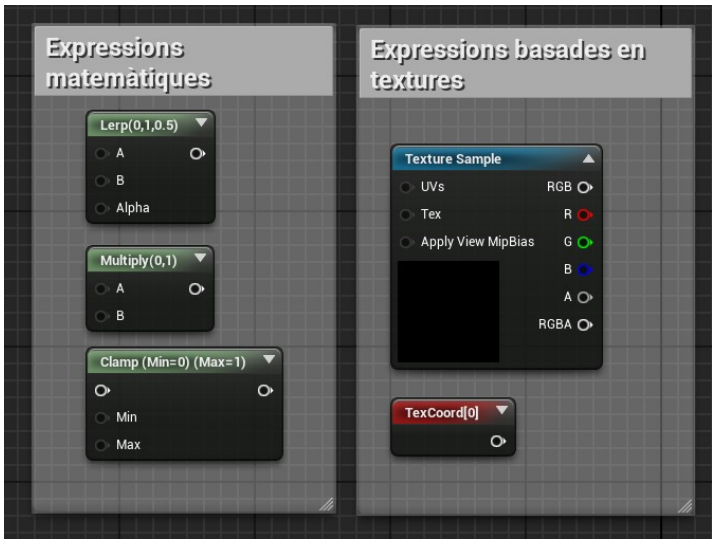


Figura 4.28: Nodes que implementen algunes Material Expressions dintre de l'editor

Per modificar un material, encara que només sigui l'aparença, fa falta recompilar-lo. Per solucionar aquesta limitació els materials suporten una característica anomenada *Material Instancing*, que permet establir una relació d'herència entre materials per poder canviar les propietats sense haver-los de recompilar. Això permet fer canvis ràpids sobre l'aparença dels materials o inclús modificar-los en temps d'execució del joc.

Per utilitzar *Material Instancing* s'ha de crear un material base amb els valors a modificar parametritzats. Al crear una instància d'aquest material, es podran modificar tots els paràmetres definits en temps real. A la Figura 4.29 es mostra la implementació d'un

material que emet un color a través de valors fixes dintre del codi, i a la Figura 4.30 es mostra la versió equivalent utilitzant paràmetres.

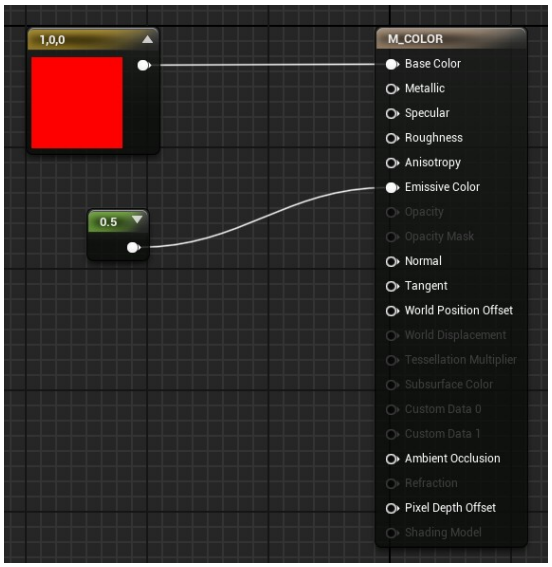


Figura 4.29: Material amb els valors de color i d'emissió definits dintre del shader

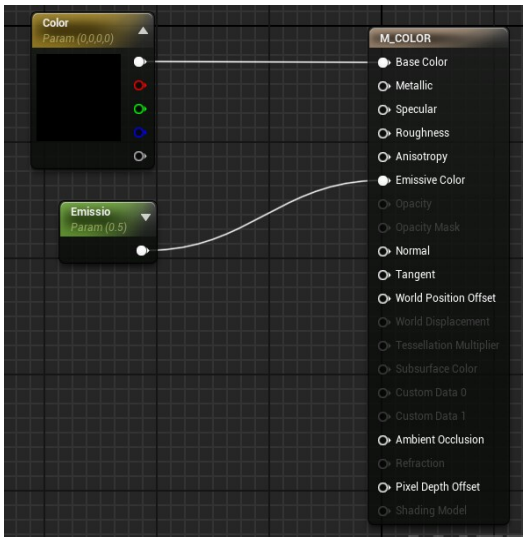


Figura 4.30: Material amb els valors de color i d'emissió del shader parametritzats

Si s'hagués de fer un canvi al material implementat a la Figura 4.29 caldria obrir l'editor de materials, modificar els paràmetres i tornar a compilar el shader per a canviar l'aspecte dels models que l'utilitzessin, mentre que el material implementat a la Figura 4.30 permetria crear una instància i modificar els paràmetres, com es pot veure marcat a la Figura 4.31.

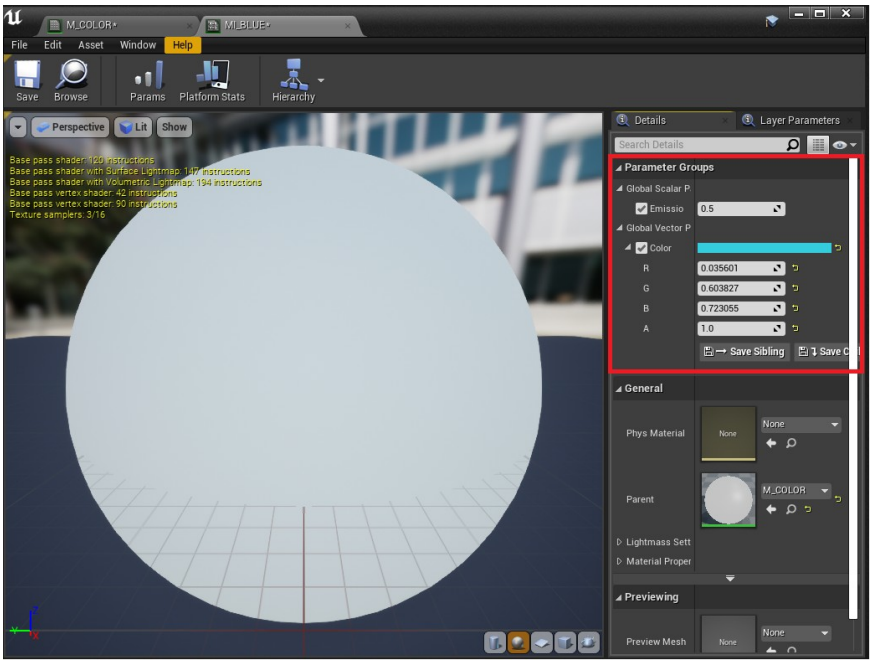


Figura 4.31: Panell de visualització de les material instances

4.2.7. Terreny

Unreal Engine 4 implementa una eina que permet generar terrenys i utilitzar-los com a base per a crear escenaris. Aquesta eina es pot utilitzar de tres maneres diferents:

- Manage Mode: Permet crear nous terrenys o modificar els paràmetres bàsics de terrenys ja existents. Els terrenys generats hereten de la classe Actor.
- Sculpt Mode: Permet modificar la geometria del terreny de diferents formes. Implementa des de modificacions bàsiques com aplanar o suavitzar fins a simulacions de l'erosió causada pel temps o per l'aigua.
- Paint Mode: Permet aplicar textures de forma manual a sobre del terreny. Inclou diferents tipus de pinzells i també permet importar pinzells propis. Cada textura que es vulgui aplicar s'ha de definir com una capa del terreny.

Els terrenys estan més optimitzats que els models 3D, fent-los ideals a l'hora de crear escenaris grans. En un terreny la informació de cada vèrtex s'emmagatzema en 4 bytes, mentre que en una `StaticMesh` s'emmagatzema en 28 bytes. Això suposa que a l'importar un escenari com a model 3D, ocuparà 6 o 7 vegades més memòria que si es genera amb l'eina de terrenys.

4.2.8. Animacions

A l'hora de treballar amb animacions, Unreal Engine 4 inclou un conjunt d'eines que permeten animar models. Aquestes eines permeten animar directament els esquelets amb un editor basat en keyframes similar als que es troben en molts programes de modelat 3D, però també permeten treballar amb animacions importades des d'aquests programes.

El sistema d'animacions gira al voltant de les *Animation Blueprints*, que són un tipus especial de Blueprints que permeten programar els controls de les animacions d'un `SkeletalMesh.` Aquestes Blueprints permeten fusionar animacions (animation blending), controlar els ossos d'un esquelet o implementar una lògica que defineixi quines animacions utilitzar en diferents situacions.

Els dos components principals de les *Animation Blueprints* són `EventGraph` i `AnimGraph,` que treballen junts per a determinar quina animació es reproduirà a cada frame del joc.

- EventGraph: Permet treballar amb events relacionats amb les animacions, com *`Begin`* o *`End`*. Actualitza els valors que s'utilitzen a la màquina d'estats d'animacions.
- AnimGraph: Defineix la màquina d'estats d'animacions i quines transicions hi ha entre animacions.

Les Figures 4.32 i 4.33 mostren aquests components implementats per al personatge del projecte.

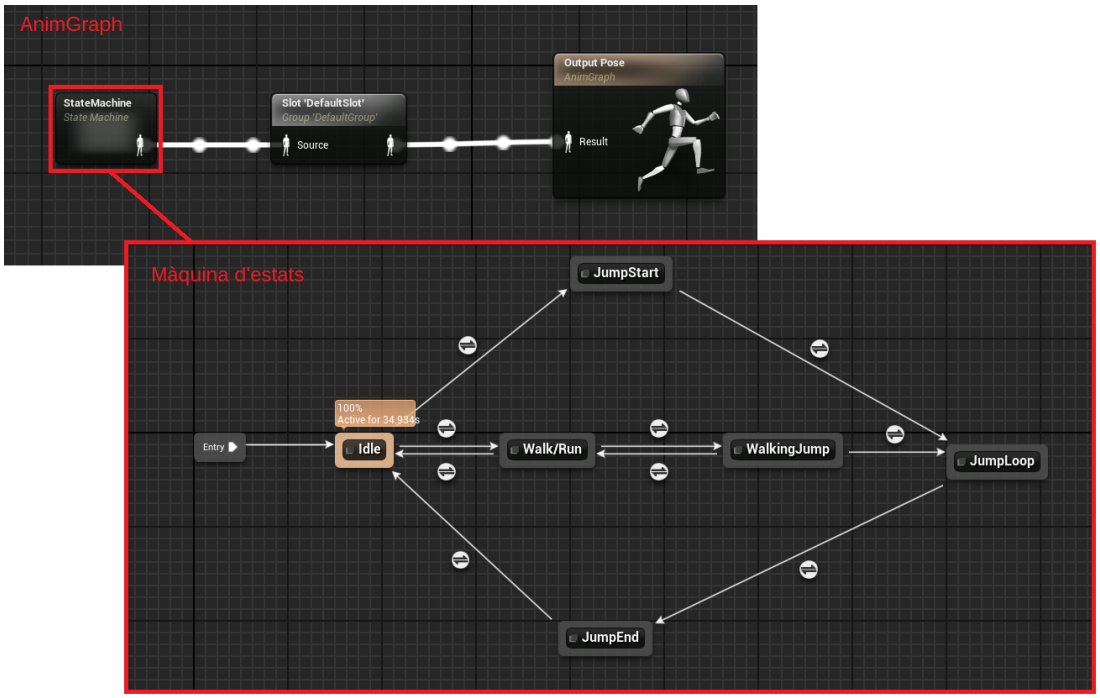


Figura 4.32: Exemple d'AnimGraph i màquina d'estats

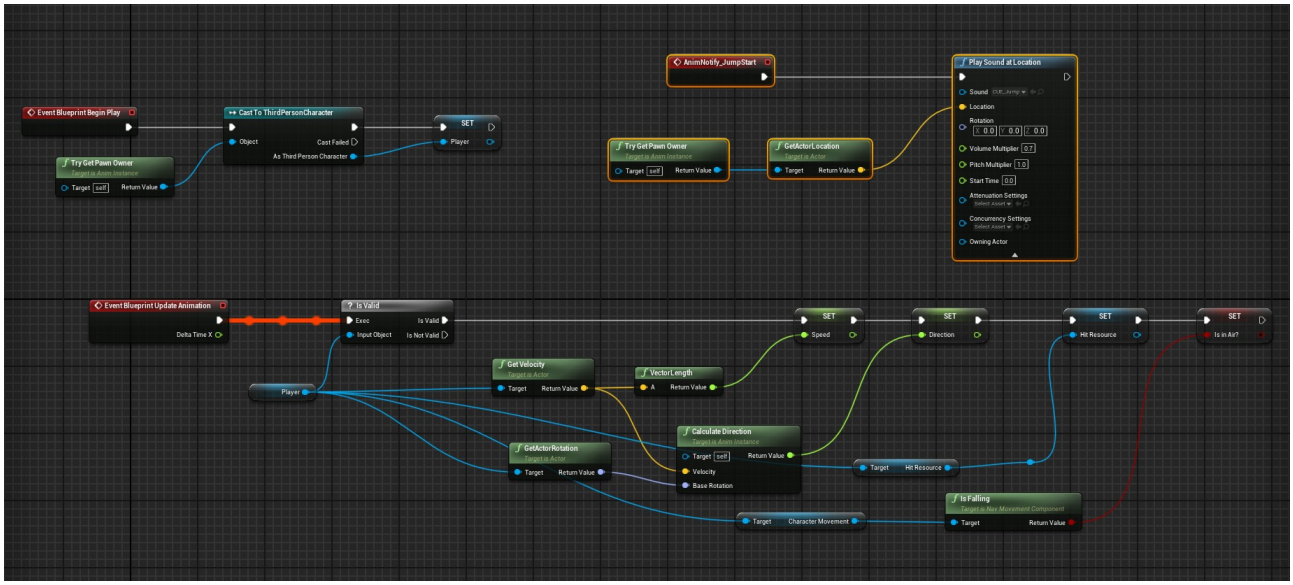


Figura 4.33: Exemple d'EventGraph que implementa l'event BeginPlay, UpdateAnimation i JumpStart

Analitzant la Figura 4.32 s'observa que hi ha un mateix estat per a caminar i córrer, tot i que són animacions diferents. Això s'ha realitzat a través de *Blend Spaces*, que són un tipus especial d'assets que permeten fusionar diferents animacions a partir dels valors de dos paràmetres. La Figura 4.34 mostra el *Blend Space* de caminar i córrer mencionat anteriorment. Al gràfic de la part inferior es pot veure com el valor de l'eix de les X equival a la velocitat, i les animacions estan representades pels dos punts blancs. Quan s'utilitza aquest *Blend Space* el motor interpola les dues animacions segons la velocitat del personatge.

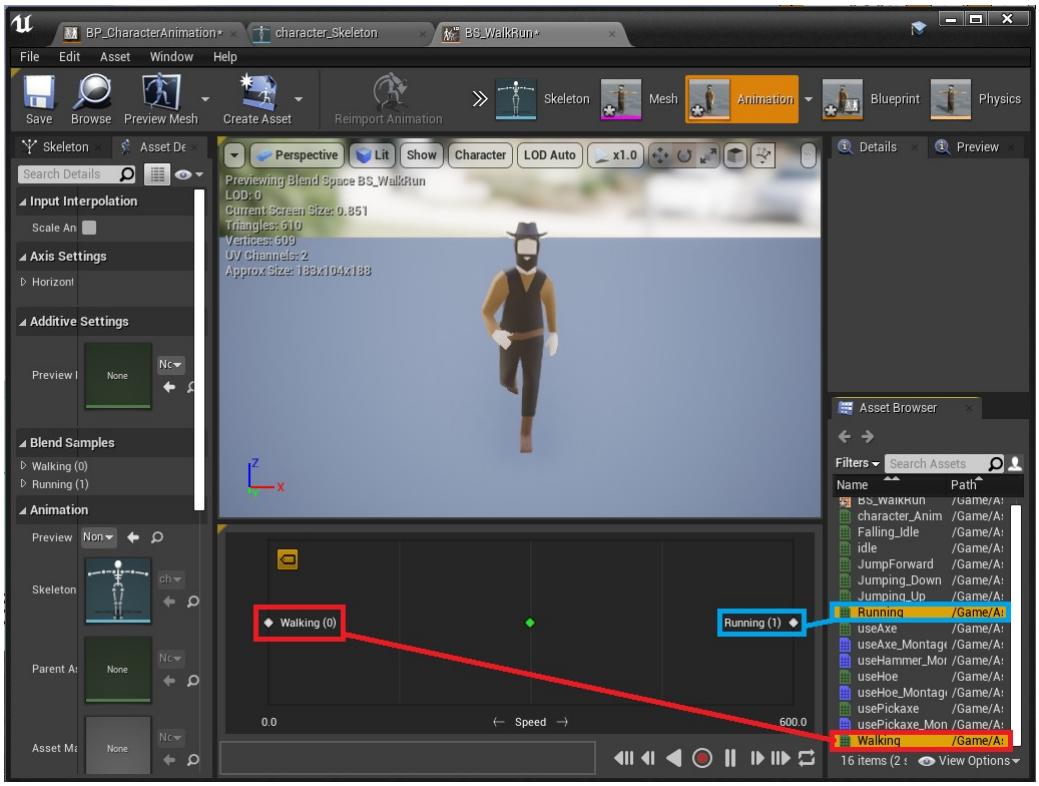


Figura 4.34: Blend Space caminar/córrer

Per acabar amb el bloc de les animacions s'explicarà el funcionament de les *Animation Notifications,* o com també es coneix dintre d'Unreal Engine 4, *Notifies,* que són una manera de disparar events personalitzats en moments específics d'una animació. Normalment s'utilitzen per a activar efectes de so o de partícules durant una animació. Aquests Notifies generen un event nou que es pot implementar a l'AnimGraph. La Figura 4.35 mostra una animació de caminar a la qual se li ha afegit un Notify anomenat `FootStep` cada cop que el personatge animat posa un peu a terra, i també mostra el codi associat a l'event.

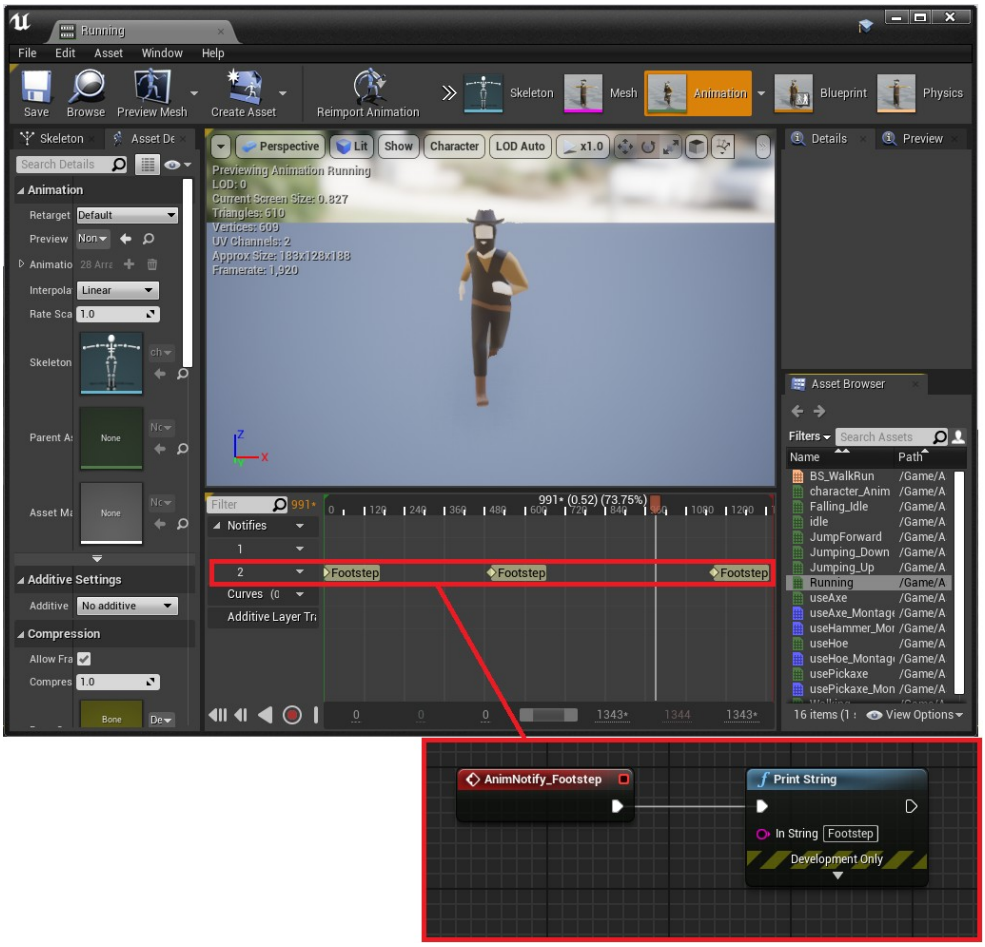


Figura 4.35: Animació amb diferents notifies tipus Footstep i el codi associa a aquests

Finalment a la Figura 4.36 es pot veure com a mesura que el jugador es mou es va executant el codi de l'esdeveniment.

Figura 4.36: Log del joc mentre el personatge camina per l'escenari

5. La vida a la Ribera d'Ebre durant el segle XX

La majoria de videojocs ambientats en el passat solen centrar-se en grans esdeveniments o imperis. Hi ha molts jocs de la 1ª o 2ª Guerra Mundial, jocs ambientats a l'antiga Roma o a Egipte, jocs medievals... Aquest projecte està ambientat en un passat proper, però en aquests cent anys que ens separen de l'època del joc la societat ha evolucionat moltíssim i de forma molt ràpida.

5.1. Ubicació

El joc passa a les Terres de l'Ebre, al sud de Catalunya. Aquesta regió està composta per les quatre comarques per on passa el riu Ebre en el seu tram final, des de l'entrada a Catalunya des de l'Aragó fins a la seva desembocadura al mar Mediterrani. D'aquesta regió s'ha escollit concretament la comarca de la Ribera d'Ebre. Aquesta comarca es la més septentrional de les quatre que formen el conjunt de les Terres de l'Ebre, i s'estén 40km a banda i banda de l'Ebre, des de la seva entrada a Catalunya per Riba-Roja fins a Miravet. Amb una superfície 827,3km^2, actualment està formada per 14 municipis als quals resideixen una quantitat de 21.647 habitants (2020), segons dades de l'Institut d'Estadística de Catalunya. La Figura 5.1 mostra la localització de la Ribera d'Ebre respecte la resta de Catalunya.

Figura 5.1: Localització de la Ribera d'Ebre

La geografia de la comarca destaca per quatre grans accidents geogràfics: el pas de l'Ase, a l'entrada del riu a Catalunya, la cubeta de Móra, que ocupa bona part del centre de la comarca, la serra de Tivissa, que s'estén fins al mar, i la serra de Cardó, al sud partint amb la comarca del Baix Ebre. D'aquestes quatre zones s'ha decidit ambientar el joc a la serra de Cardó, ja que és la regió de procedència de l'equip de desenvolupament.

La serra de Cardó s'estén abruptament entre els municipis de Rasquera, a la Ribera d'Ebre, i Benifallet, al Baixe Ebre. Formada per roques calcàries, destaca per una gran varietat de flora i de fauna. Els boscos són de pins, carrasques i teixos, i abunda el margalló, matèria primera per la producció de la llata amb que es fan cabassos i molts altres atuells de pauma. Pel que fa a la fauna, es poden trobar espècies com el jabalí, la rabosa o el fardatxo. A la Figura 5.2 es pot observar la serra de Cardó vista des de la Creu de Santos.

Figura 5.2: Serra de Cardó

5.2. Activitat econòmica

L'activitat econòmica principal era l'agricultura i la ramaderia, i variava segons la ubicació dels diferents municipis, veient-se influenciada de forma directa per la proximitat al riu. Els municipis amb accés a l'aigua podien treballar cultius de regadiu, com els fruiters o els cítrics, mentre que els municipis més llunyans treballaven cultius de secà com l'ametlla o l'olivera. A les zones de muntanya també destacaven les tasques de recol·lecció de recursos dels boscos, com la resina i l'escorça dels arbres, anomenada escarrotxa, o el margalló.

L'any 1898 comença la industrialització de la comarca amb la construcció de la Electroquímica de Flix, i continua durant la segona meitat del segle XX amb la construcció

de les centrals hidroelèctriques de Flix i Riba-Roja i la central nuclear d'Ascó. Aquesta industrialització va comportar un augment de la població i un desplaçament de la feina del camp cap a les feines de construcció d'aquestes infraestructures.

5.3. Vincle entre la història i el joc

Les tècniques que s'han integrat en el joc han estat les de recol·lecció de recursos, obtenció de fusta i pedra, construcció i creació de camps de conreu. Per a poder implementar aquestes tècniques s'han fet servir eines com la destral, el pic, l'aixada i el martell.

Pel que fa als ítems del joc tots estan basats en elements característics de la regió, encara que s'hagin hagut d'adaptar per a poder ser representats a través dels models 3D disponibles. Les zones de producció implementades representen oficis típics de l'època, com el de fuster o ferrer.

6. Disseny del videojoc

6.1. Narrativa

El joc passa en un poble de la Ribera d'Ebre durant la primera meitat del segle XX. El jugador pren el control d'un personatge jove que ha d'adaptar-se a l'estil de vida de l'època per poder sobreviure. El personatge haurà d'aprendre les tècniques s'utilitzaven per a poder obtenir recursos que l'ajudaran durant l'aventura.

Al ser un joc de món obert s'ha optat per no fer una història tradicional on el jugador hagi d'anar completant missions que li permetin avançar en la trama, sinó que s'ha plantejat una història integrada dintre del joc. Les diferents interfícies donaran al jugador informació sobre els objectes i com s'utilitzaven en l'època on s'ambienta el joc. Per a ajudar a guiar al jugador s'implementarà un sistema d'objectius que anirà introduint les diferents mecàniques.

6.2. Personatges

Aquesta versió del joc només tindrà un personatge, que serà el controlat pel jugador. Inicialment es va plantejar un disseny del personatge que transmetés la forma de vestir de l'època on s'ambienta el joc, però donada la poca experiència de l'equip modelant personatges s'ha decidit utilitzar un model extern. Al treballar amb una estètica low-poly no es pot donar gaire nivell de detall, ja que causaria incoherències amb la resta de l'escenari.

La Figura 6.1 mostra l'aspecte físic del personatge, que serà representat per un home adult d'entre 20 i 30 anys, i no disposarà de cap opció de personalització per part del jugador. No tindrà expressió facial, ja que com s'ha indicat anteriorment crear un personatge molt detallat no encaixaria amb l'estètica de la resta d'elements del joc. Anirà vestit amb roba de treball i un barret.

Psicològicament no tindrà uns trets característics, ja que l'objectiu del joc és que el personatge sigui un contenidor on el jugador pugui posar la seva pròpia personalitat.

Figura 6.1: Aspecte físic del personatge

6.3. Espai i escenaris

Al tractar-se d'un joc de món obert tota l'acció passa en un escenari gran. La Figura 6.2 mostra l'espai complet del joc.

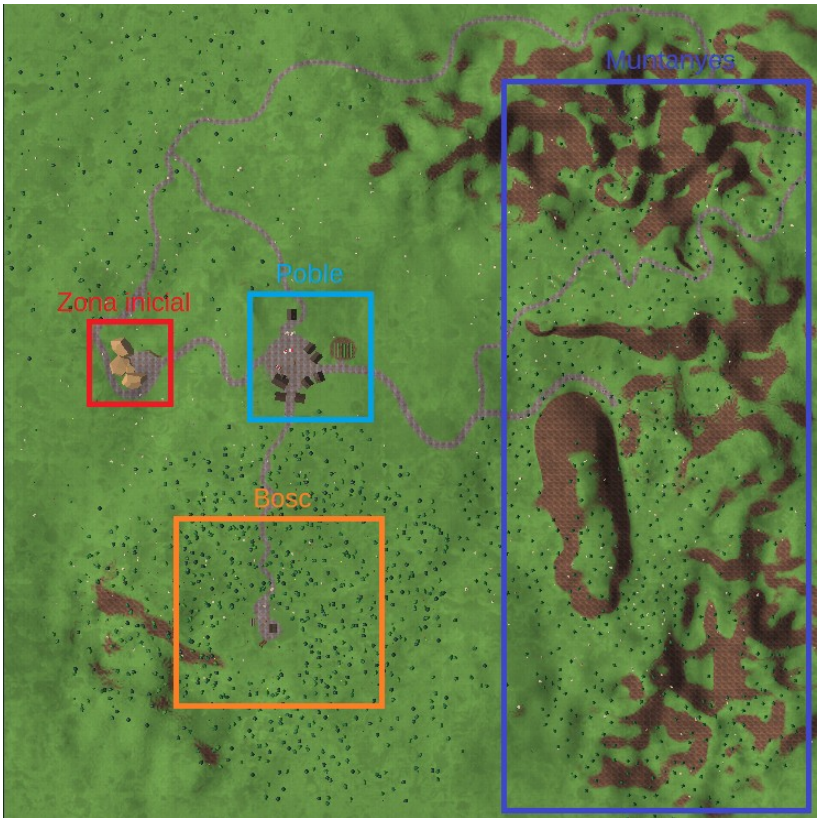
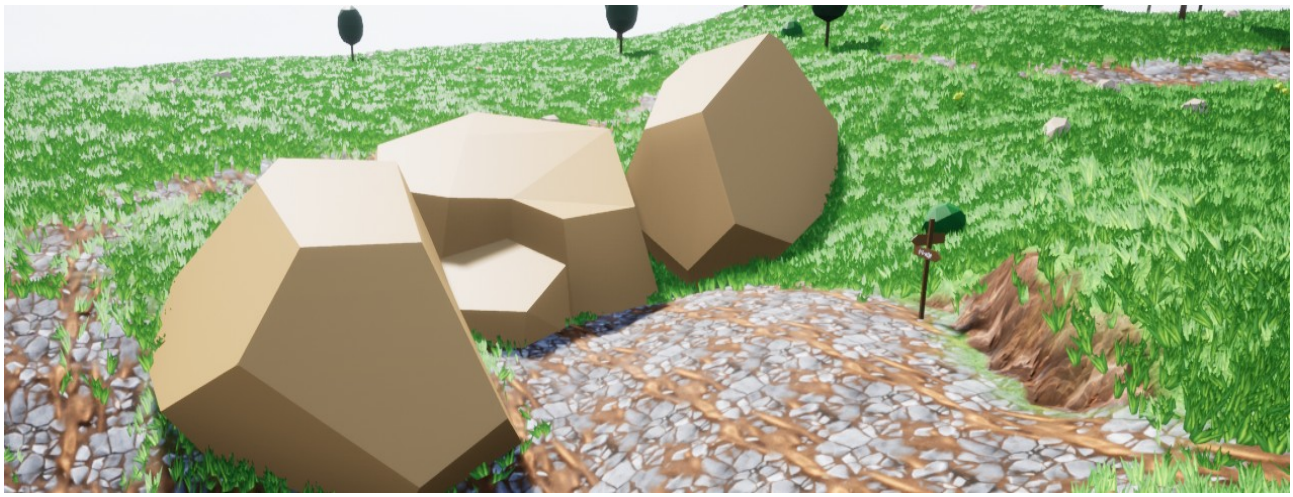


Figura 6.2: Espai del joc

Aquest espai s'ha dividit en diferents zones pensades per a facilitar al jugador la recol·lecció de recursos importants.

- Zona inicial: Situa al jugador al mapa mitjançant camins i cartells.



- Poble: Representa un poble tradicional de la Ribera d'Ebre

- Bosc: Zona rica en fusta

- Muntanyes: Zona rica en pedra

6.4. Mecàniques

Com ja s'ha mencionat anteriorment el videojoc a desenvolupar pertany al gènere dels videojocs de supervivència. Abans de definir les mecàniques concretes d'aquest projecte es farà una breu introducció a les mecàniques més típiques d'aquest gènere.

- Recol·lecció: El jugador té la capacitat de poder agafar elements de l'escenari i guardar-los en un inventari. Aquesta mecànica es pot implementar de moltes formes diferents, però la majoria de jocs opten per alguna d'aquestes dos solucions:
    - Recol·lecció automàtica: Quan el jugador passa per sobre d'algun dels ítems que es poden recollir el joc l'afegeix automàticament a l'inventari. Aquesta implementació permet tenir un gameplay més àgil a canvi de reduir el realisme.

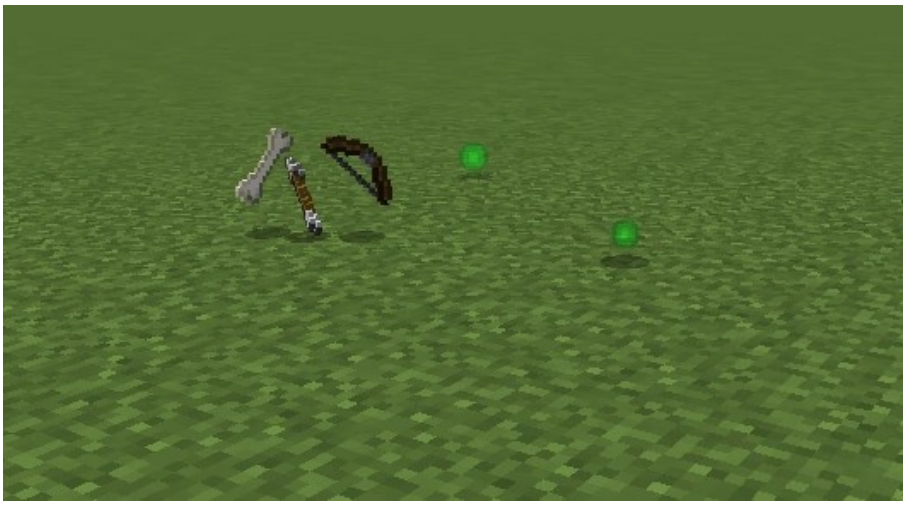


Figura 6.3: Minecraft implementa un sistema de recol·lecció automàtica

- Recol·lecció manual: El jugador ha de prémer un botó per a poder recollir els ítems. Aquesta implementació dona més control al jugador, ja que pot escollir quins ítems agafar de forma manual, però pot resultar en un gameplay més lent.

Figura 6.4: Valheim implementa un sistema de recol·lecció manual

- Crafteig: Permet combinar diferents ítems per a obtenir-ne de nous. Normalment es combinen ítems que es poden recol·lectar de l'escenari per a convertir-los en recursos que no es poden aconseguir de l'escenari. Aquesta mecànica es sol implementar en dos parts, una més senzilla i a la que el jugador pot accedir en qualsevol moment, i una de més avançada que requereix que el jugador hagi d'utilitzar algun element extern, com una taula de crafteig. Algunes implementacions d'aquesta mecànica en títols reals són:
  - *The Souls Project*: El jugador deixa caure a terra els ítems bàsics i si la combinació es correcta es canvien per l'ítem nou.
  - *Stranded Deep*: El jugador deixa caure els ítems bàsics a terra, i segons l'eina que utilitza el jugador s'obre un menú amb els objectes que es poden craftejar.
  - *Metro Exodus*: El jugador porta una motxilla que pot utilitzar en tot moment per a craftejar ítems bàsics com municions o paquets de medicines. A més, al llarg dels escenaris hi ha unes taules de treball que permeten crear coses més avançades com millores a l'armadura o a les armes.
  - *Minecraft*: Aquest sistema és diferent a la resta, ja que utilitza unes matrius on s'han de col·locar els ítems correctes en ordre. El jugador té accés en tot moment a una matriu de 2x2, però si interactua amb una taula de crafteig pot utilitzar una matriu de 3x3.

- Construcció: Es una mecànica similar al crafteig, però enlloc de crear ítems nous el jugador té la capacitat de construir objectes dintre de l'escenari. La profunditat del sistema depèn de la implementació, havent-hi jocs que permeten construir de forma lliure al jugador, fomentant la creativitat, o jocs que limiten el sistema permetent només construir coses predeterminades.
- Pesca: Com el seu nom indica permet al jugador pescar recursos de l'aigua. La mecànica en si és senzilla, però la seva complexitat rau en la quantitat d'ítems diferents que es poden pescar.
- Combat: Hi ha jocs d'aquest gènere que afegeixen una dificultat extra a l'hora de sobreviure afegint enemics que ataquen al jugador. La mecànica de combat pot ser més o menys complexa, segons el nombre d'enemics i armes que es troba el jugador. Els jocs que implementen aquesta mecànica solen incorporar una opció per jugar en mode pacífic (sense enemics), pensada per als jugadors que busquen només l'experiència de sobreviure a l'entorn.
- Agricultura: Al començament d'un joc de supervivència és entretingut haver d'explorar per a aconseguir menjar i recursos inicials, però arriba un punt en que haver de buscar menjar es comença a tornar repetitiu. Aquí entra en joc la mecànica de l'agricultura, permetent al jugador establir granges per a disposar d'una font fiable d'aliments. Hi ha jocs que implementen sistemes més complexos on s'ha de plantar, regar i abonar les plantes per a produir recursos, i n'hi ha que implementen un únic cultiu que creix automàticament.

A partir d'aquestes característiques descrites es pot fer una descripció més detallada de les mecàniques que formaran el projecte, que seran **recol·lecció, crafteig, construcció i agricultura.** S'ha descartat implementar la mecànica de pesca, ja que tot i estar a prop del riu Ebre no era una activitat econòmica típica, i la mecànica de combat, ja que el joc busca ensenyar la vida de l'època de la forma més fidel possible.

6.4.1. Mecàniques bàsiques

Abans d'explicar el funcionament de les mecàniques típiques del gènere s'han de definir les mecàniques bàsiques a sobre de les quals s'implementaran les més específiques.

Moviment del jugador

El jugador podrà moure's lliurement per l'escenari utilitzant una vista en primera persona. S'implementarà el moviment típic dels jocs que utilitzen aquesta vista, que consisteix en poder controlar el moviment del jugador amb les tecles W (endavant) A (esquerra) D (dreta) S (endarrere), i el ratolí, que controlà el moviment de la càmera. A la Figura 6.5 es pot observar un esquema del funcionament dels controls.

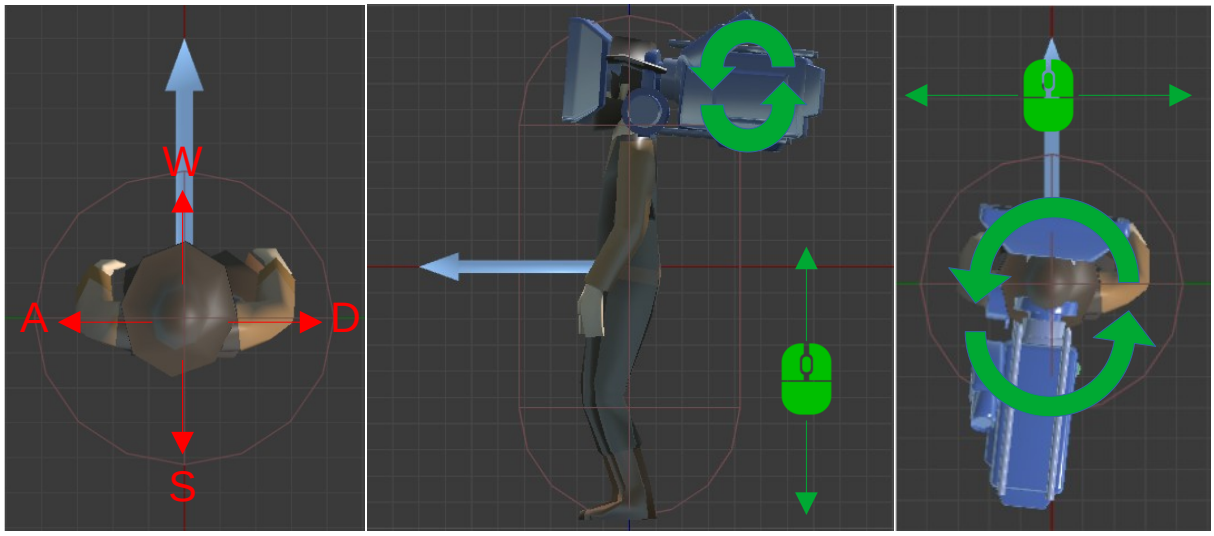


Figura 6.5: Esquema dels controls i com interactuen amb el personatge

Inventari

Per poder implementar les mecàniques més complexes fa falta un sistema d'inventari on el jugador pugui guardar ítems. Per poder dissenyar un inventari fa falta saber un parell de conceptes provinents de l'anglès que s'utilitzen per definir les característiques del sistema:

- **Slot**: Posició o ranura de l'inventari. Un inventari pot tenir un nombre màxim de slots, obligant al jugador a gestionar el seu inventari, ja que mai podrà portar més ítems que nombres de slots, o pot tenir una mida infinita, permetent que el jugador pugui tenir tants ítems com siguin necessaris a l'inventari.
- **Stack**: Pila d'ítems. Per permetre que el jugador pugui portar més ítems es pot implementar un sistema que permeti apilar un tipus d'ítems en un slot, per exemple, el jugador pot tenir una ranura de l'inventari amb fusta i cada cop que agafi més fustes enlloc d'ocupar ranures noves s'apilaran a la ranura que ja té el recurs. Alguns jocs utilitzen stacks d'una mida predefinida, per exemple, de 50 unitats, de forma que cada posició de l'inventari pot guardar com a molt 50 unitats d'un ítem.

Hi ha moltes formes d'implementar un sistema d'inventari, però per aquest projecte s'ha optat per fer un inventari infinit on es guardin stacks d'ítems. Al treballar amb un inventari amb slots infinits s'ha optat per a no limitar artificialment la mida dels stacks, permetent que a cada ranura de l'inventari es pugui guardar una quantitat màxima d'ítems equivalent al nombre màxim que pot representar un enter (2.147.483.647).

6.4.2. Recol·lecció

La mecànica de recol·lecció és una de les que té un paper més important dintre del joc, ja que és la primera que aprendrà el jugador i s'utilitzarà durant tota la partida. Abans s'ha mencionat dues formes d'implementar aquesta mecànica, recol·lecció automàtica i recol·lecció manual. Per aquest projecte s'ha optat per un sistema de recol·lecció manual, ja que ofereix una visió més realista de com seria agafar una cosa a la vida real, a més a més de donar més control al jugador sobre que vol agafar. Aquesta mecànica està relacionada amb l'inventari mencionat anteriorment, ja que els ítems que agafi el jugador es guardaran a l'inventari.

A base de recol·lectar recursos el jugador podrà crear eines que li permetran obtenir alguns ítems de forma més ràpida i simple, per exemple, al començament de la partida el jugador només podrà agafar troncs de terra, però a quan tingui suficients recursos per crear una destral podrà tallar arbres i obtenir fusta.

6.4.3. Crafteig

La mecànica de crafteig estableix les bases necessàries per a poder obtenir recursos més avançats que no es poden recol·lectar de l'escenari. Permet al jugador intercanviar ítems de l'inventari per alguns de nous. Aquesta mecànica està basada en receptes, que no són res més que relacions entre ítems, com per exemple, 2 fustes i 2 pedres donen 1 martell. Per afegir més profunditat al joc s'han definit diferents categories de receptes que podran tenir associada una zona de crafteig, de forma que el jugador podrà craftejar diferents receptes interactuant amb diferents zones.

S'han creat diferents categories per simular les diferents feines tradicionals de l'època on s'ambienta el joc, ja que no és el mateix haver de construir una eina a partir de ferro que haver d'elaborar vi a partir de raïm, i no seria realista tenir una única zona on es poguessin fer totes les conversions juntes. A part d'aquestes zones el jugador podrà craftejar alguns ítems bàsics en tot moment.

6.4.4. Construcció

La mecànica de construcció permet que el jugador pugui col·locar edificis o eines a l'escenari. La principal funció d'aquesta mecànica és la de crear les zones de crafteig mencionades anteriorment.

Aquest sistema està basat en la mecànica de crafteig, però, enlloc de convertir ítems en nous recursos permet canviar-los per construccions que es poden col·locar a l'escenari. Per accedir a aquesta mecànica el jugador haurà d'utilitzar el martell, que és una eina que caldrà haver craftejat primer, i que serveix per a col·locar els edificis a l'escenari.

6.4.5. Agricultura

La mecànica d'agricultura permet que el jugador pugui establir una font fiable d'aliments per poder anar progressant en el joc sense haver d'estar sempre depenent de la recol·lecció. En el joc plantejat aquesta mecànica s'utilitzarà també per introduir els cultius tradicionals de la regió, i també permetrà obtenir algunes matèries primeres necessàries per algunes de les tècniques que s'explicaran.

Per a activar aquesta mecànica el jugador haurà de crear una aixada per poder definir àrees de cultiu a l'escenari. En aquestes àrees el jugador podrà plantar llavors que al cap d'un temps es convertiran en ítems que es poden recol·lectar.

6.5. Objectes del joc

Els objectes del joc són tots els elements amb els quals el jugador pot interactuar. Dintre del joc existeixen quatre tipus d'objectes diferents: **ítems**, que són objectes que el jugador pot recol·lectar o craftejar, **eines**, que són objectes que el jugador ha de craftejar i permeten realitzar noves accions, **zones de producció**, que són objectes que permeten al jugador desbloquejar noves receptes de crafteig, i **recursos**, que són objectes que a l'interactuar amb les eines correctes donen materials.

6.5.1. Ítems

Són els objectes més bàsics del joc. Es poden obtenir a través de la recol·lecció o del crafteig. Els ítems es divideixen en dos categories:

Aliments: Són un tipus d'ítem que el jugador pot menjar-se per a recuperar vida.

- Massana
- Bolets
- Pastanaga
- Panís
- Mel
- Carbassa
- Carn

Materials: Són un tipus d'ítem que serveixen per a craftejar o construir.

- Pedra
- Fusta
- Fibres vegetals
- Resina
- Argila

6.5.2. Eines

Serveixen per a incrementar les accions que pot fer el jugador. S'han de craftejar a partir d'ítems.

- Destral: Permet tallar arbres i arbustos
- Pic: Permet trencar pedres grans
- Aixada: Permet definir zones per a plantar
- Martell: Permet construir

6.5.3. Zones de producció

Desbloquegen noves receptes de crafteig. S'han de construir amb el martell a partir d'ítems.

- Enclusa: Desbloqueja receptes relacionades amb la metal·lúrgia
- Caldera: Desbloqueja receptes relacionades amb els aliments

- Banc de treball: Desbloqueja receptes relacionades amb la fusta

6.5.4. Recursos

Si el jugador interactua amb l'eina correcta es converteixen en ítems.

- Arbres: Donen fusta i resina
- Roques: Donen pedra

6.6. Objectius

Com s'ha explicat anteriorment el joc tindrà un sistema d'objectius que seran petites missions que el jugador ha de complir. Aquestes missions no tindran un pes narratiu, sinó que serviran com a guia de les mecàniques del joc per evitar que el jugador es quedi encallat en algun punt de la partida.

Cada missió serà un conjunt d'objectius més senzills. Hi haurà cinc tipus d'objectius:

- Arribar a una destinació
- Agafar un ítem
- Craftejar
- Construir

Les missions del joc seran les següents:

- Explorar el mapa
  - Anar al poble
  - Anar a les muntanyes
  - Anar al bosc
- Recollir ítems bàsics
  - Recollir 10 Fustes
  - Recollir 10 Pedres
  - Recollir 5 Fibres Vegetals
- Crear les eines bàsiques

    - Crear una destral
    - Crear un pic
    - Crear una aixada
    - Crear un martell
- Construir les zones de producció
    - Construir una caldera
    - Construir una enclusa
    - Construir un banc de treball
- Construir una casa

El jugador podrà completar aquests objectius en qualsevol ordre, i quan els completi tots s'acabarà el joc.

6.7. Economia del joc

L'economia del joc estarà formada per dos recursos: vida i ítems. La vida serà el recurs més important, ja que perdre tota la vida suposarà acabar el joc.

Al definir l'economia d'un joc s'ha d'especificar com interactuaran aquests recursos entre ells. Existeixen quatre tipus de funcions que defineixen aquesta interacció: **source**, o com s'obtenen els recursos, **drain**, o com es perden els recursos, i **traders i converters**, que defineixen relacions d'intercanvi entre recursos. L'economia del joc dissenyat implementa les següents funcions:

Vida

- Drain: A mesura que passa el temps el jugador perd vida.
- Converter: El jugador pot consumir aliments per a recuperar punts de vida.

Ítems

- Source: Els recursos s'obtenen de l'escenari.
- Converter: És poden intercanviar ítems mitjançant el sistema de crafteig, obtenint nous recursos.

6.8. Interfícies

Les interfícies del joc es classificaran en dues categories: interfícies integrades a la pantalla i interfícies desplegables. Les primeres estaran superposades a la càmera del jugador, sent visibles en tot moment, mentre que les segones només seran visibles quan el jugador decideixi que vol obrir-les. S'ha optat per fer aquesta diferenciació ja que els menús d'inventari i de crafteig tenen bastants elements d'interfície i limitarien molt la visió del jugador en cas d'estar sempre disponibles.

6.8.1. Menú inicial

El menú inicial mostrarà el títol del joc sobre un fons representatiu del joc. Oferirà les opcions de jugar i de sortir.

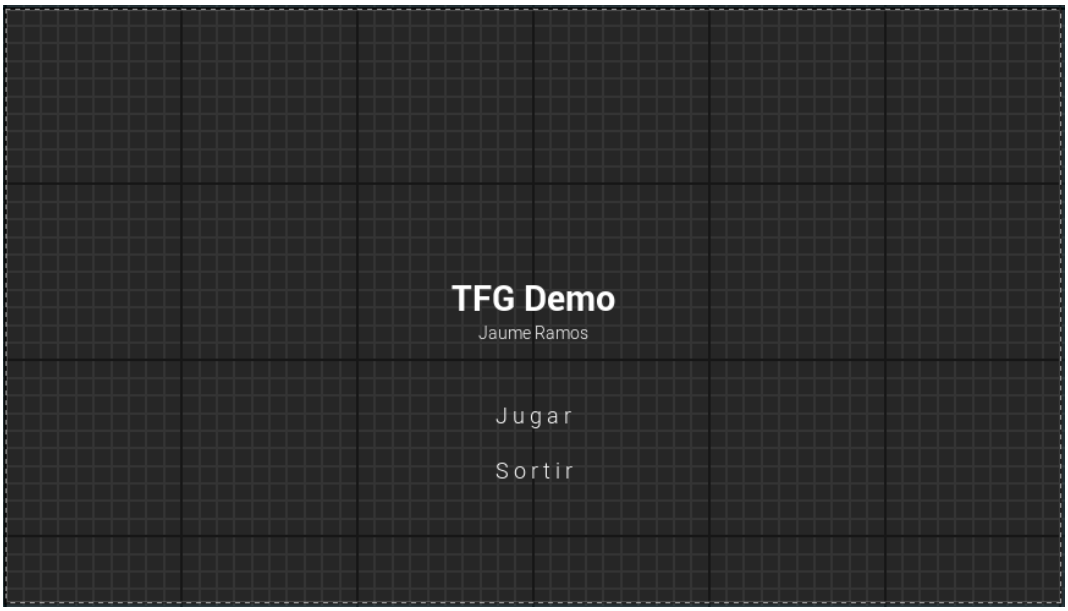


Figura 6.6: Menú inicial

6.8.2. Menú de pausa

El menú de pausa servira per aturar momentàniament el joc. De fons es veurà la imatge de la càmera del personatge però amb un difuminat per ressaltar les opcions de continuar i de sortir.

Figura 6.7: Menú de pausa

6.8.3. Vida

La vida es representarà mitjançant una barra de color vermell que anirà incrementant o decrementant a mesura que el jugador perdi o guanyi vida. A més, dintre de la barra s'indicarà el valor numèric de la vida.

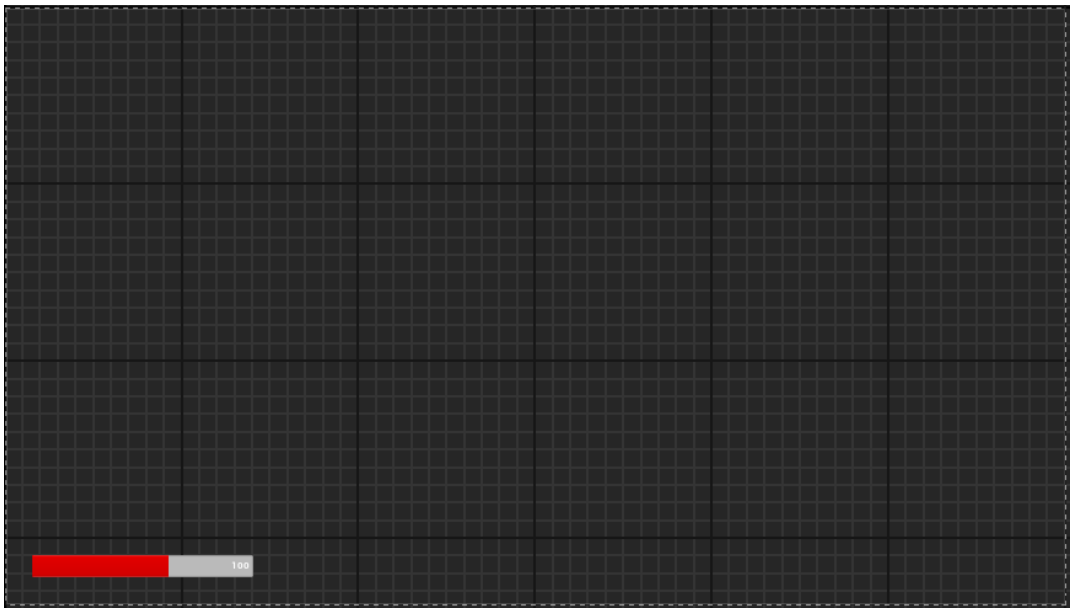


Figura 6.8: Barra de vida

6.8.4. Selecció d’ítems

Aquesta interfície permet al jugador saber amb què està interactuant. Està formada per un punt que indica el centre de la càmera, i quan es detecta algun objecte amb el que es pot interactuar també es mostra el seu nom.

Figura 6.9: Selecció d'ítems

6.8.5. Controls

Aquesta interfície s’ha dissenyat per oferir al jugador una manera ràpida d’accedir als controls en cas que es quedi encallat en algun punt. Aquesta interfície està formada per dues parts, una visible tota l’estona que indica quina tecla permet obrir una segona interfície, que mostra una llista amb tots els controls del joc.

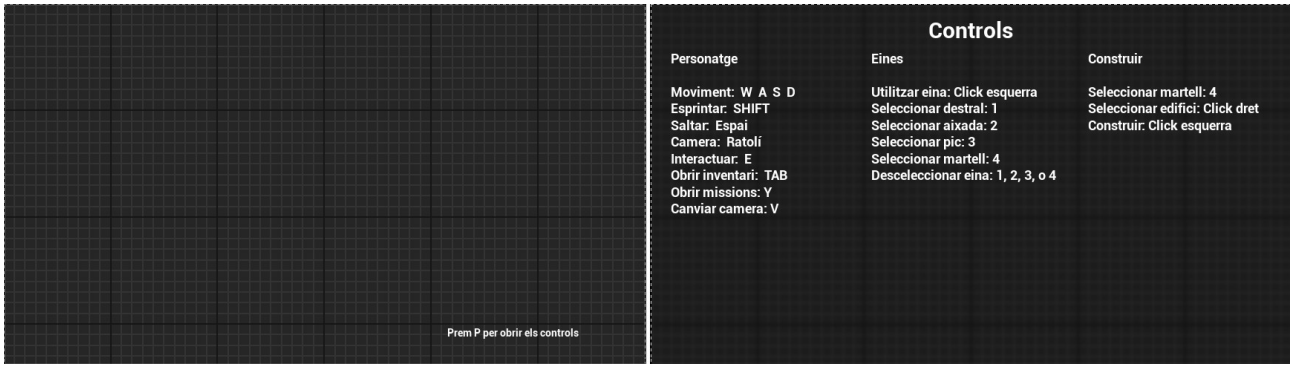


Figura 6.10: Interfícies dels controls

6.8.6. Inventari i crafteig

Com que el crafteig està molt relacionat amb l'inventari, ja que depèn del nombre d'ítems disponibles a l'inventari per tal de poder crear nous objectes, s'ha decidit implementar una única interfície que mostri tant l'inventari com el crafteig. A la part esquerra de la pantalla el jugador podrà veure tota la informació relacionada amb el crafteig, com les receptes, els ítems necessaris... i a la part dreta es podrà veure l'inventari.

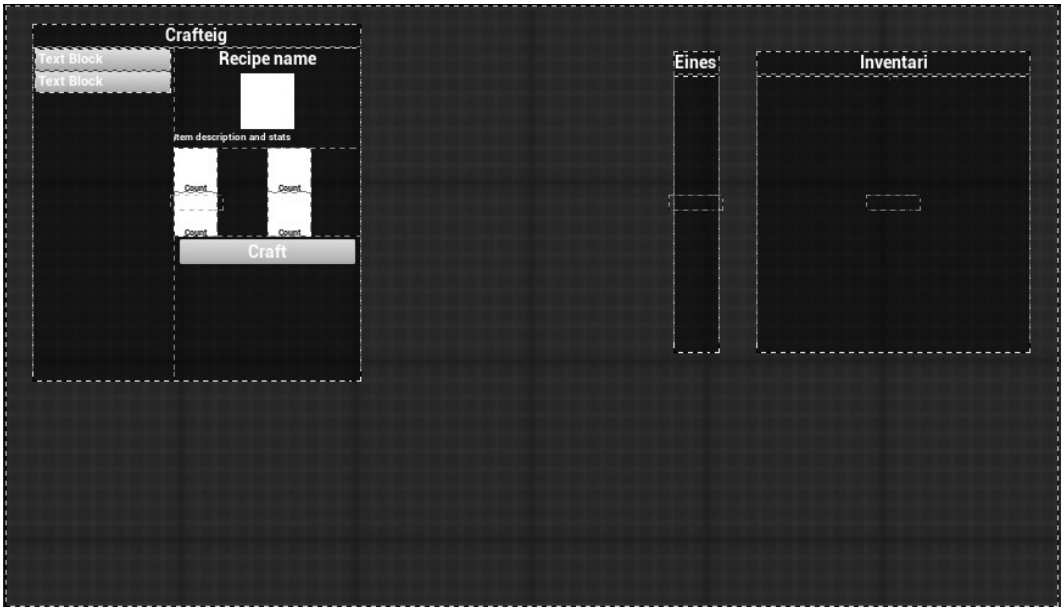


Figura 6.11: Interfície d'inventari i crafteig

6.8.7. Construcció

La interfície de construcció també està formada per dues parts, una per escollir els edificis a construir i una segona que mostra informació sobre l'edifici escollit.

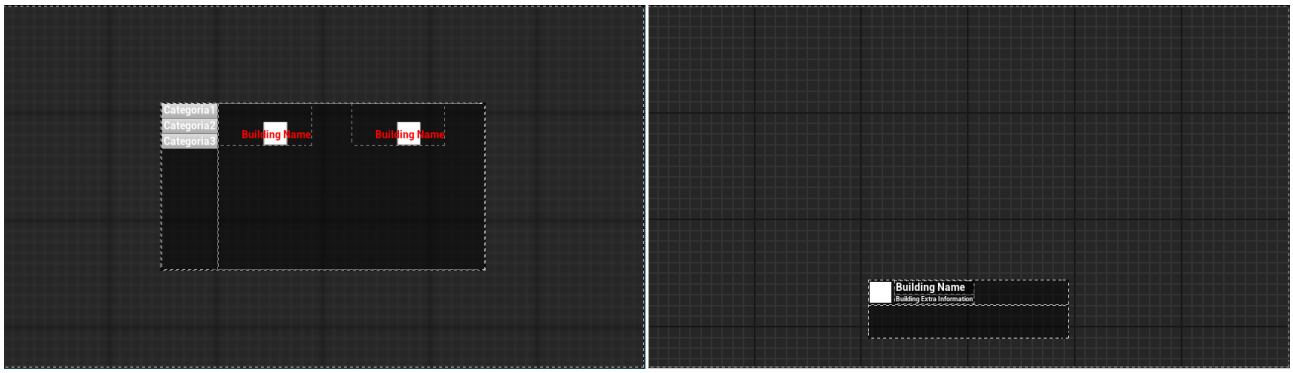


Figura 6.12: Interficies de construcció

6.8.8. Objectius

La interfície dels objectius mostra la llista de missions que es poden desplegar per a mostrar els objectius que les formen.

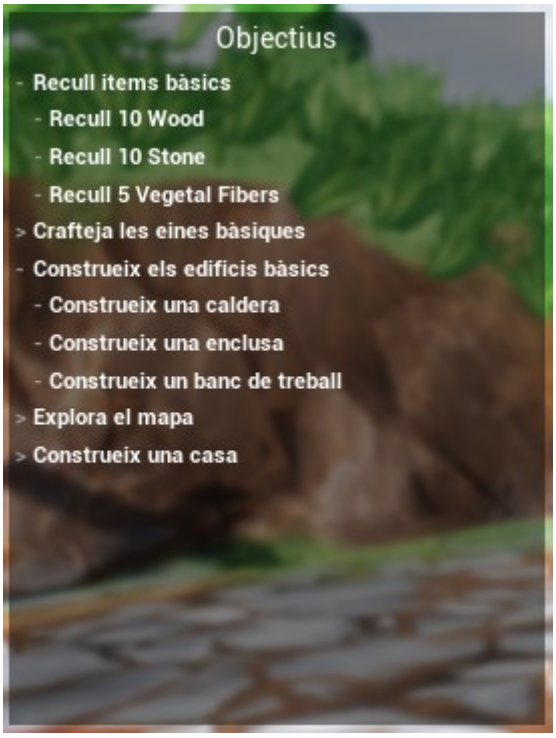


Figura 6.13: Objectius

6.8.9. Pantalla final

La pantalla final mostra un missatge conforme s'ha acabat el joc i permet als jugadors sortir o obrir un formulari per donar feedback sobre el joc.

Figura 6.14: Menú de final

6.9. Diagrama d’execució del joc

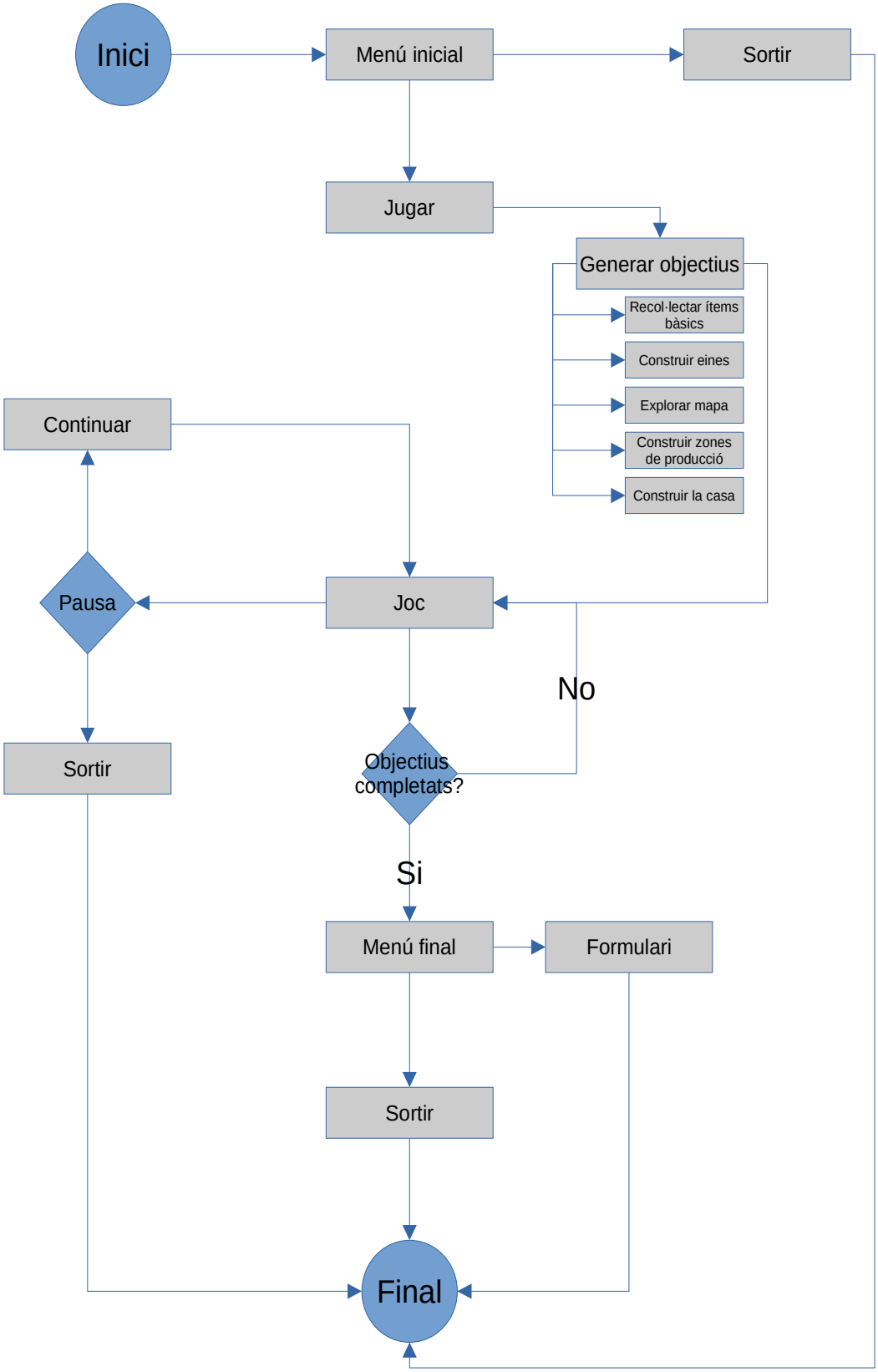


Figura 6.15: Diagrama de flux del joc

6.10. Elements a desenvolupar

Els elements a desenvolupar s'han dividit en tres categories:

- Elements estètics: Aquesta categoria està formada pels elements que es mostraran dintre del joc, com models 3D o textures.
- Elements gràfics: Aquesta categoria estarà formada per tots els elements que formaran les interfícies d'usuari i els menús.
- Elements jugables: Aquesta categoria estarà formada per tots els elements que conformaran la jugabilitat, com les mecàniques o la lògica del joc.

A partir d'aquests grups, i considerant que el perfil dels desenvolupadors del projecte és més tècnic que artístic, s'ha decidit externalitzar la producció dels elements estètics per poder dedicar més temps a polir la jugabilitat. Pel que fa als elements gràfics no s'externalitzarà la producció, però s'utilitzaran els elements d'interfície que porta Unreal Engine 4 per defecte.

7. Implementació i proves

7.1. Estructura del projecte

Com s'ha mencionat anteriorment, l'objectiu d'aquesta part del projecte és implementar una base sòlida a sobre de la qual es pugui afegir contingut de forma ràpida i senzilla. Per aconseguir-ho s'ha dissenyat el videojoc intentant seguir la metodologia de combinar C++ i Blueprints, explicada al Punt 4, ja que permet implementar mòduls de lògica robustos en C++ i utilitzar el sistema de Blueprints per a crear objectes específics a partir de les bases definides. La Figura 7.1 mostra el diagrama de classes de tot el projecte. Les classes que s'implementaran en C++ s'han marcat amb color verd, mentre que les classes que s'implementaran en Blueprints s'han marcat amb blau.

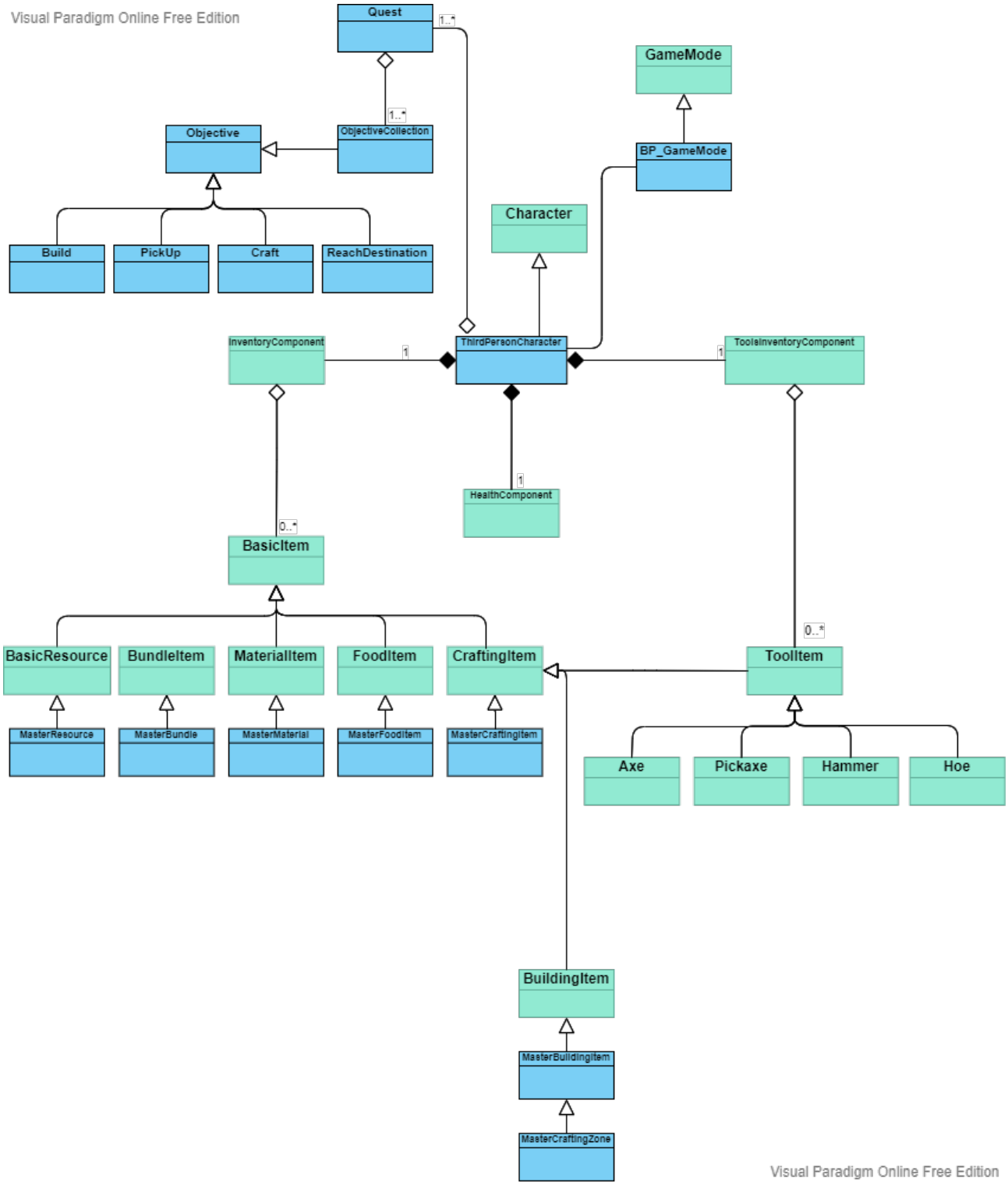


Figura 7.1: Diagrama de classes global del projecte

7.2. Jugador

El jugador fa referència a un conjunt de classes i d'assets que, combinats, implementen com es veu i es controla el personatge principal. Com ja s'ha mencionat anteriorment, Unreal Engine 4 permet treballar de forma modular, ja que integra diferents eines específiques que permeten implementar les diferents funcionalitats que necessita el jugador.

7.2.1. Disseny

Seguint la idea principal del projecte, s'ha dissenyat l'estructura del jugador combinant C++ i Blueprints. La Figura 7.2 mostra de forma més detallada el diagrama de classes que conforma el jugador, on s'han marcat amb taronja les classes base implementades dintre del propi motor, amb verd les classes personalitzades implementades en C++, i en blau les classes personalitzades implementades en Blueprints.

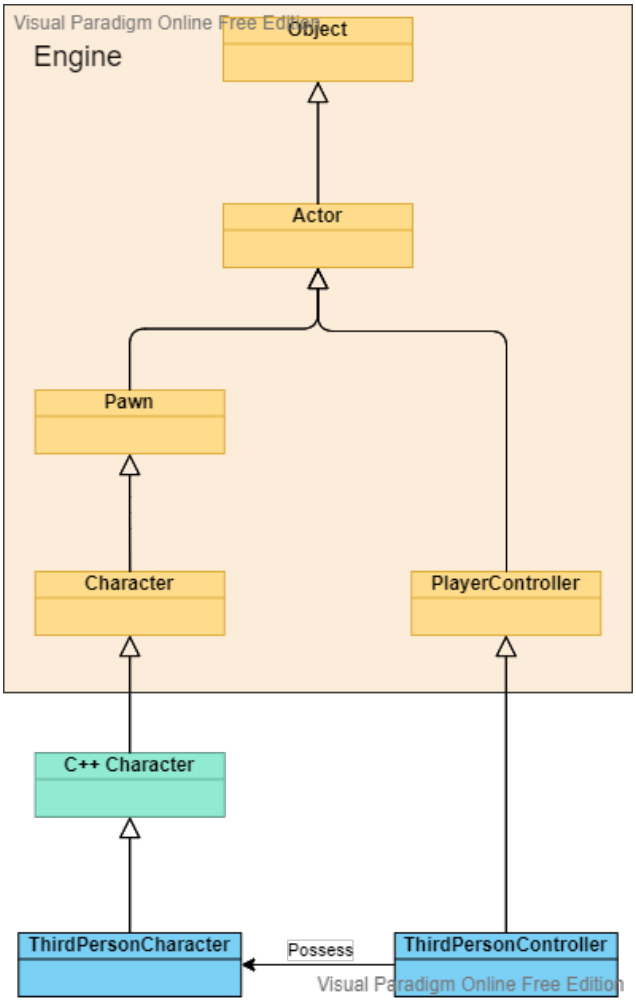


Figura 7.2: Diagrama de classes detallat del jugador

La classe *C++Character* hereta de la classe d'Unreal Engine *Character* les funcionalitats que li permeten gestionar la *mesh*, les col·lisions i la lògica de moviment, i defineix funcionalitats específiques del projecte, com la càmera o la gestió d'inputs.

La classe *ThirdPersonCharacter* permet utilitzar les funcionalitats específiques definides a *C++Character* des de les Blueprints, i també permet gestionar l'aspecte visual del personatge.

La classe *ThirdPersonController* permet utilitzar les funcionalitats de *PlayerController* des de les Blueprints. El *PlayerController* està pensat per a gestionar els inputs del jugador i transmetre'ls al *PlayerCharacter*. En jocs multijugador això permet que l'input sigui independent del personatge, i permet mantenir valors a través de diferents instàncies de la classe *Character*. Per exemple, en un joc tipus shooter, on el jugador pot matar enemics i també pot morir, la gestió d'inputs seria millor fer-la al *PlayerController*, ja que en cas que el jugador moris i hagués de fer *respawn*, el *Character* actual es reiniciaria i perdria el valor de totes les variables associades, però el *Controller* seguiria sent el mateix. En jocs d'un sol jugador la gestió dels inputs es pot fer tant des del *Controller* com des del *Character*. En aquest projecte s'ha decidit gestionar els inputs relacionats amb la jugabilitat, com moure's o interactuar, des del *Character* i gestionar des del *Controller* els inputs externs a la partida, com posar el joc en pausa.

Unreal Engine 4 implementa moltes de les funcionalitats com a Components, ja que permeten treballar de forma modular, definint comportaments genèrics que es poden utilitzar en diferents objectes del joc. La Figura 7.3 mostra la jerarquia de components que conformen el jugador, i indica a quina classe de les anteriors han estat declarats. Els components marcat en taronja hereten de la classe del motor *Character*, els marcats en verd hereten de la classe personalitzada *C++Character*, i finalment els marcats en blau hereten de la Blueprint *ThirdPersonCharacter.*

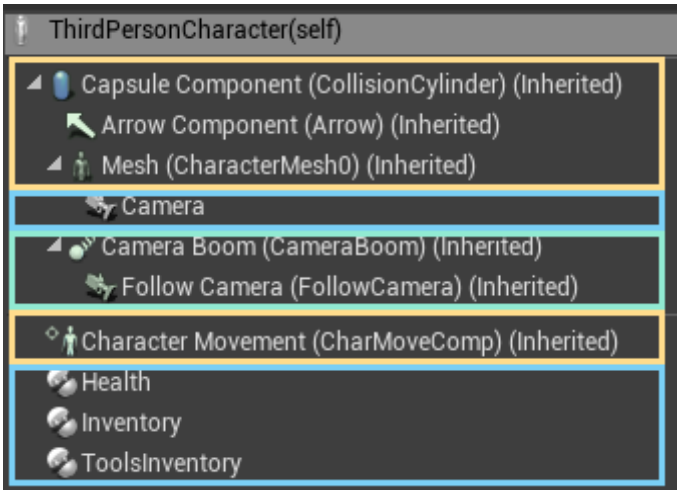


Figura 7.3: Jerarquia de components del jugador

Els següents fragments de codi mostren la declaració dels components que formen la jerarquia anterior. El primer fragment correspon a la classe *`Character`* implementada dintre del motor.

```
class ENGINE_API ACharacter : public APawn
{

private:
	/** The main skeletal mesh associated with this Character (optional sub-object). */
	UPROPERTY(...)
	USkeletalMeshComponent* Mesh;

	/** Movement component used for movement logic in various movement modes (walking, falling, etc), containing relevant settings and functions to control movement. */
	UPROPERTY(...)
	UCharacterMovementComponent* CharacterMovement;

	/** The CapsuleComponent being used for movement collision (by CharacterMovement). Always treated as being vertically aligned in simple collision check functions. */
	UPROPERTY(...)
	UCapsuleComponent* CapsuleComponent;

#if WITH_EDITORONLY_DATA
	/** Component shown in the editor only to indicate character facing */
	UPROPERTY()
	UArrowComponent* ArrowComponent;
#endif

...
}
```

El segon correspon a la classe *C++`Character`* creada específicament per al projecte.

```
class AProjecte_TFGCharacter : public ACharacter
{
	GENERATED_BODY()

	/** Camera boom positioning the camera behind the character */
	UPROPERTY(...)
	class USpringArmComponent* CameraBoom;

	/** Follow camera */
	UPROPERTY(...)
	class UCameraComponent* FollowCamera;

...
}
```

Un cop vist el disseny del personatge des de la perspectiva de les classes i el codi, es mostrarà com es tradueix aquesta estructura de classes a assets o elements que es puguin utilitzar dintre de l'editor d'Unreal Engine 4. La Figura 7.4 mostra els assets que conformen el jugador, i com estan relacionats entre ells. L'element principal és la Blueprint *`ThirdPersonCharacter`*, ja que és on es gestionen tots els components explicats anteriorment. En taronja s'han marcat tots els assets que conformen la *skeletal mesh* del jugador, que són el *`character_Skeleton`*, que representa la jerarquia d'ossos, el *character*, que representa el model 3D, i el *`character_PhysicsAsset`*, que s'utilitza per aplicar físiques al model. La Animation Blueprint *`BP_CharacterAnimation`* s'encarrega de gestionar les animacions a sobre del model a partir de les accions del *`ThirdPersonCharacter`*. Finalment la Blueprint *`ThirdPersonController`* gestiona els inputs externs al gameplay, com s'ha explicat anteriorment.

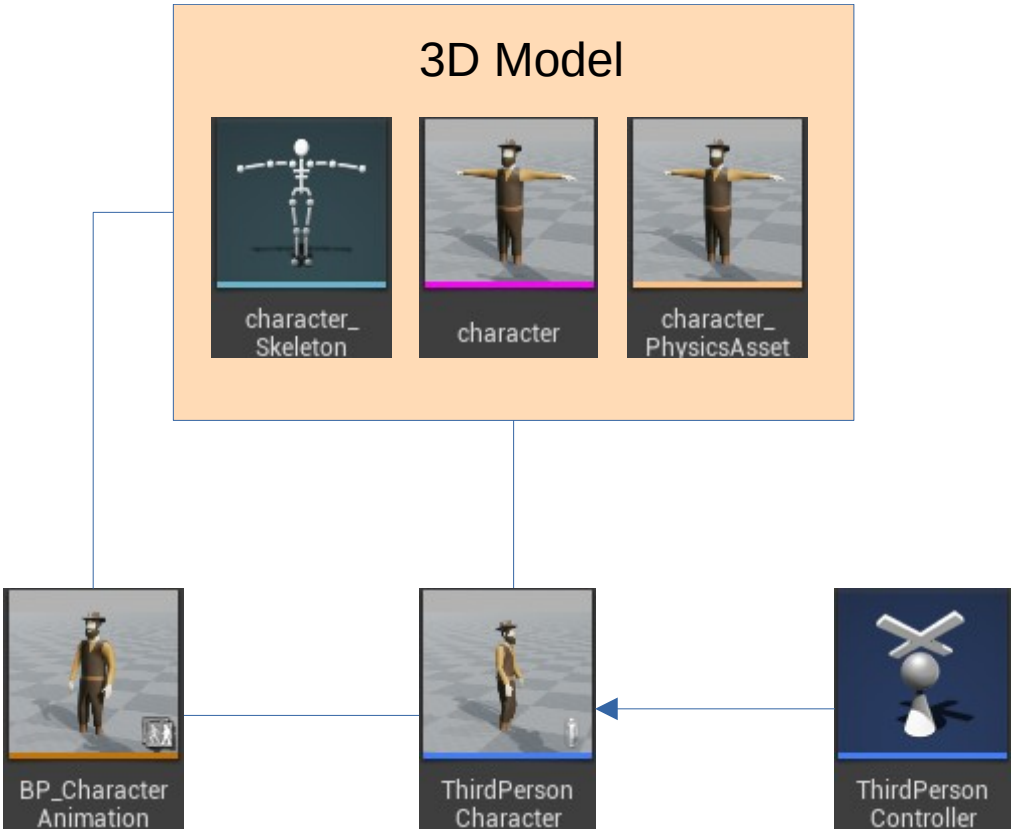


Figura 7.4: Assets que conformen el jugador

7.2.2. Accions

La funció de l'estructura definida anteriorment és processar els inputs del jugador real, executar les funcionalitats corresponents i finalment mostrar el resultat al món del joc. En aquest punt s'explicarà la gestió de les diferents accions del jugador i com es transmeten.

7.2.2.1. Inputs

La Figura 7.5 mostra els inputs que s'han definit pel projecte i quines tecles els disparen.

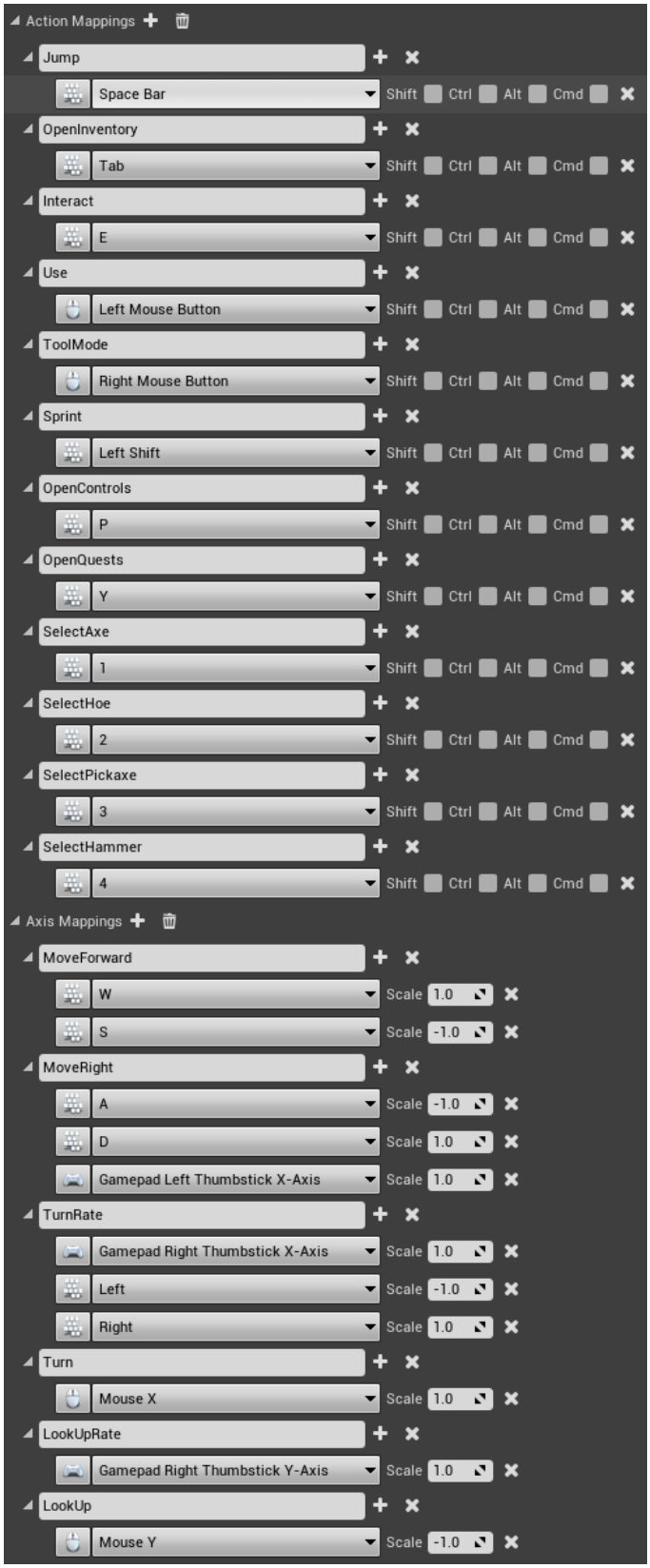


Figura 7.5: Input mappings dels controls del joc

7.2.2.2. Interacció

Els únics objectes amb els què el jugador pot interactuar són els ítems i les zones de crafteig. Quan el jugador vol interactuar amb algun objecte prem la tecla vinculada a l'esdeveniment InputAction *`Interact`*, moment en el qual el joc comprova si el jugador està apuntant a algun objecte. Si el jugador apunta a un objecte, es comprova si és del tipus ítem o zona de crafteig, i s'afegeix a l'inventari o es mostra la interfície corresponent, com s'il·lustra al diagrama de flux de la Figura 7.6.

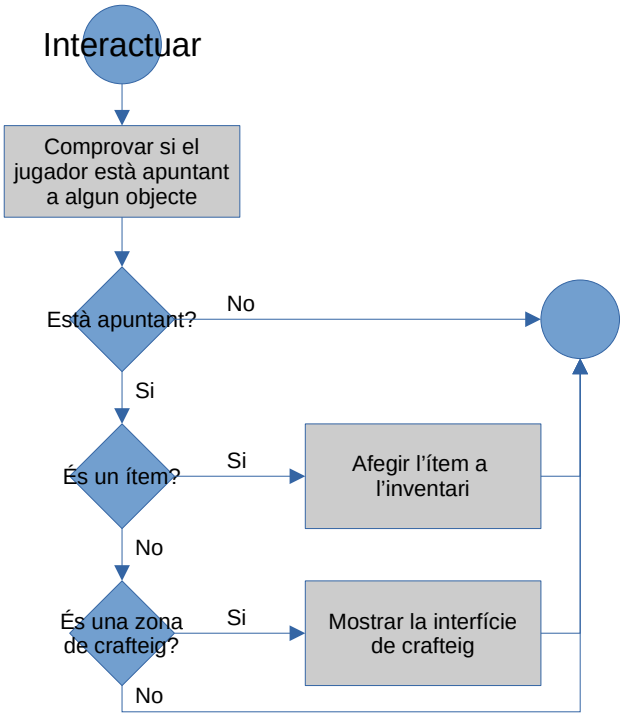


Figura 7.6: Diagrama de flux de l'acció interactuar

Per comprovar si el jugador està apuntant a algun objecte, s'ha utilitzat la tècnica del *raycasting*, que consisteix en llençar un raig invisible des d'un punt d'origen a un punt de destí per detectar si hi ha algun model 3D entre els dos punts. Unreal Engine 4 permet detectar tots els objectes situats a l'escenari o permet definir categories de col·lisions i només detectar objectes d'una o vàries categories concretes. En el cas d'interactuar s'han de detectar les col·lisions amb els objectes de les categories *`Item`* i *`Interactuable`*.

Com que els objectes que es volen detectar estaran al camp de visió del jugador, l'origen dels rajos serà el centre de la càmera, i es calcularà el destí aplicant una distància màxima a l'origen. La Figura 7.7 mostra com es representen aquests punts respecte al jugador.

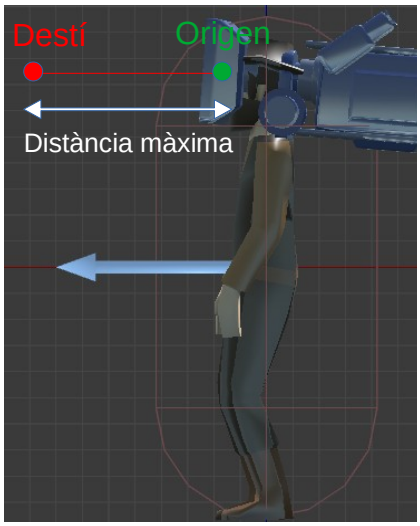


Figura 7.7: Configuració dels rajos respecte al jugador

A nivell d'implementació només s'ha de calcular el punt de destí, ja que l'origen és la posició actual de la càmera, a la qual es pot accedir a través de la funció `GetWorldLocation()`. Per calcular el destí s'ha utilitzat el `ForwardVector` de la càmera, que indica cap on apunta un objecte dintre de d'una escena 3D, multiplicat per una variable entera que defineix la distància màxima a la qual es poden agafar objectes. Finalment per trobar el destí s'ha sumat el punt calculat a l'origen. La Figura 7.8 mostra com s'han implementat aquests càlculs, i la Figura 7.9 mostra els punts i els rajos generats dintre del joc.

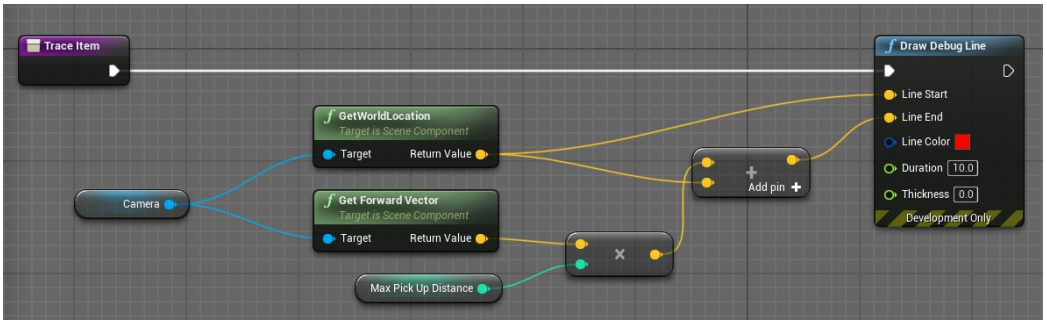


Figura 7.8: Implementació dels càlculs per detectar els punts d'origen i de destí

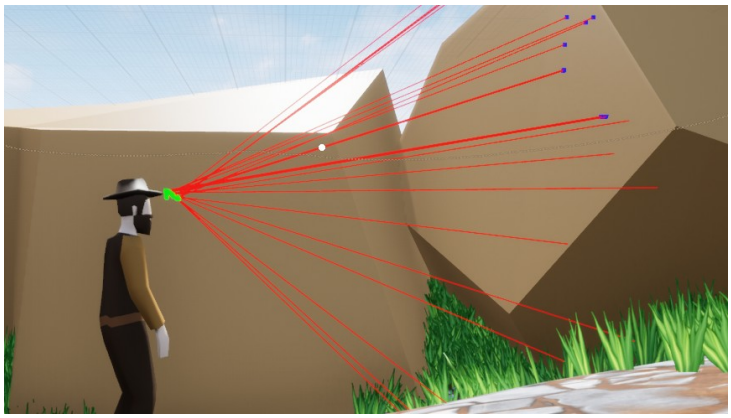


Figura 7.9: Visualització dintre del joc dels punts d'origen (verd), punts de destí (blau) i rajos (roig)

Mitjançant els punts generats i la funció `LineTraceForObjects()` es pot detectar si hi ha algun model entre el jugador i el punt de destí. Aquesta funció rep les categories d'objectes a detectar, el punt d'origen i de destí i retorna una estructura `HitResult` amb informació sobre la col·lisió. La Figura 7.10 mostra la funció `LineTraceForObjects()` i l'estructura que retorna.

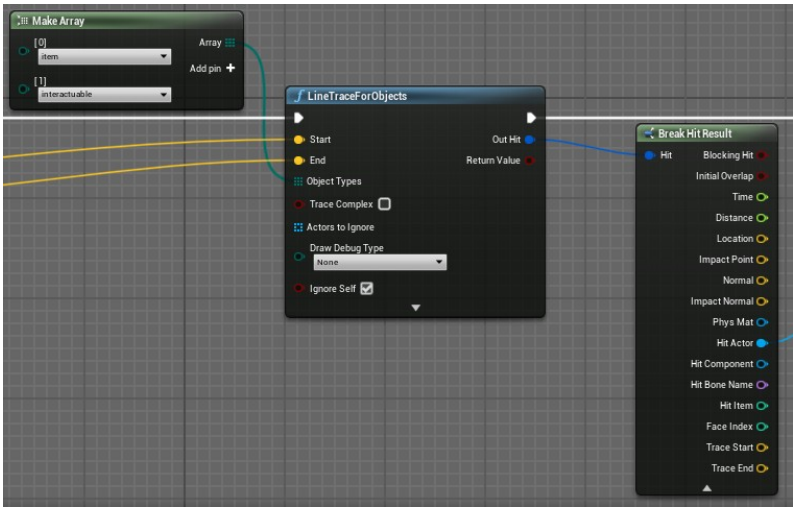


Figura 7.10: Funció LineTraceForObjects i estructura HitResult

Tots aquests punts junts conformen la funció `TraceItem()`, que es pot veure sencera a la Figura 7.11.

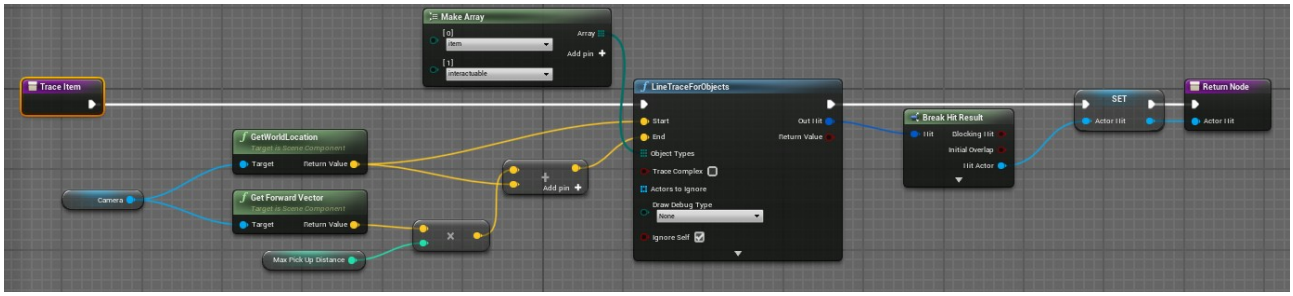


Figura 7.11: Funció TraceItem

Un cop detectat l'objecte amb el qual s'ha d'interactuar, s'ha de determinar si és del tipus ítem o zona de crafteig, ja que la referència que retorna la funció `TraceItem()` és del tipus `Actor`. Si l'objecte és del tipus `Item` s'ha d'afegir a l'inventari, i si és del tipus zona de crafteig s'ha de mostrar la interfície de crafteig.

El jugador té dos inventaris diferents, un on es guarden els recursos i un altre on es guarden les eines. El primer pas abans d'afegir l'ítem a l'inventari es comprovar si és una eina o si és un altre tipus d'ítem. En ambdós casos la lògica és la mateixa, i només canvia el lloc on es guarden les dades. Quan se sap el tipus d'ítem es crida a la funció de l'inventari `AddToInventory()` i es passa la referència de l'ítem. La implementació de l'inventari es veurà més endavant. Un cop s'ha afegit l'ítem a l'inventari s'actualitzen les interfícies i els objectius, es reprodueix el so conforme s'ha agafat un objecte i es desactiva l'objecte de l'escena. La Figura 7.12 mostra la implementació de la funció `AddToInventory()`.

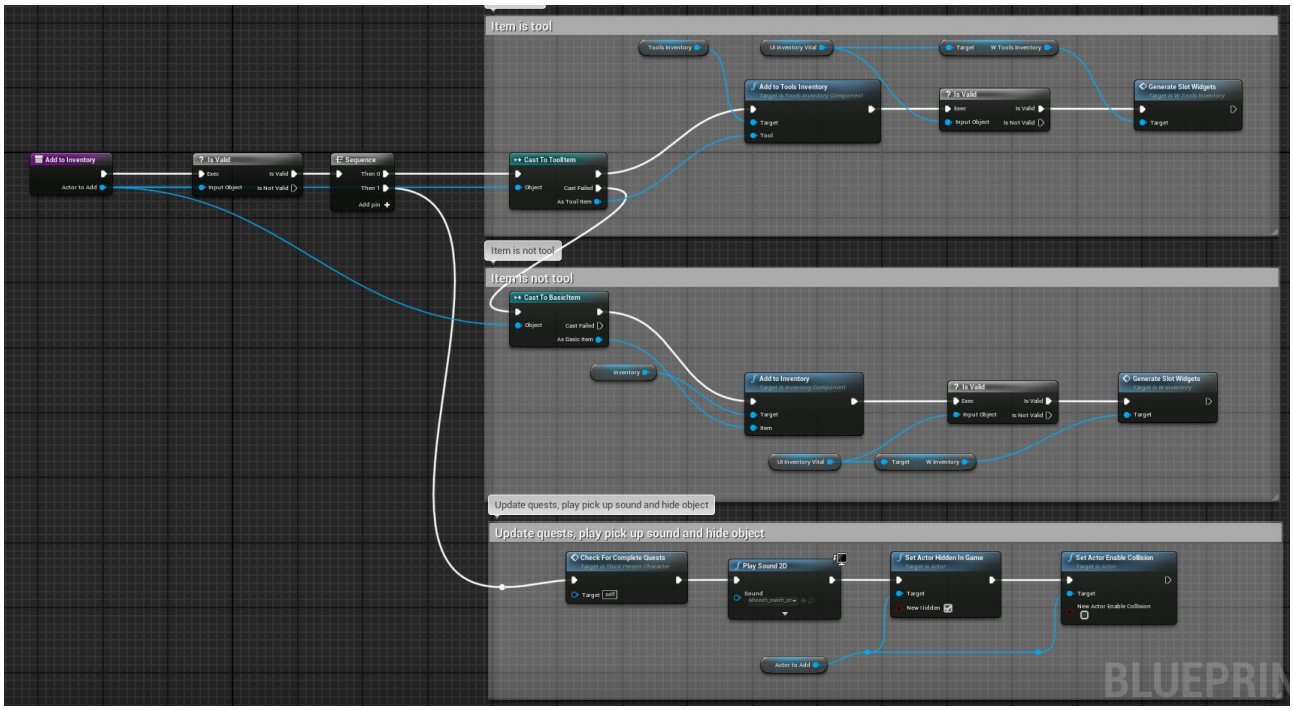


Figura 7.12: Funció AddToInventory

Si l'objecte és del tipus zona de crafteig es crea el *widget* que conté la interfície (Figura 6.11), s'afegeix a la càmera del jugador i es mostren les receptes del tipus concret de la zona de crafteig amb la què s'està interactuant. La Figura 7.13 mostra el codi que crea aquesta interfície.

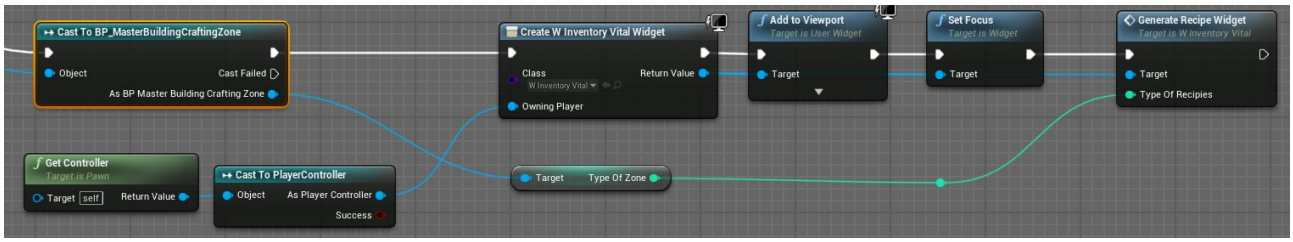


Figura 7.13: Crear interfície de crafteig

Finalment, la Figura 7.14 mostra la implementació completa de l'esdeveniment d'interactuar, on s'utilitzen les funcions explicades en aquest apartat.

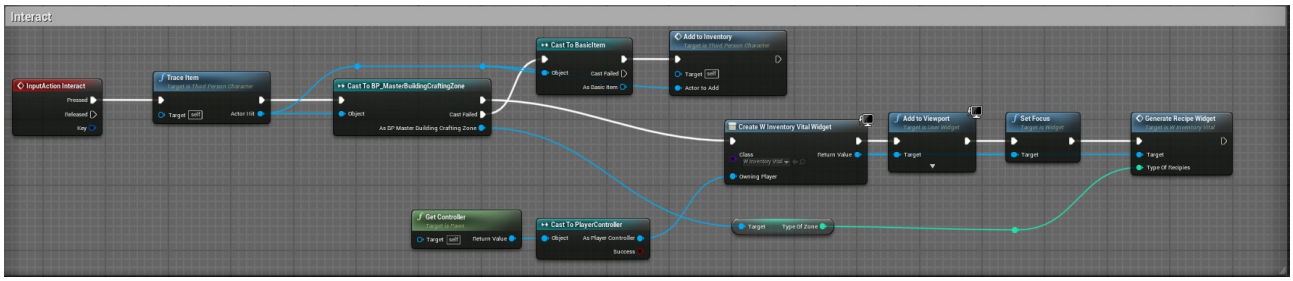


Figura 7.14: Esdeveniment InputActionInteract

7.2.2.3. Seleccionar eines

El jugador pot utilitzar diferents eines que li permeten realitzar accions extra, com tallar arbres o construir edificis. Les eines fabricades a partir d'elements bàsics, s'emmagatzemen en un inventari d'eines, i per poder utilitzar-les el jugador se les ha d'equipar. Per a seleccionar les eines s'han definit les tecles de l'1 al 4, com a `InputActions` que disparen l'esdeveniment que s'encarrega de seleccionar l'eina corresponent. La tecla 1 permet seleccionar la destral, la 2 l'aixada, la 3 el pic i la 4 el martell. Si el jugador té una eina seleccionada i vol escollir-ne una altra, ha de prémer la tecla de la nova eina a seleccionar per desseleccionar l'antiga i equipar la nova. La Figura 7.15 mostra mitjançant un diagrama de flux el procés de selecció d'una eina.

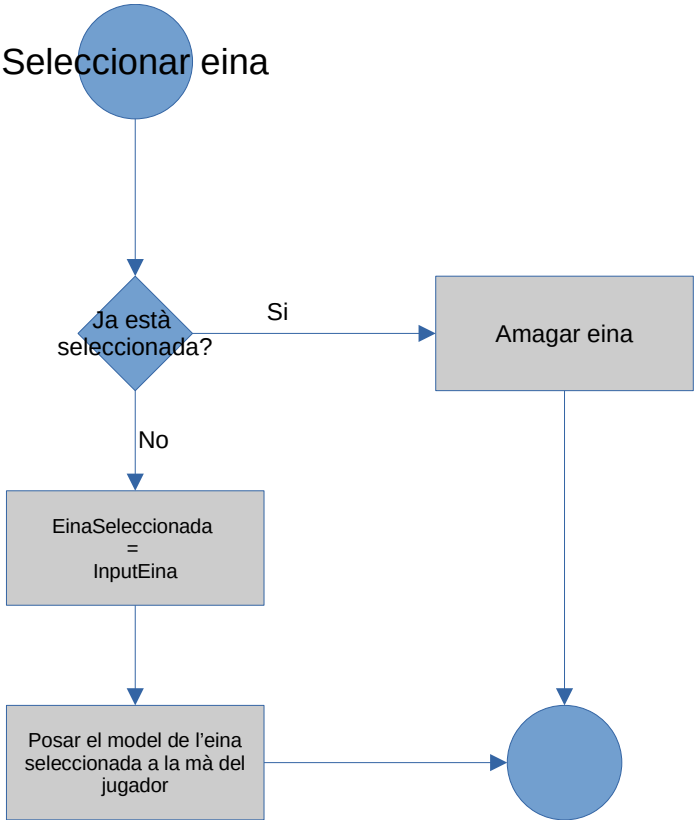


Figura 7.15: Diagrama de flux de l'acció seleccionar eina

S'ha implementat amb Blueprints la funció *`SelectTool()`*, on s'implementa tota la lògica relacionada amb la part visual de seleccionar les eines, que consisteix en instanciar i spawnejar a l'escenari l'ítem de l'eina seleccionada i vincular-lo al model del jugador per a que es mogui amb ell. També s'encarrega d'amagar el model de l'eina en cas que el jugador la desseleccioni.

La Figura 7.16 mostra el codi que amaga el model de l'eina seleccionada, que es troba a la variable *`ActiveToolActor`*, i la interfície que mostra el nom de l'eina, que es troba a la variable *`SelectedToolWidget`*, en cas que el nom de l'eina seleccionada sigui «None».

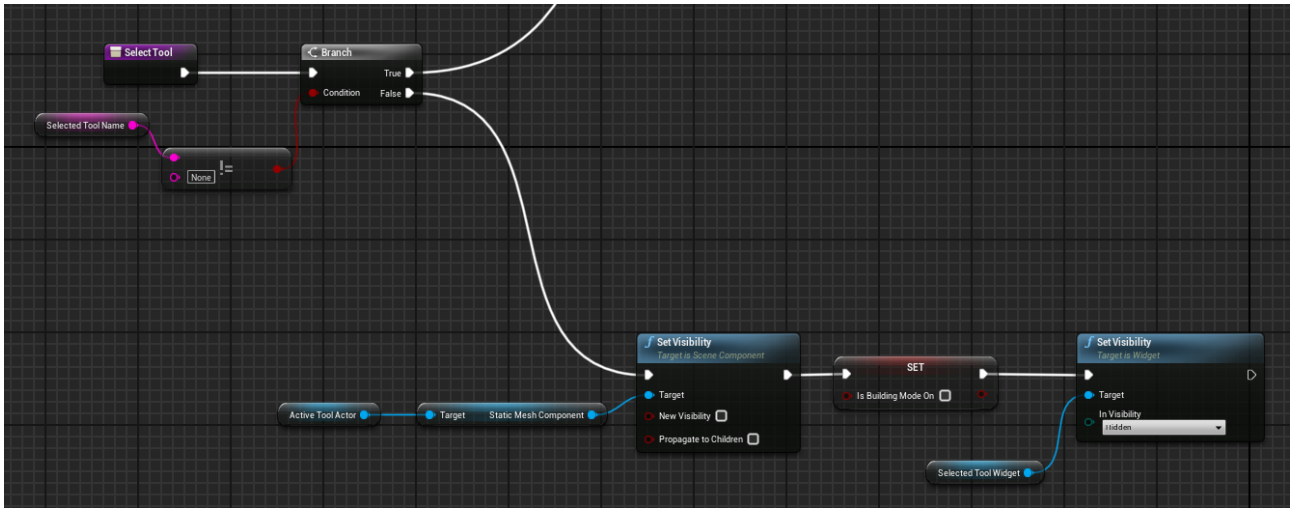


Figura 7.16: Amagar eina

Si la variable *`SelectedToolName`* té el nom d'alguna eina, es comprova que aquesta estigui a l'inventari del jugador, ja que no es pot equipar una eina sense haver-la fabricat. Per fer-ho s'ha utilitzat la funció *`FindTool()`* implementada al component que gestiona l'inventari de les eines. Si l'eina nova ja existeix a l'inventari, es destrueix el model de l'eina anterior. La Figura 7.17 mostra la implementació d'aquesta part de l'script.

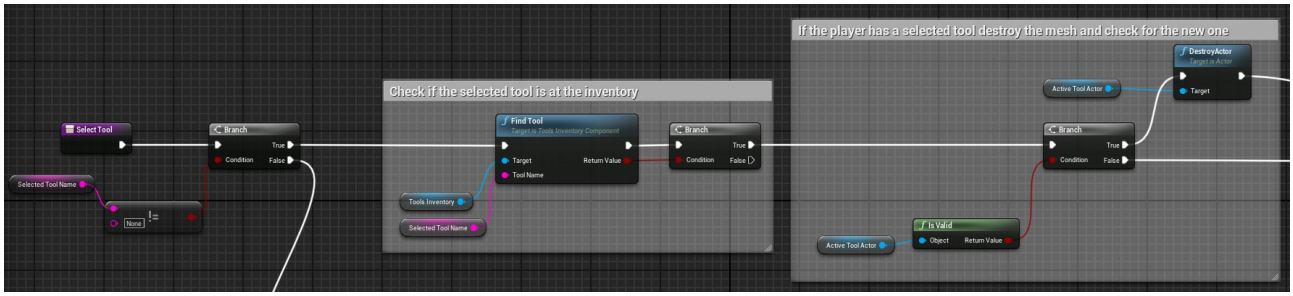


Figura 7.17: Comprovar si l'eina existeix

Per poder instanciar un objecte des d'Unreal Engine 4, fa falta una referència a la classe corresponent, per això, abans de spawnejar l'eina, s'ha d'assignar la classe corresponent segons el nom de la nova eina. També s'han d'activar o desactivar els flags de plantar i construir segons les necessitats de la nova eina. La Figura 7.18 mostra el switch que implementa aquesta lògica.

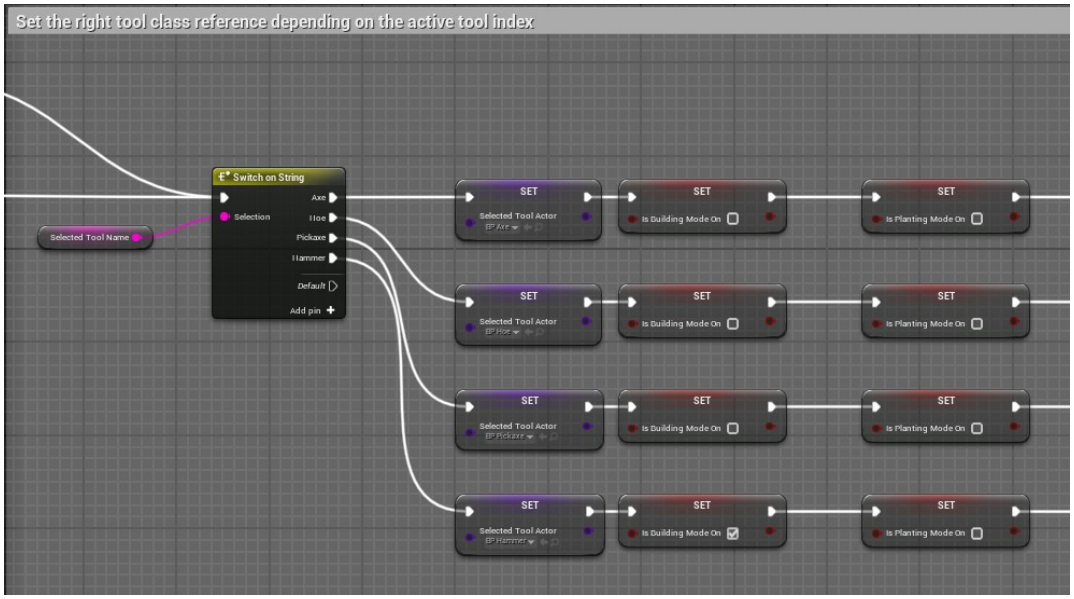


Figura 7.18: Seleccionar la classe de l'eina actual

A partir de la classe assignada es pot instanciar l'objecte de l'eina seleccionada i fixar-lo a la mà del jugador. Per fer-ho s'ha creat un *socket* a la mà del model del personatge anomenat `toolSocket`. Un *socket* és una funcionalitat de les *StaticMesh* que permet definir punts dintre dels models que es poden utilitzar des del codi per a enganxar eines, armes, efectes, etc al model. La Figura 7.19 mostra el model del jugador amb el socket *toolSocket.*

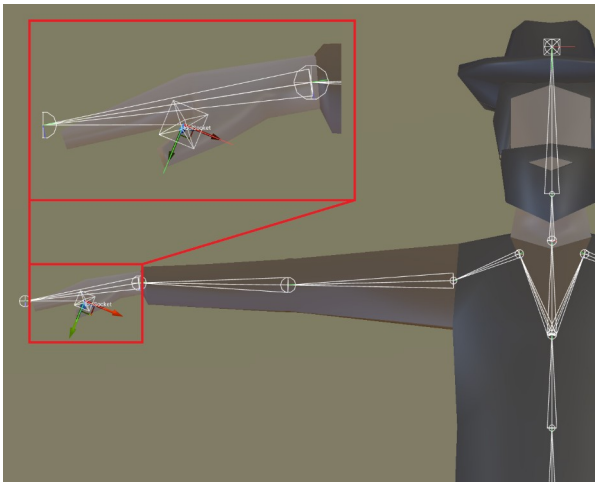


Figura 7.19: Socket dintre del model

Un cop instanciada l'eina s'ha d'utilitzar la funció del motor `AttatchActorToComponent()` per vincular un objecte a un *socket*. La Figura 7.20 mostra com, mitjançant la funció `SpawnActor()`, es genera la instància de l'eina i, mitjançant la funció `AttatchActorToComponent()`, es vincula l'Actor de l'eina amb el *socket* `toolSocket`. Finalment es desactiven les col·lisions de l'eina per evitar que interactui amb el jugador.

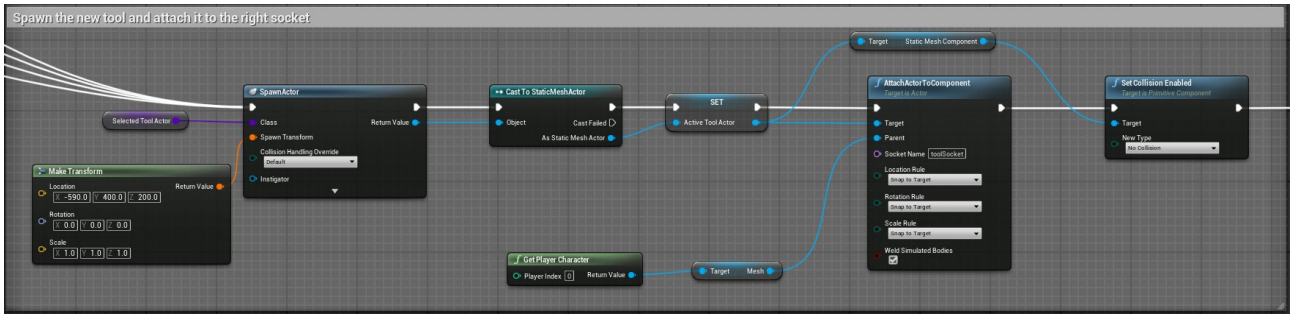


Figura 7.20: Instanciar i vincular una eina amb el jugador

Per acabar amb la implementació de la funció `SelectTool()`, s'ha d'actualitzar la interfície que mostra al jugador el nom de l'eina seleccionada perquè mostri el nom de la nova eina. A la Figura 7.21 es pot observar el codi que s'encarrega de fer-ho.

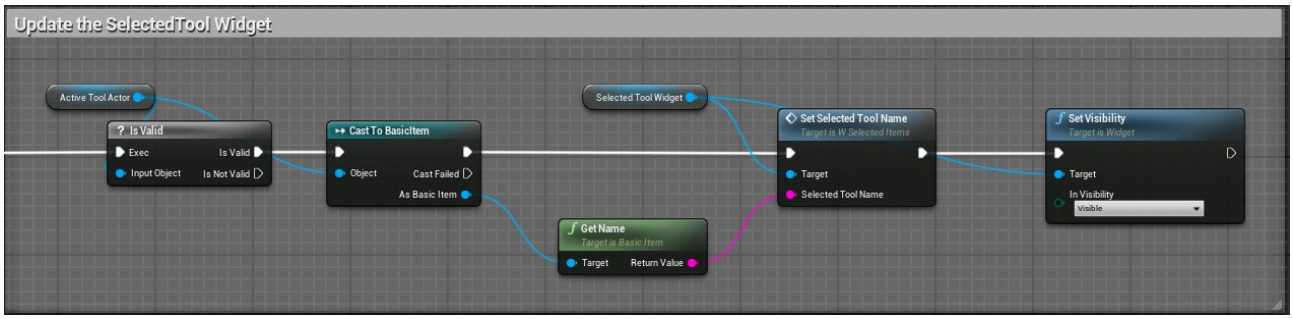


Figura 7.21: Actualitzar la interfície

La Figura 7.22 mostra la implementació completa de la funció `SelectTool()`.

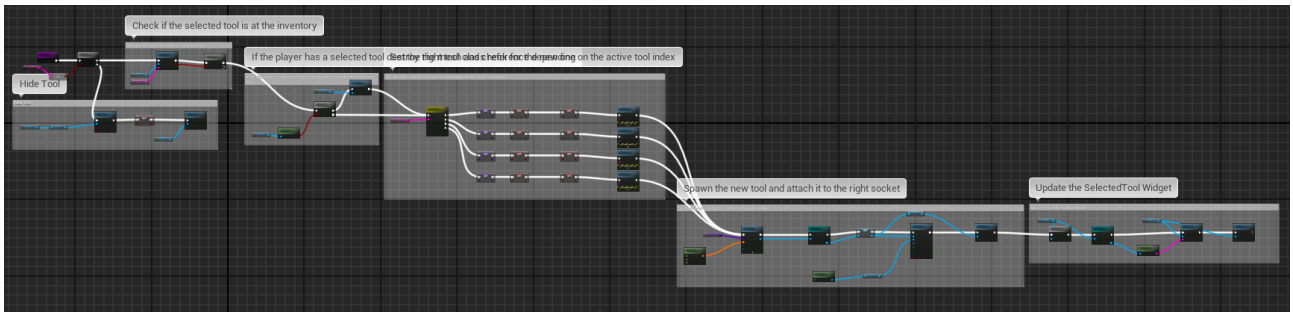


Figura 7.22: Implementació de la funció SelectTool()

Per saber quina eina hi ha seleccionada el jugador té una variable *SelectedToolName,* de tipus String, que guarda el nom de l'eina, o «None» en cas que no hi hagi cap eina seleccionada. Quan el jugador prem la tecla per seleccionar alguna eina es comprova que el valor de la variable *SelectedToolName* no sigui igual al de l'eina que s'està seleccionant. Si el valor és diferent s'assigna el nom de la nova eina a la variable *SelectedToolName,* però si el valor és igual cal desseleccionar l'eina assignant el valor «None» a aquesta variable. Un cop assignat el valor de l'eina seleccionada es crida a la funció *SelectTool()* que gestiona els models 3D de les eines. La Figura 7.23 mostra la implementació dels esdeveniments que seleccionen eines.

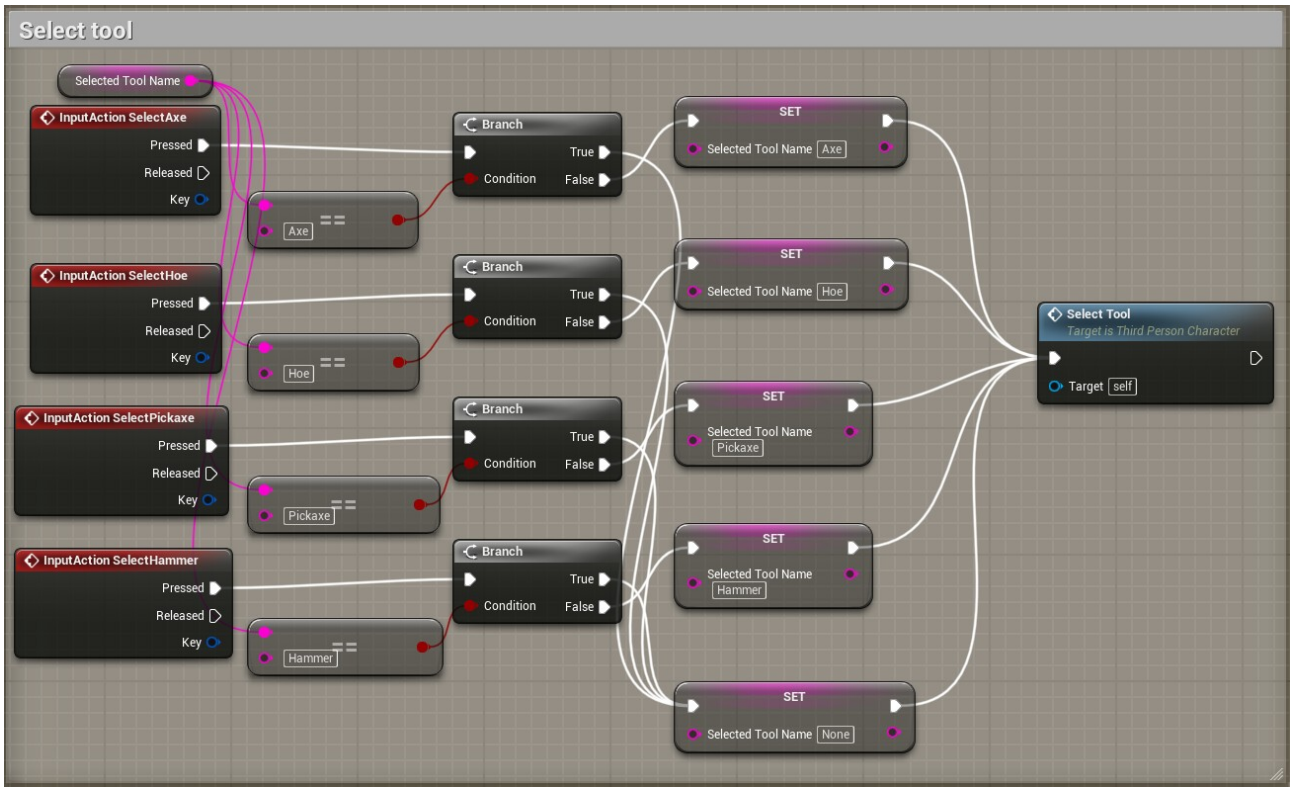


Figura 7.23: Implementació dels esdeveniments InputActionSelectTools

7.2.2.4. Utilitzar eines

Un cop el jugador ha seleccionat una eina, pot utilitzar-la per a realitzar algunes tasques. Per a utilitzar les eines s'ha definit el botó esquerre del ratolí com a un *InputAction* que dispara l'esdeveniment *InputActionUse.* Aquest esdeveniment comprova que el jugador tingui l'eina seleccionada a l'inventari, i si la té crida a la funció *UseTool().* Aquesta és una funció molt senzilla que simplement crida a la funció *use()* de l'eina seleccionada, que és on s'implementa la funcionalitat de cada eina, com es veurà més endavant. La Figura 7.24 mostra la implementació de la funció *UseTool()*.

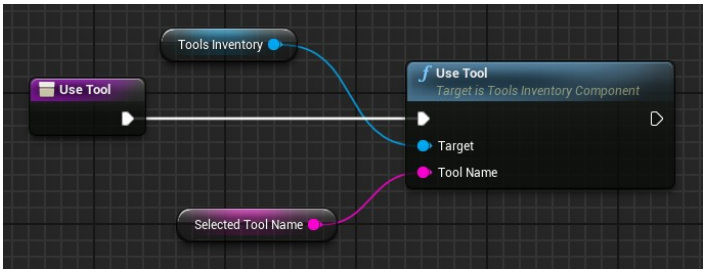


Figura 7.24: Implementació de la funció UseTool()

Quan es va decidir afegir animacions per a representar l'ús de les diferents eines de forma més visual, van sorgir un parell de problemes:

- Cridar directament a la funció `UseTool()` no funcionava bé, ja que la càmera es movia amb l'animació i quan es calculava, mitjançant *Raycasting*, si el jugador estava interactuant amb alguna cosa, no s'obtenien resultats fiables, ja que l'origen dels raigs canviava segons la posició de la càmera.
- També va aparèixer un problema de sincronització, ja que l'eina actuava immediatament després de clicar, però l'animació encara no havia arribat al punt on l'eina interactuava amb l'objecte. Això es notava, per exemple, al tallar arbres, ja que s'obtenia la fusta abans que la destral hagués tocat el tronc de l'arbre.

Per solucionar el primer problema s'ha fet el *Raycast* abans de reproduir les animacions, guardant el resultat en una variable anomenada `ActorHit`. Al Punt 7.2.3 s'explicarà el funcionament de les animacions i de les notificacions. La Figura 7.25 mostra la implementació de l'esdeveniment `InputActionUse`, on es pot veure com després de comprovar si l'eina existeix, es crida la funció `TraceResource()` i segons l'eina seleccionada es reprodueix l'animació corresponent. El martell no té cap animació, i es crida directament la funció *UseTool().*

Per solucionar el segon problema s'ha fet la crida a la funció `UseTool()` des de les animacions de les eines mitjançant Animation Notifications. Aquesta solució sincronitza l'execució de la funció amb el frame de l'animació on l'eina interactua amb l'escenari.

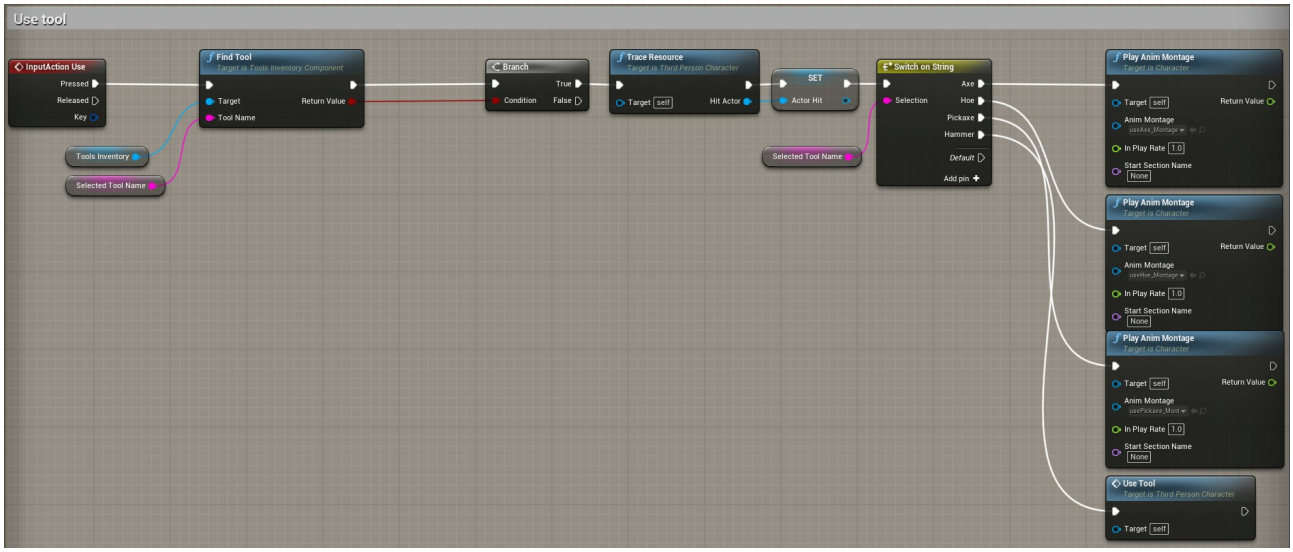


Figura 7.25: Implementació de l'esdeveniment InputActionUse

7.2.2.5. Construir i plantar

El jugador pot col·locar ítems a l'escenari mitjançant el martell o l'aixada. Aquestes dues accions s'implementen igual, però canvien els models i receptes que utilitzen, per això en aquest punt s'explicarà la implementació de l'acció construir per a definir una base a sobre de la qual s'explicaran les característiques úniques del mode plantar.

Quan el jugador selecciona el martell s'activa el mode de construcció. Aquest mode mostra l'edifici seleccionat i permet escollir on vol col·locar-lo. Mitjançant dos materials de previsualització l'edifici es veu verd o roig segons si es pot col·locar o no. La Figura 7.26 mostra en un diagrama de flux el funcionament d'aquest mode.

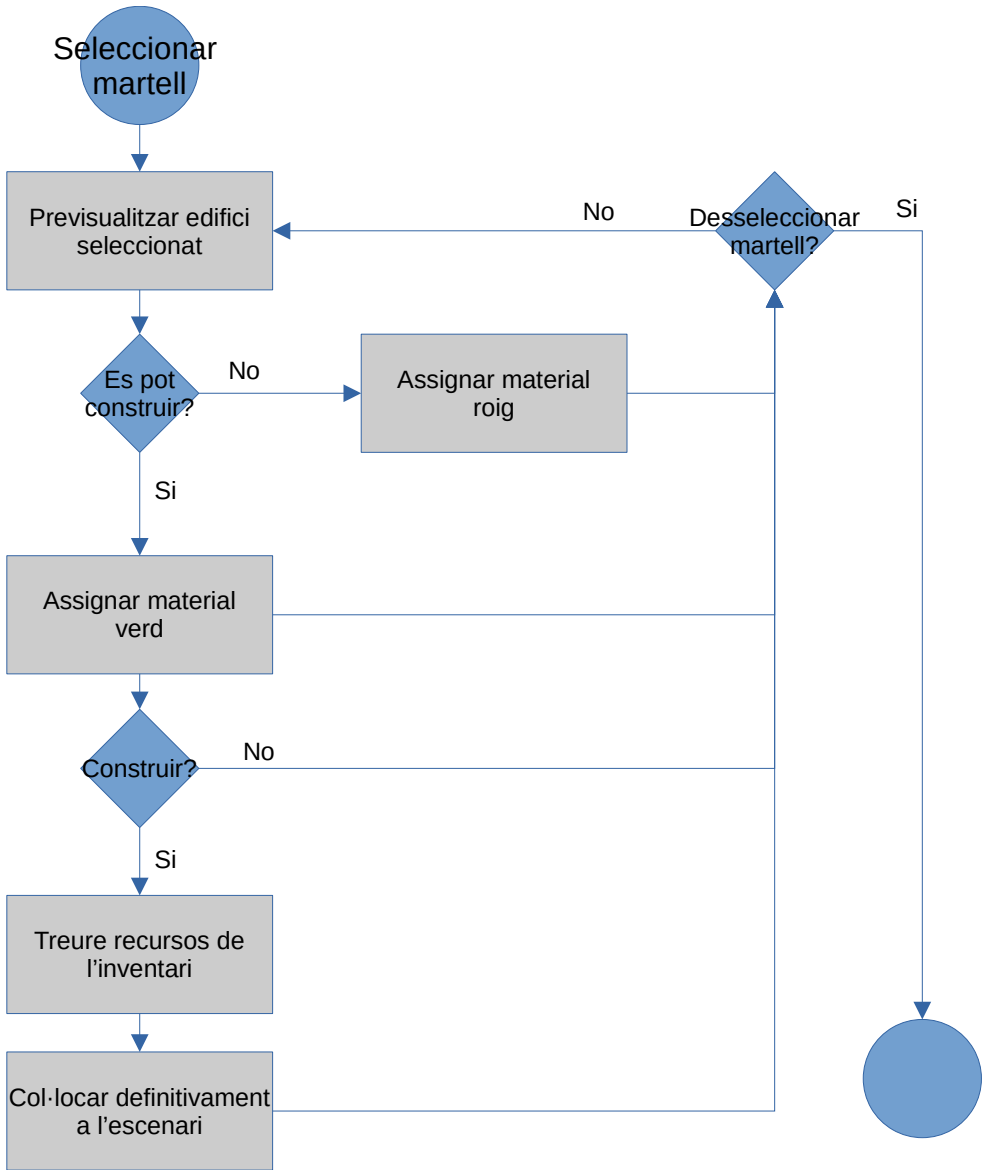


Figura 7.26: Diagrama de flux de l'acció construir

La previsualització de l'edifici s'ha d'anar actualitzant a mesura que el jugador va movent-se per l'escenari, per permetre que pugui col·locar-lo on vulgui. S'ha implementat un esdeveniment anomenat `PreviewBuilding()` que s'executa, mitjançant un *timer*, cada 0.2 segons. Aquest esdeveniment està dividit en tres parts:

- Col·locar l'edifici a l'escena
- Actualitzar la posició de l'edifici segons on apunta el jugador
- Actualitzar el material per informar al jugador si pot construir o no

Per col·locar l'edifici a l'escena s'ha d'instanciar un `Actor` a partir de la classe de l'edifici seleccionat. Amb l'edifici col·locat a l'escena s'ha de guardar una referència per poder

accedir als seus atributs i mostrar la interfície amb la informació de l'edifici. La Figura 7.27 mostra la implementació d'aquesta primera part. Mitjançant un `DoOnce()`, que és un node que executa el codi que va a continuació només un cop, s'assegura que només s'instancia una vegada l'objecte seleccionat. Per poder repetir l'execució d'aquest codi s'ha creat un esdeveniment personalitzat que reinicia el node `DoOnce()`.

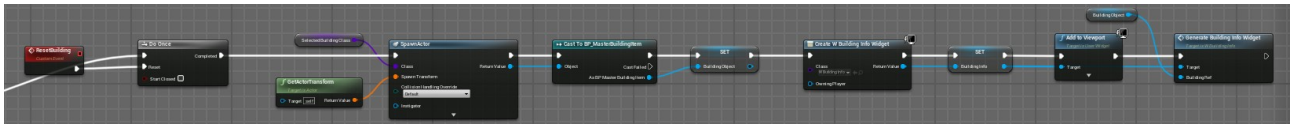


Figura 7.27: Implementació de la part de col·locar l'edifici a l'escenari

La previsualització de l'edifici s'ha de mostrar a la posició on està apuntant el jugador. Per a determinar aquesta posició s'ha implementat la funció `TraceBuildingPoint()`, que, mitjançant *Raycasting*, retorna la posició i la normal del punt on s'ha de construir.

En aquest cas s'ha de detectar el punt del terreny on està apuntant el jugador, per tant s'ha fet un *Raycast* tinguen en compte tots els elements visibles de l'escenari menys l'edifici amb el què s'està previsualitzant. Per ignorar aquest objecte s'ha emmagatzemat la referència en un vector i s'ha passat al paràmetre `ActorsToIgnore` de la funció `LineTraceByChannel()`. La Figura 7.28 mostra la configuració d'aquest primer *Raycast*.

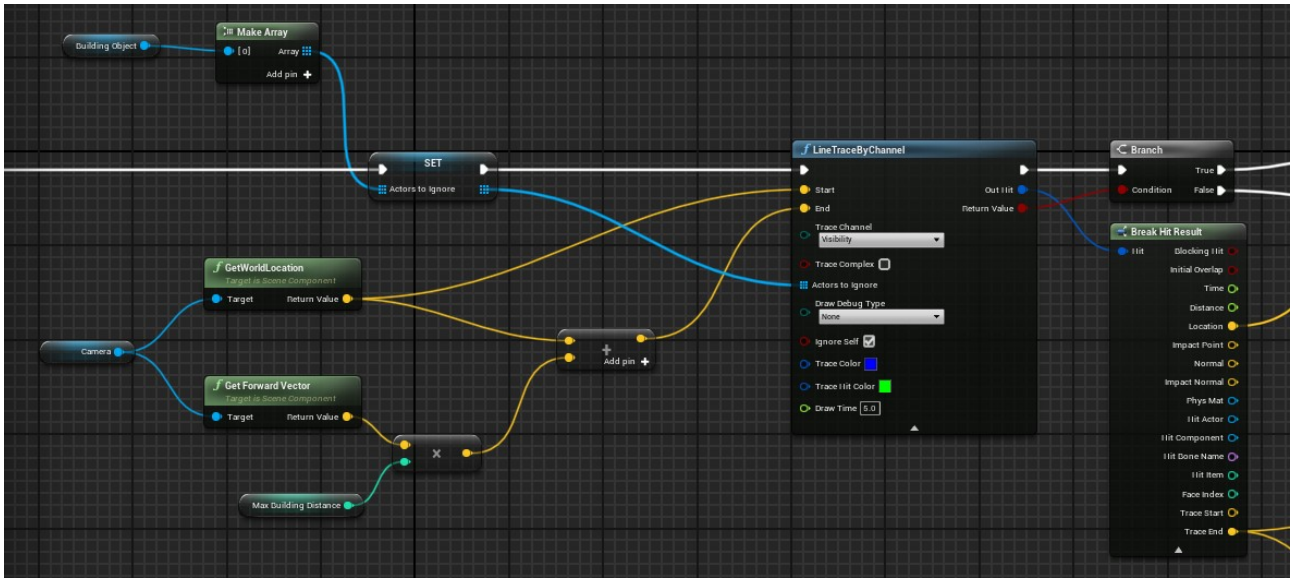


Figura 7.28: Detectar el punt on previsualitzar l'edifici

Un cop fet el primer *Raycast* es poden donar dues situacions, que no s'hagi detectat cap objecte o que si s'hagi detectat un objecte.

La primera situació vol dir que el jugador no estava apuntant cap al terra ni a cap objecte, ja que el raig ha acabat el seu recorregut sense haver detectat res. En aquest cas s'ha de generar un nou raig des de la posició on s'ha quedat l'anterior fins al terra, per tal de previsualitzar l'edifici a la distància màxima però sobre el terreny. La Figura 7.29 mostra en blau els rajos llençats des de la càmera i en verd els rajos llençats des de la posició final fins al terra, i la Figura 7.30 mostra el codi que a partir del punt final del primer raig genera un nou *Raycast* fins al terra i retorna la posició i la normal del punt de col·lisió.

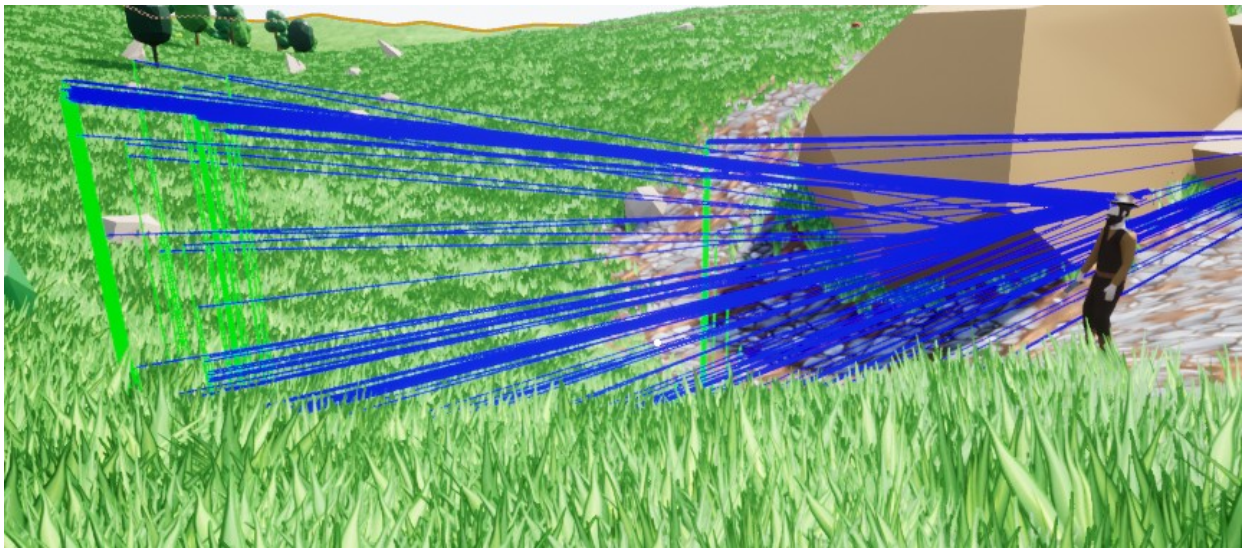


Figura 7.29: Buscar el terra en cas de no haver detectat cap objecte

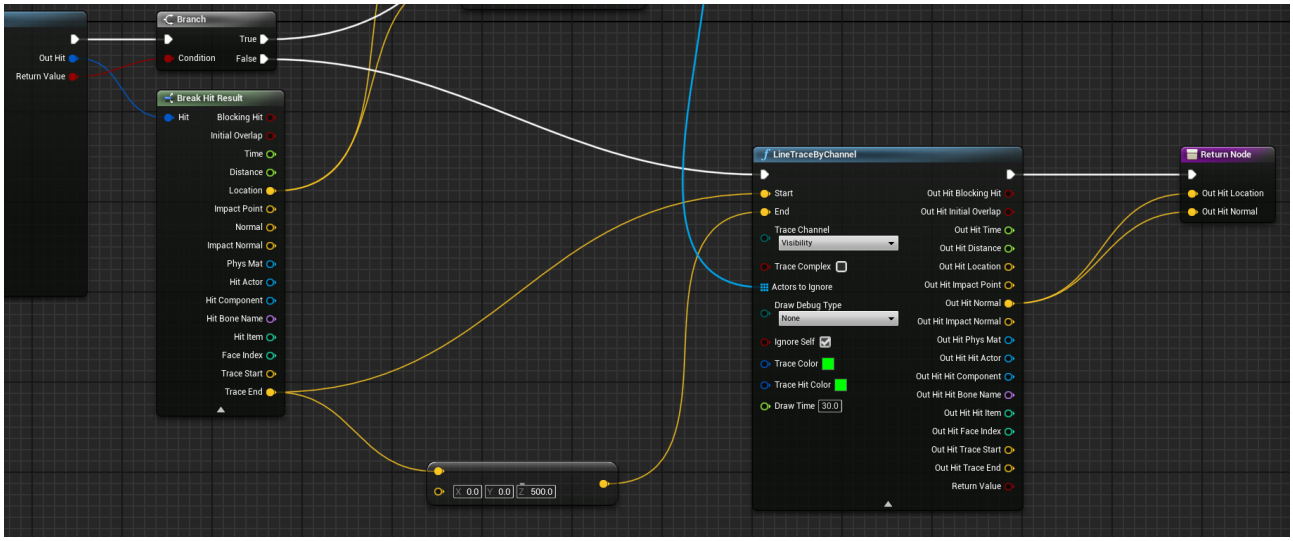


Figura 7.30: Generació del segon raig en cas de que no s'hagi detectat cap objecte

La segona situació es dona quan el primer raig detecta un objecte. En aquest cas el què s'ha de buscar és el punt superior on s'acaba aquest objecte per evitar que l'edifici es previsualitzi a sobre d'un altre objecte. La Figura 7.32 mostra en blau els rajos llençats

des de la càmera i en groc el punt superior de l'objecte detectat, i la Figura 7.31 mostra el codi que genera el segon raig a partir del punt on ha impactat el primer i retorna la posició i la normal del punt de col·lisió.

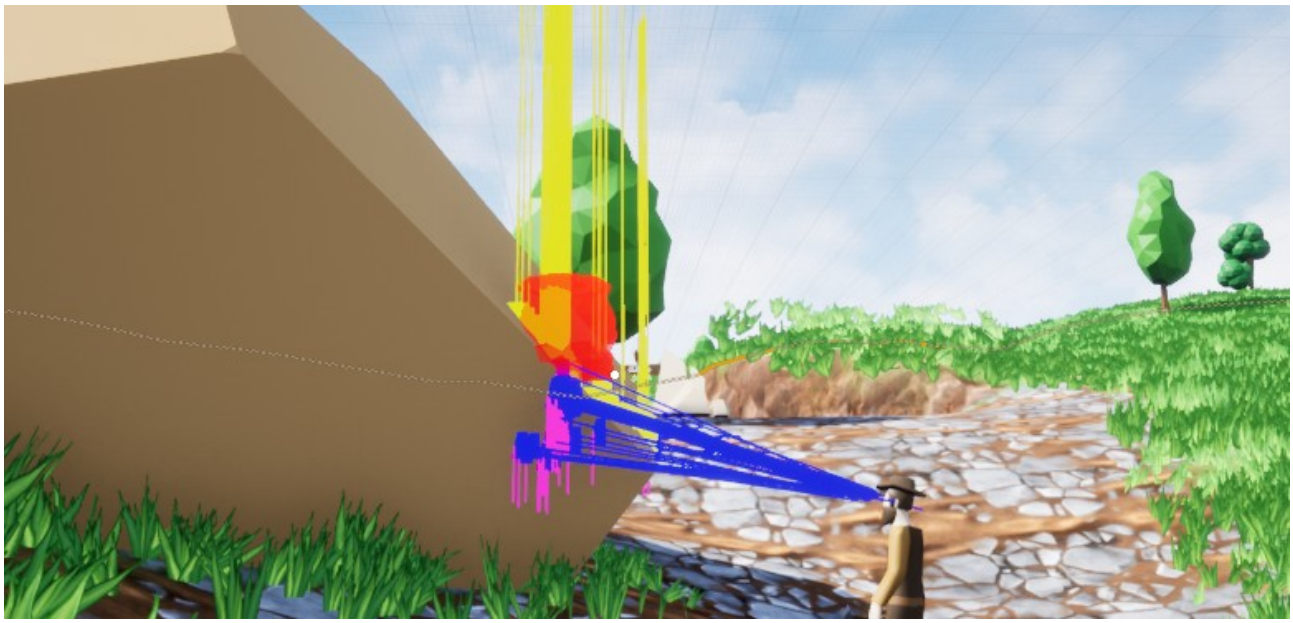


Figura 7.32: Buscar la part superior en cas d'haver detectat un objecte

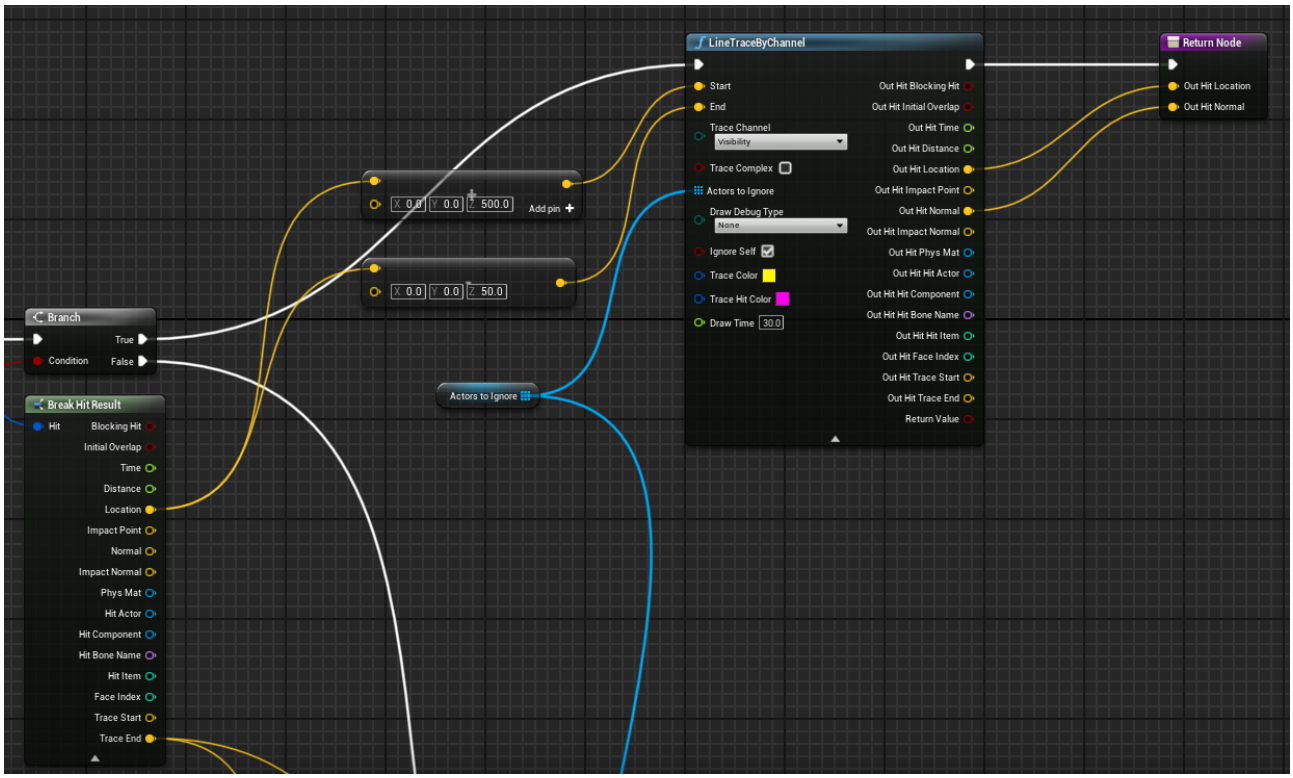


Figura 7.31: Generació del segon raig en cas d'haver detectat algun objecte

Si en qualsevol de les dues situacions el segon raig no col·lisiona amb res l'edifici a previsualitzar no es mostra. La Figura 7.33 mostra la implementació completa de la funció `TraceBuildingPoint()`.

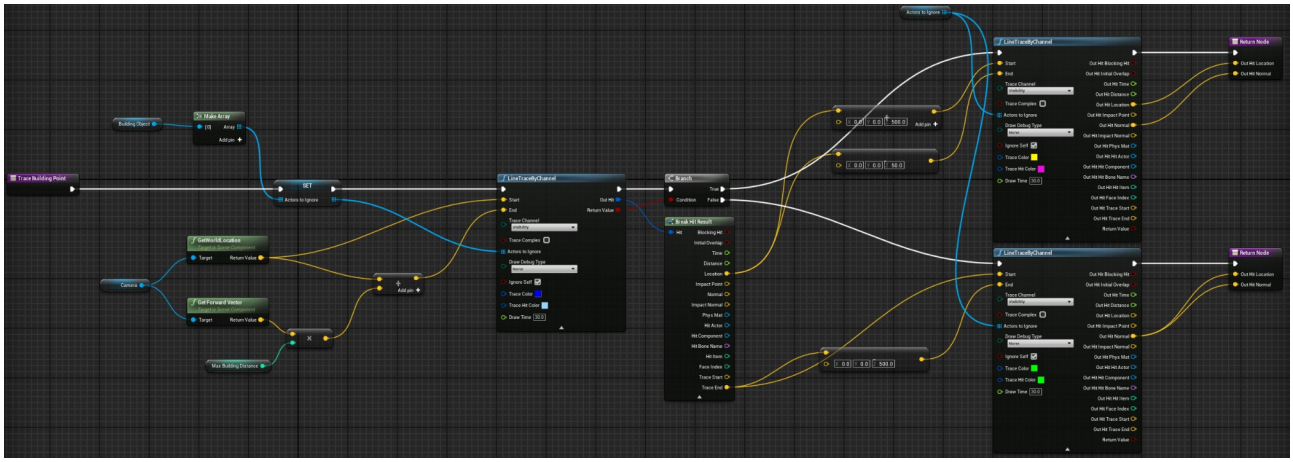


Figura 7.33: Implementació de la funció TraceBuildingPoint

A partir del resultat d'aquesta funció es pot determinar la posició de l'edifici de previsualització. Per fer-ho s'ha utilitzat la funció de la classe Actor `SetActorLocation()`.

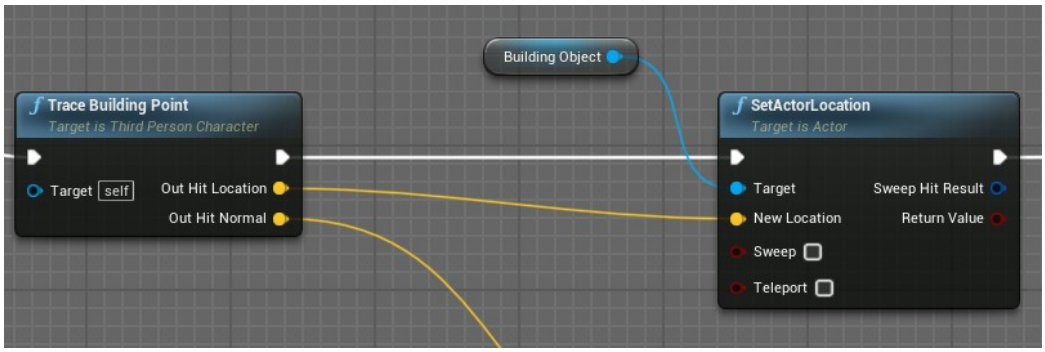


Figura 7.34: Actualitzar la posició de l'edifici a previsualitzar

Un cop col·locat l'edifici només falta assignar-li el material que indiqui si es pot o no construir. Els paràmetres que s'han tingut en compte a l'hora de determinar si es pot o no construir un edifici han estat que el jugador tingui els recursos suficients per a poder-lo construir, i que la zona on vol col·locar-lo no tingui molta pendent. Per saber si el jugador té suficients recursos s'ha utilitzat el flag *EnoughResources,* que s'assigna des de la interfície de l'edifici quan es comproven els ítems necessaris, i per a saber la pendent del terreny al punt de construcció s'ha utilitzat el valor de la normal al punt de col·lisió i se li ha aplicat un threshold. La Figura 7.35 mostra la implementació d'aquesta part.

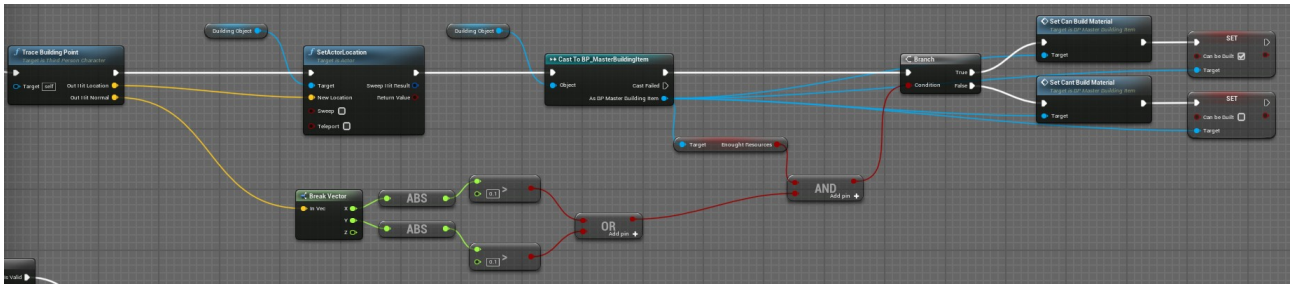


Figura 7.35: Actualitzar el material segons si es pot construir o no

Per acabar amb l'esdeveniment `PreviewBuilding` s'ha de controlar que passa en el moment en què el jugador deixa de seleccionar el martell i es desactiva el mode de construir. Quan es desactiva aquest mode s'ha de destruir l'edifici de previsualització i la seva interfície. A més a més, s'ha afegit un esdeveniment personalitzat anomenat `ChangeSelectedBuilding` que permet treure un edifici de la pantalla de forma controlada. Aquest esdeveniment utilitzat juntament amb el `ResetBuilding` explicat anteriorment permet eliminar la previsualització antiga i instanciar la nova, en cas que el jugador canvi de selecció. La Figura 7.36 mostra la implementació d'aquest esdeveniment.

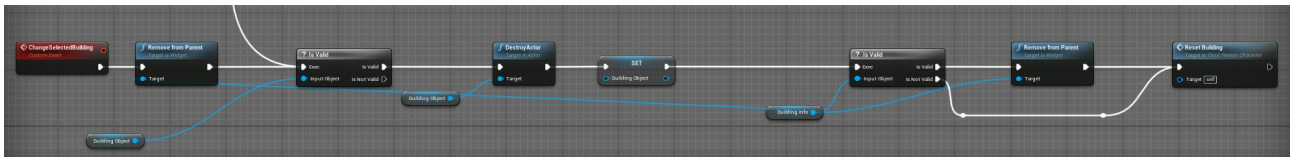


Figura 7.36: Implementació de l'esdeveniment que elimina la previsualització

La Figura 7.37 mostra la implementació de l'esdeveniment `PreviewBuilding` complet.

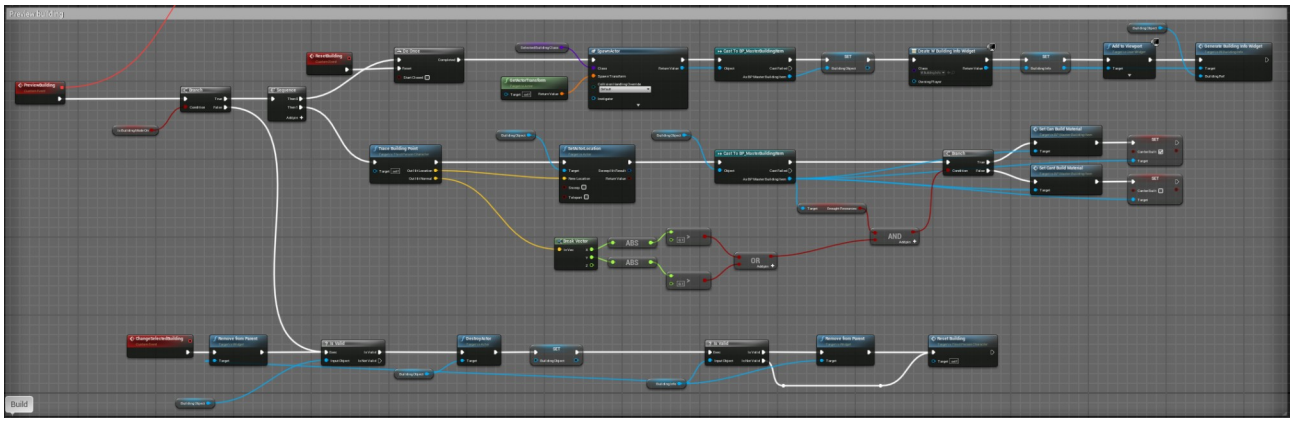


Figura 7.37: Implementació de l'esdeveniment PreviewBuilding

Per a poder construir els edificis s'ha implementat l'esdeveniment `Build`, que es crida quan el jugador utilitza el martell. Aquest esdeveniment comprova que l'edifici es pugui

construir i s'encarrega de col·locar-lo de forma definitiva i actualitzar tots els sistemes afectats per la construcció. Aquest esdeveniment es pot dividir en tres blocs diferents:

- Treure els ítems utilitzats en la construcció de l'inventari del jugador

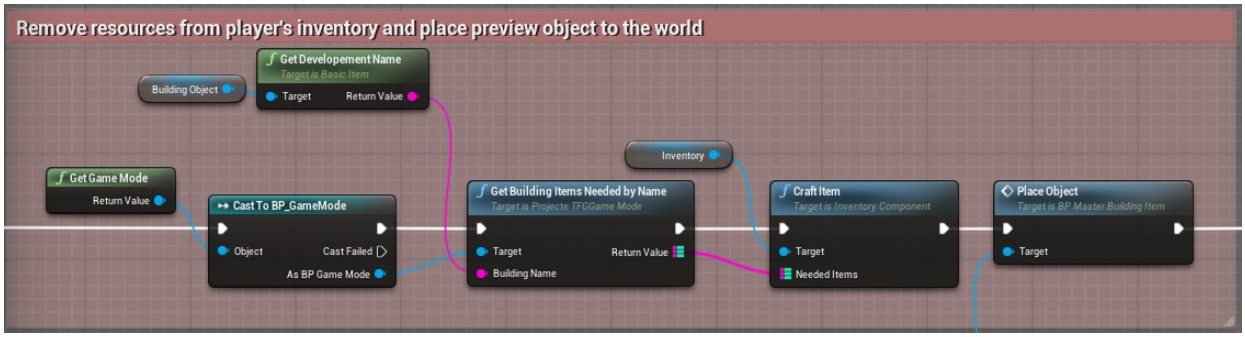


Figura 7.38: Treure els ítems de l'inventari del jugador

- Registrar l'edifici construït al *GameMode* i comprovar si s'ha completat algun objectiu

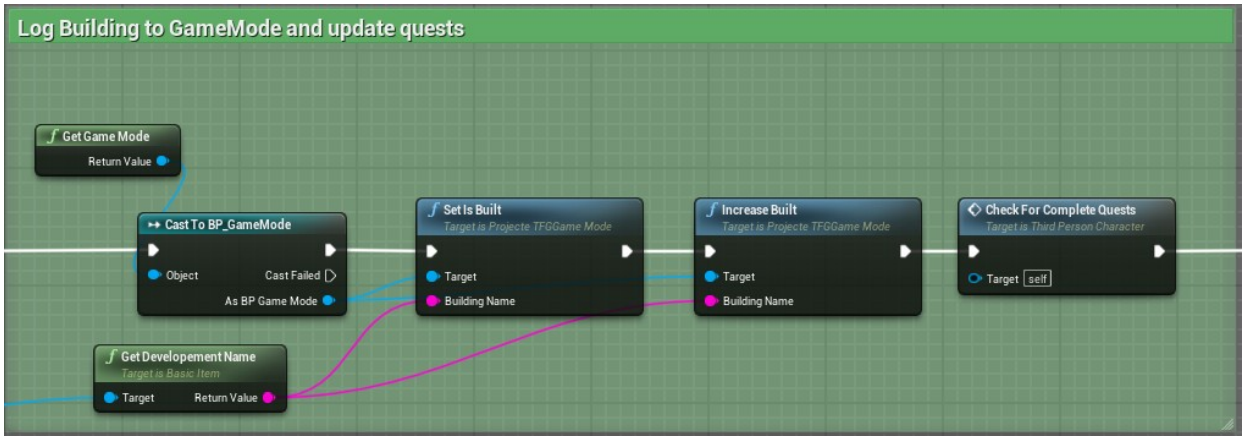


Figura 7.39: Registrar l'edifici al GameMode i actualitzar els objectius

- Reinicialitzar l'estat del sistema de construcció

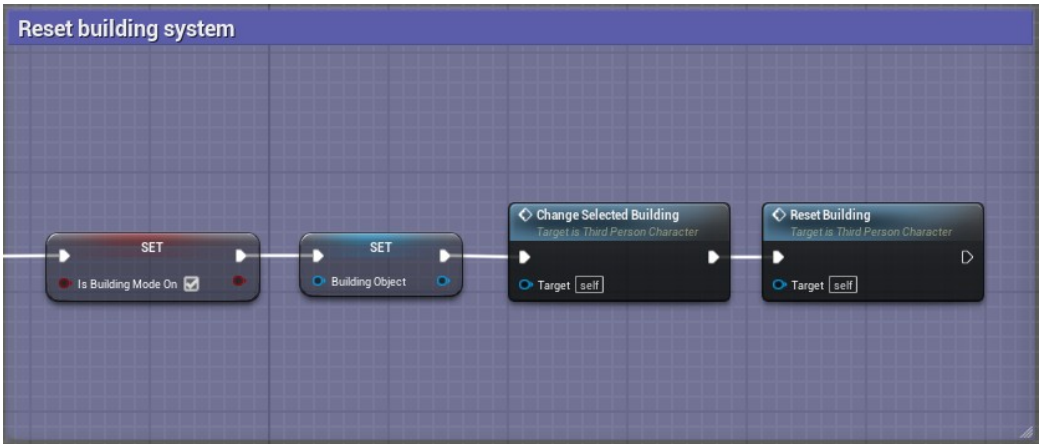


Figura 7.40: Reiniciar el sistema de construcció

La principal diferència del mode plantar respecte a la implementació vista són les condicions que determinen si l'objecte es pot col·locar o no. En els edificis es té en compte la pendent i els recursos del jugador, però per les plantes fa falta saber també si la zona on s'han de plantar és cultivable o no. Les zones cultivables venen determinades per una textura que representa l'escenari projectat des de dalt, i on les parts negres són zones no cultivables i les parts blanques són zones cultivables, com es pot veure a la Figura 7.41.

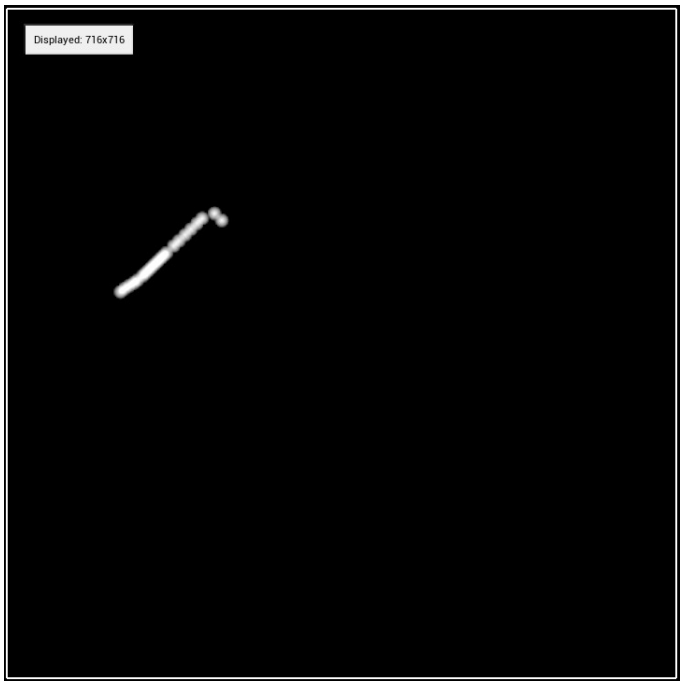


Figura 7.41: Textura que determina les zones cultivables

Per determinar si el jugador vol plantar en una zona cultivable s'ha de comprovar que el punt on s'ha de col·locar l'objecte estigui en un pixel blanc d'aquesta textura. Per saber-ho s'ha utilitzat la funció del motor `ReadRenderTargetPixel()`. Aquesta funció rep les coordenades X i Y d'un pixel i la textura sobre la qual determinar el color, i retorna un vector de quatre dimensions amb els valors de color RGBA del pixel. Des de la documentació recomanen utilitzar aquesta funció amb cura, ja que és molt poc eficient.

Per calcular el valor del pixel s'ha de convertir la posició on s'ha de plantar a coordenades de la textura. Aquesta conversió es fa dividint les components X, Y i Z de la posició per la mida del mapa, per obtenir el punt on es projecta aquesta posició i per convertir-lo a coordenades de textura s'ha de multiplicar la X i Y resultants per la mida de la textura. Finalment, es compara el color que retorna la funció `ReadRenderTargetPixel()` amb el color blanc (1.0, 1.0, 1.0, 1.0) i si són iguals vol dir que el jugador vol plantar en una

zona cultivable. La Figura 7.42 mostra el fragment de l'esdeveniment `PreveiwPlanting` que implementa aquestes comprovacions.

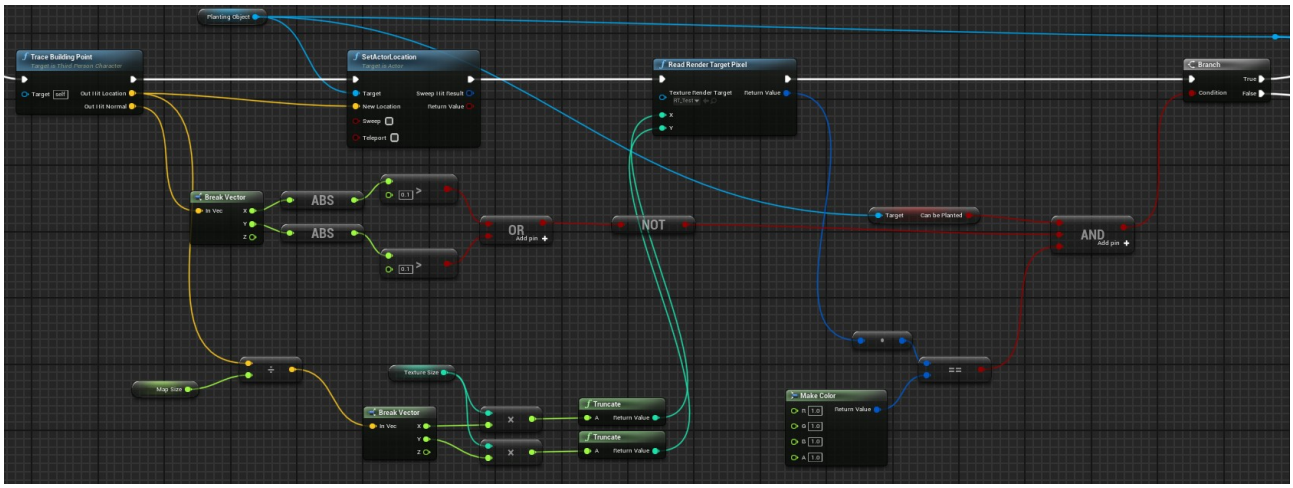


Figura 7.42: Comprovacions per determinar si es pot plantar al punt detectat

7.2.3. Animacions

Per a controlar les animacions del personatge, s'ha implementat una *Animation Blueprint*. Aquesta Blueprint s'encarrega de gestionar tant les animacions bàsiques del personatge, com el moviment, i també s'encarrega de gestionar els esdeveniments generats per les *Animation Notifications*. Les animacions que s'han utilitzat al projecte s'han obtingut de la plataforma d'Adobe Mixamo [6].

Les Animation Blueprints estan dividides en dos parts: `EventGraph`, on es programa la lògica de les animacions, i `AnimGraph`, on es controla quines animacions es mostren. Per poder fer transicions entre animacions fa falta tenir informació com la velocitat o la posició del jugador, per poder escollir quina animació s'ha de reproduir segons la situació. Per a obtenir aquesta informació Unreal Engine 4 disposa d'un esdeveniment anomenat `BlueprintUpdateAnimation()`, que s'executa cada frame, i és ideal per a accedir a la informació actualitzada dels objectes. Per poder controlar les animacions d'aquest projecte fa falta saber la velocitat i la direcció del jugador, i si està tocant a terra o no. Per simplificar les tasques dels desenvolupadors el component `CharacterMovment()` implementa una funció que retorna cert o fals segons si el personatge està tocant alguna superfície o no.

Una bona pràctica per accedir a referències d'objectes des de d'una Blueprint és fer-ho a l'esdeveniment `BeginPlay()`, que s'executa quan es crea l'objecte, i guardar la

referència per poder accedir de forma ràpida des de qualsevol altre esdeveniment o funció. A més a més, aquesta tècnica també suposa un avantatge en cas d'haver d'accedir a la referència cada frame. En aquest cas, s'ha accedit a la referència del jugador des de l'esdeveniment `BeginPlay()`, i s'ha guardat com un atribut `Player`, permetent que l'esdeveniment `BlueprintUpdateAnimation()` pugui accedir a la informació del jugador a través d'aquest. La Figura 7.43 mostra la implementació dels esdeveniments `BeginPlay()` i `BlueprintUpdateAnimation()`, que emmagatzema en atributs els valors de la velocitat, la posició i si està a l'aire del jugador.

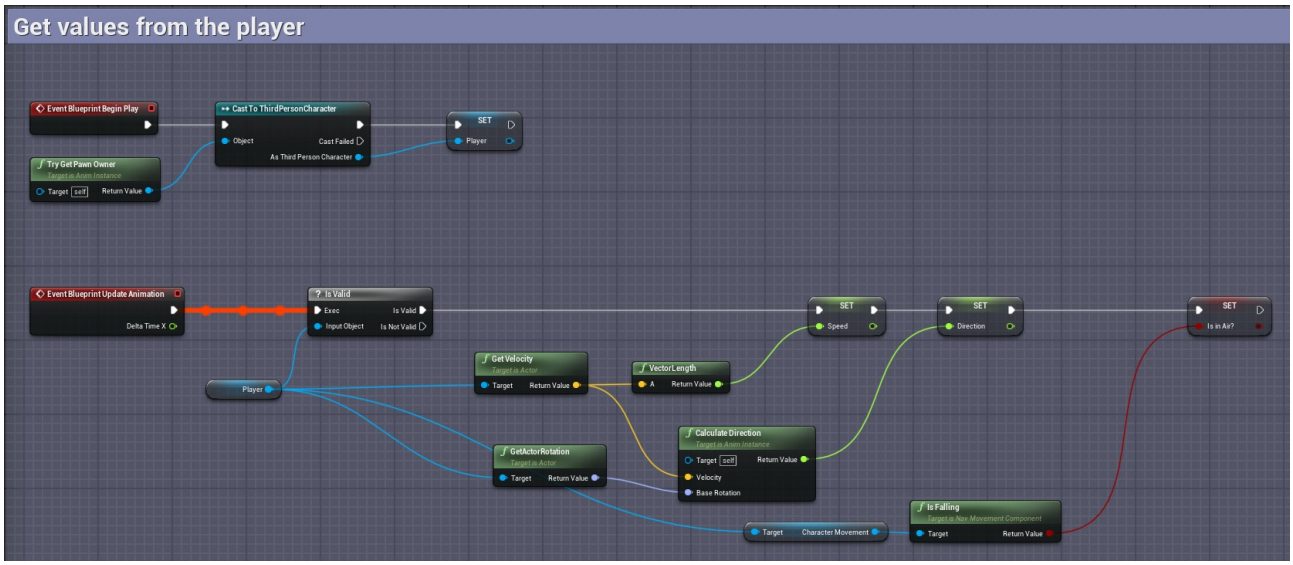


Figura 7.43: Implementació dels esdeveniments BeginPlay i BlueprintUpdateAnimation

7.2.3.1. Màquina d'estats

Les màquines d'estats permeten representar les animacions d'una *Skeletal Mesh* a partir d'una sèrie d'estats que estan connectats a través de regles de transició, que s'encarreguen de definir quan s'han de produir canvis d'estat i entre quins estats. Les animacions del moviment del jugador s'han implementat a partir d'una màquina d'estats definida a *l'AnimationBlueprint*, on cada estat representa una animació. Els estats en els que pot estar el jugador són:

- Idle: El jugador no s'està movent
- JumpStart: El jugador ha començat a saltar
- JumpLoop: El jugador encara està a l'aire
- JumpEnd: El jugador torna a toca a terra
- Walk/Run: El jugador s'està movent

- WalkingJump: El jugador salta mentre es mou

La Figura 7.44 mostra aquests estats i les relacions que els uneixen dintre de la màquina d'estats.

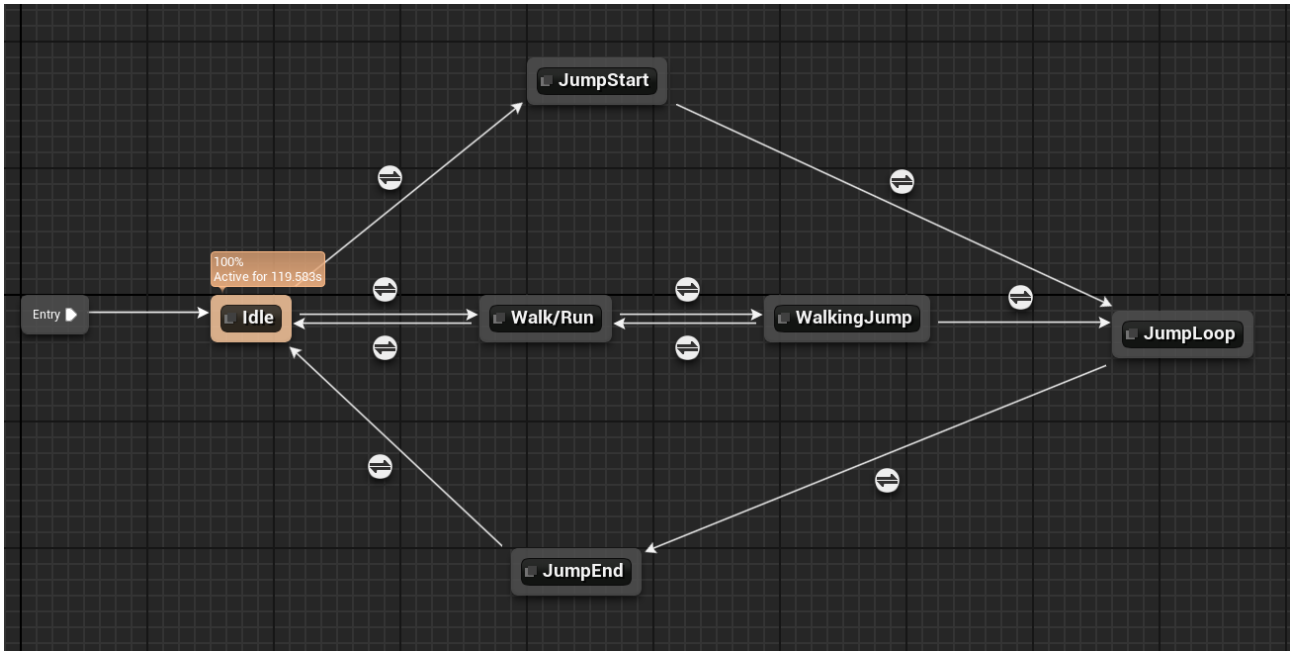


Figura 7.44: Màquina d'estats del personatge

7.2.3.2. Animacions i eines

A l'incorporar les animacions que representen la utilització de les eines, va sorgir un problema de sincronització, ja que el codi s'executava o abans de començar o després de completar l'animació, traient molta immersió al joc, ja que la resposta de l'escenari a les accions del jugador no coincidia amb el moment en què l'eina interactuava amb aquest.

Per solucionar-ho s'han utilitzat *Animation Notifications*, que, com ja s'ha explicat, és una eina d'Unreal Engine 4 que permet disparar esdeveniments durant una animació. Aquests esdeveniments s'han de programar des de `EventGraph` de la `AnimationBlueprint`.

S'ha creat un esdeveniment per cadascuna de les tres eines que tenen una animació al ser utilitzades. Mitjançant aquesta solució, el jugador, enlloc d'executar el codi relacionat amb utilitzar les eines, reprodueix l'animació corresponent, i és des d'aquesta animació on es dispara l'esdeveniment que executa el codi d'utilitzar les eines. La Figura 7.45 mostra la implementació dels esdeveniments `useAxe()`, `usePickaxe()` i `useHoe()`, i com es pot observar,

cada esdeveniment crida la funció `UseTool()` i passa com a paràmetre el nom de l'eina que ha activat l'esdeveniment. L'esdeveniment `useHoe()` també s'encarrega de reproduir el so d'utilitzar l'aixada.

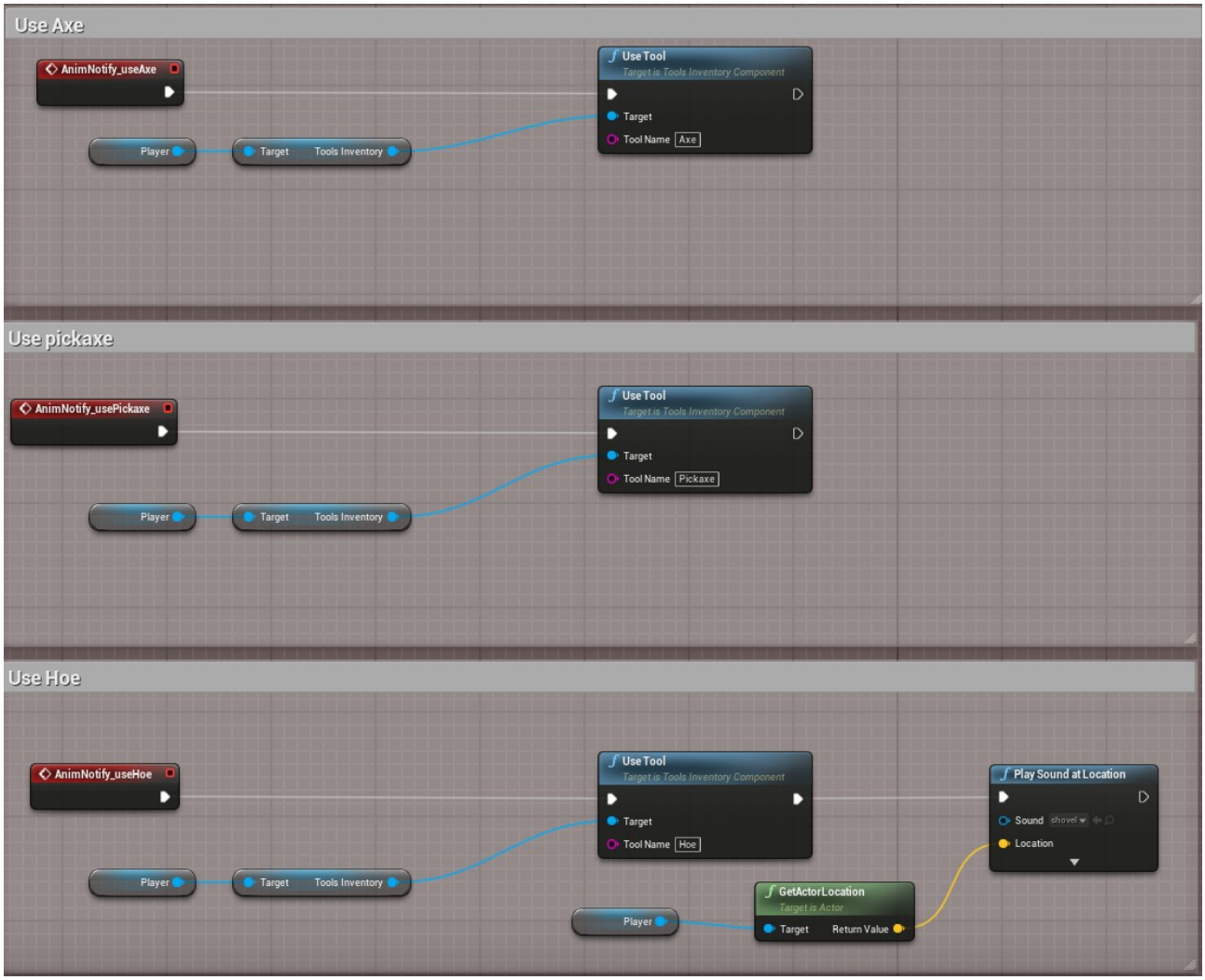


Figura 7.45: Implementació dels esdeveniments useAxe, usePickaxe i useHoe

Finalment s'han de relacionar aquests esdeveniments amb el frame de l'animació que els ha d'activar. Per fer-ho s'ha d'afegir un *Notify* del tipus corresponent al frame de la línia temporal de l'animació que escollim. La Figura 7.46 mostra l'animació d'utilitzar la destral amb la crida a l'esdeveniment corresponent.

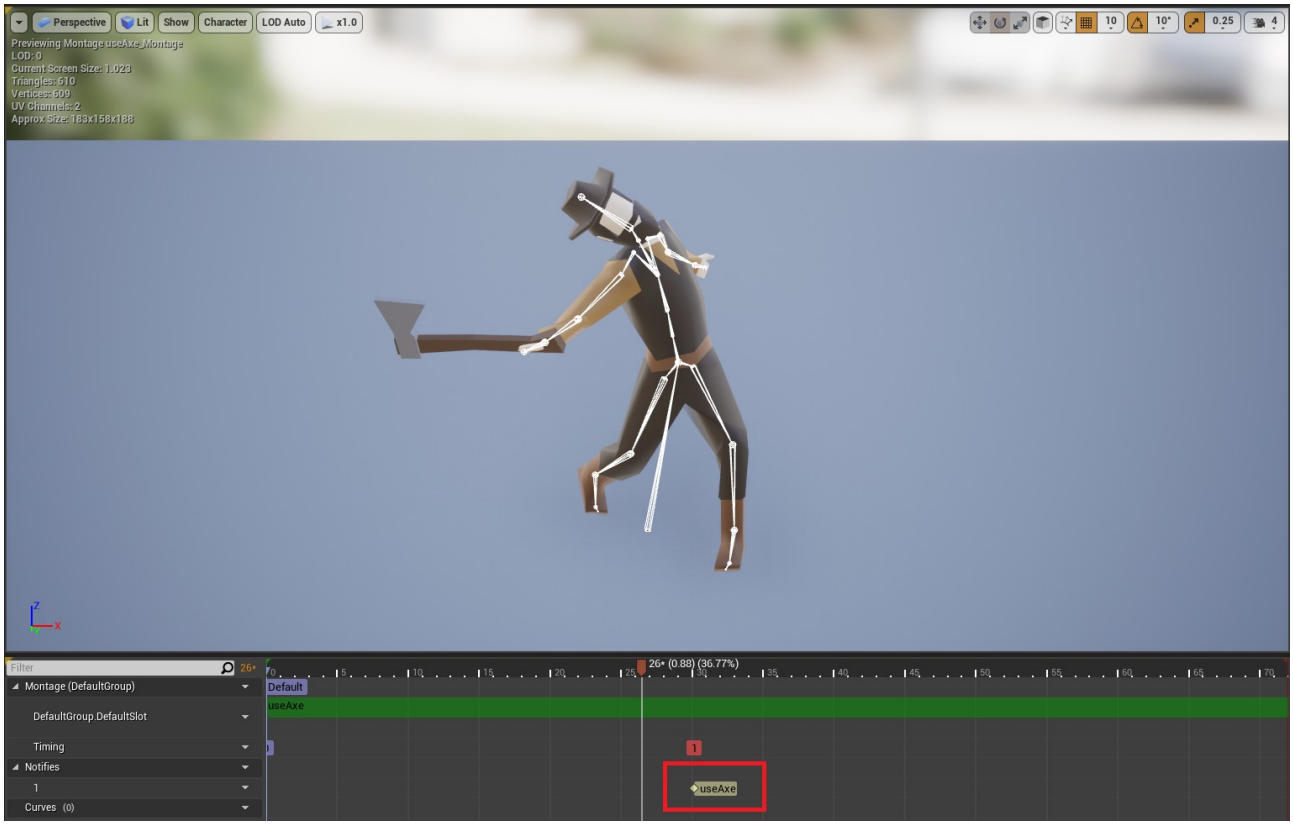


Figura 7.46: Animació d'utilitzar la destral i Notify useAxe()

7.2.3.3. Animacions i sons

Una forma de donar més immersió al joc és a partir de sons que indiquin que el jugador s'està movent, com per exemple els passos. Per implementar un sistema que reprodueixi un so cada cop que el jugador posa el peu a terra es pot utilitzar la idea de les notificacions, tal i com s'ha fet amb les eines. Aquesta solució, tot i que incrementa la immersió, encara es podria millorar més, ja que quan una persona es mou els sons de les seves passes canvien segons quina superfície estigui trepitjant. Com que en aquest projecte s'ha treballat amb un terreny format per tres capes, herba, terra i pedra, s'ha decidit implementar un sistema de sons de passes basat en notificacions però que tingui en compte la superfície on es troba el jugador per reproduir sons diferents.

Un altre aspecte important a l'hora de dissenyar passes immersives és que al caminar no es fan dues passes iguals, encara que s'estigui sobre la mateixa superfície. Per a simular aquesta variabilitat de sons s'ha treballat amb 5 o 6 sons diferents per cada superfície, de forma que cada cop que el jugador trepitja el terra s'escull aleatòriament un d'aquests sons segons la superfície.

Per implementar la selecció de sons aleatoris s'han utilitzat *Sound Cues*, que són uns assets que permeten encapsular funcionalitats relacionades amb el disseny d'àudio

mitjançant un editor de nodes similar al dels materials, però amb funcions de so. Per cada superfície s'ha creat un *Sound Cue* al qual s'han importat els diferents sons de passes sobre la superfície, i mitjançant un node `random()` s'ha aconseguit que cada cop que es reprodueixi el *Sound Cue* soni aleatòriament algun dels sons connectats. La Figura 7.47 mostra el Sound Cue de la superfície d'herba.

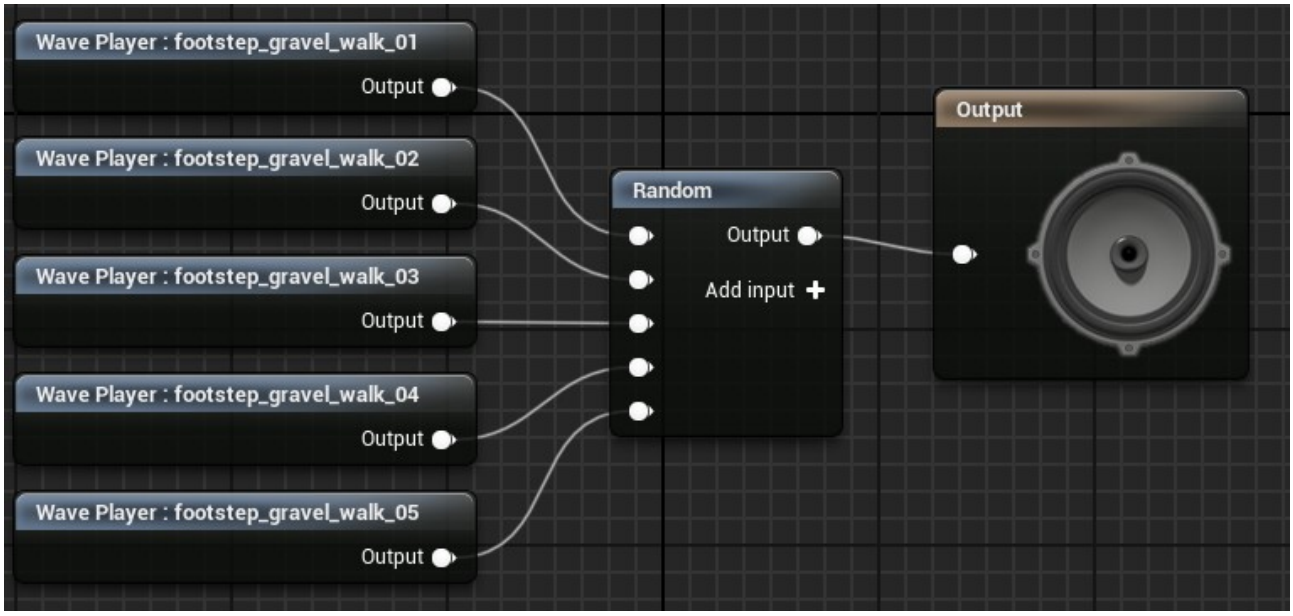


Figura 7.47: Sound Cue amb els sons dels passos a l'herba

Per determinar el tipus de superfície del terreny s'ha utilitzat el motor de físiques, concretament les *Physical Surfaces* i els *Physical Materials.* Mitjançant aquestes superfícies es poden crear capes i vincular-les a *Physical Materials.* Aquests materials determinen la resposta d'un objecte quan interactua amb l'escenari. La Figura 7.48 mostra les capes que s'han creat i com es vinculen a un material.

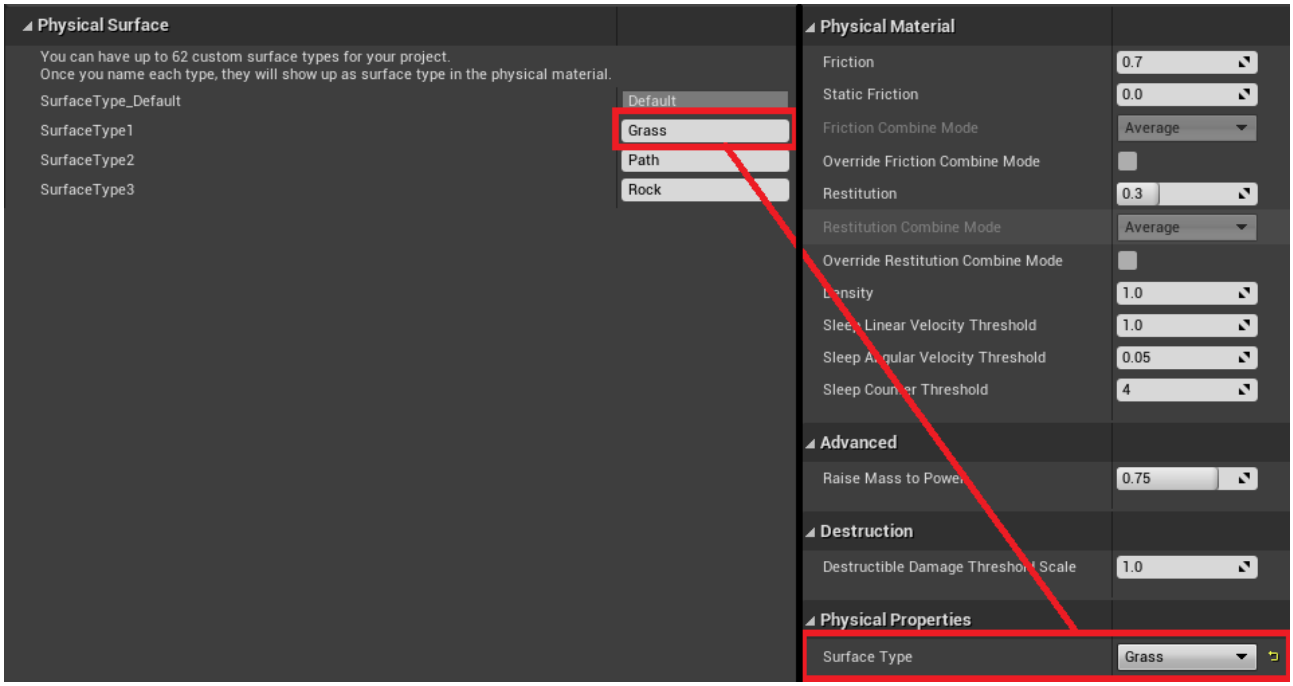


Figura 7.48: Superfícies definides per al projecte i exemple de vinculació d'una capa a un Physical Material

Unreal Engine 4 permet mostrar els terrenys a partir de capes de textures, de forma que al aplicar els *Physical Materials* de cada tipus de superfície a la capa de la textura que els representa es pot diferenciar entre zones. La Figura 7.49 mostra com es vincula el *Physical Material* de la capa d'herba amb la capa de textura.

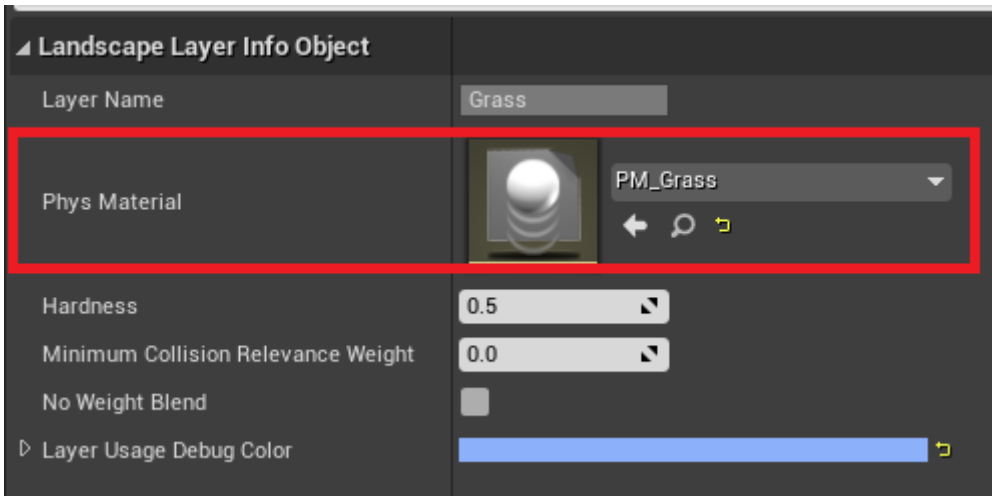


Figura 7.49: Assignació d'un Physics Material a una capa de textura

Per detectar sobre quin tipus de superfície es troba el jugador també s'ha utilitzat la tècnica del *Raycast*, però en aquest cas projectant un raig vertical des de la posició del jugador fins al terra. Per determinar la superfície s'ha accedit al camp `PhysMat` de l'estructura `HitResult` explicada anteriorment, i mitjançant la funció `GetSurfaceType()` s'ha obtingut el tipus de superfície a la posició del jugador.

Finalment, mitjançant un *Select* s'escull el *Sound Cue* de la superfície corresponent. La Figura 7.50 mostra la implementació de l'esdeveniment `Footstep()` que es dispara des de les animacions mitjançant notificacions.

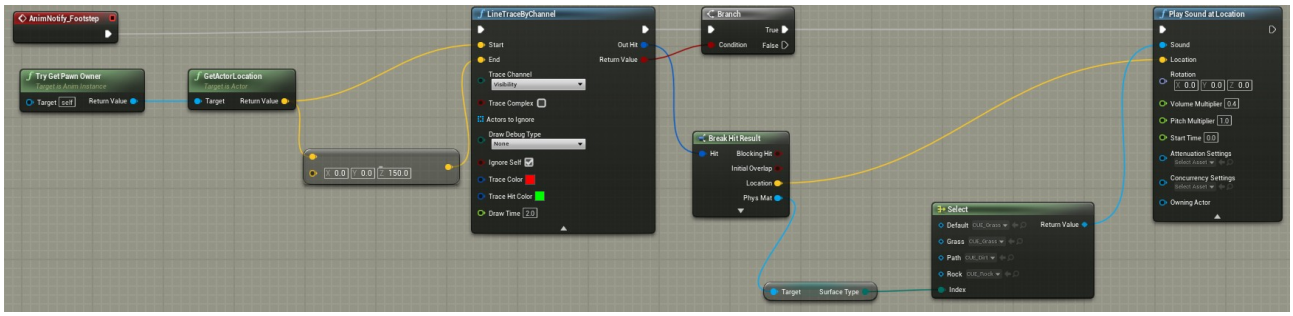


Figura 7.50: Implementació de l'esdeveniment Footstep

Per a acabar de polir l'apartat de sons del jugador, s'ha decidit implementar un so quan s'impulsa per a saltar, creant l'esdeveniment `JumpStart()` el qual reprodueix un *Sound Cue* amb diferents sons de salts aleatoris. La Figura 7.51 mostra la implementació de l'esdeveniment `JumpStart`.

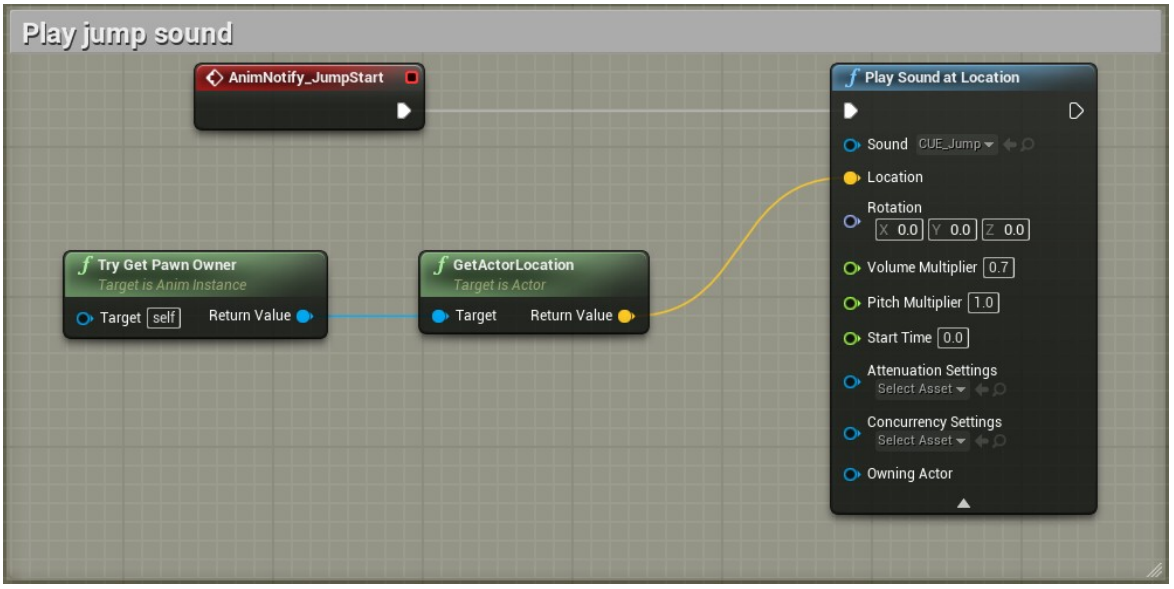


Figura 7.51: Implementació de l'esdeveniment JumpStart

7.2.4. Moviment

El moviment del personatge s'ha definit a partir del component `CharacterMovement`, implementat per defecte dintre del motor. Aquest component determina totes les variables necessàries per a controlar el jugador, com la velocitat, la rotació, etc, i implementa una sèrie de funcions vinculades als inputs mappings que permeten moure el jugador. S'ha decidit utilitzar aquest component perquè, per defecte, implementa el control d'un personatge humanoide amb els inputs estàndard dels jocs en 1ª i 3ª persona.

Aquest component només implementa les funcionalitats de caminar i saltar, així que s'ha decidit implementar un sistema de sprint per permetre als jugadors desplaçar-se de forma més ràpida pel mapa. Aquesta funcionalitat s'ha implementat en Blueprints des del `ThirdPersonCharacter`.

Per defecte el component de moviment determina que la velocitat màxima del jugador és de 300 unitats, el que equival a caminar, i per aquest projecte s'ha considerat la velocitat màxima de sprint com a 600 unitats. La solució més simple per al sistema de sprint és modificar a 600 el valor de la velocitat màxima quan el jugador prem la tecla vinculada a esprintar, i reinicialitzar-lo a 300 quan el jugador deixa la tecla. Tot i que aquesta solució funciona, suposa una transició molt brusca entre córrer i caminar, quan en realitat la velocitat màxima s'aconsegueix després d'una petita acceleració. Per a simular aquesta acceleració s'ha utilitzat un *TimeLine,* que és un tipus de node que permet definir ràpidament animacions basades en temps. Aquest TimeLine dura un segon i passa de forma lineal de 0 a 1. La Figura 7.52 mostra la funció implementada al TimeLine.

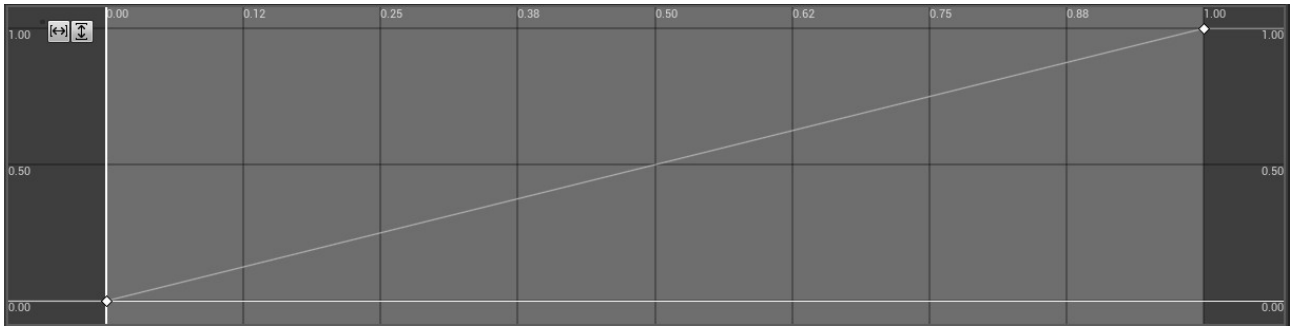


Figura 7.52: Sprint Timeline

El resultat d'aquesta funció, durant el segon que dura, s'utilitza per a interpolar el valor de la velocitat entre el mínim i el màxim. La interpolació es fa a través de la funció del motor `Lerp()` que implementa una interpolació lineal. La Figura 7.53 mostra el codi que implementa aquesta funcionalitat.

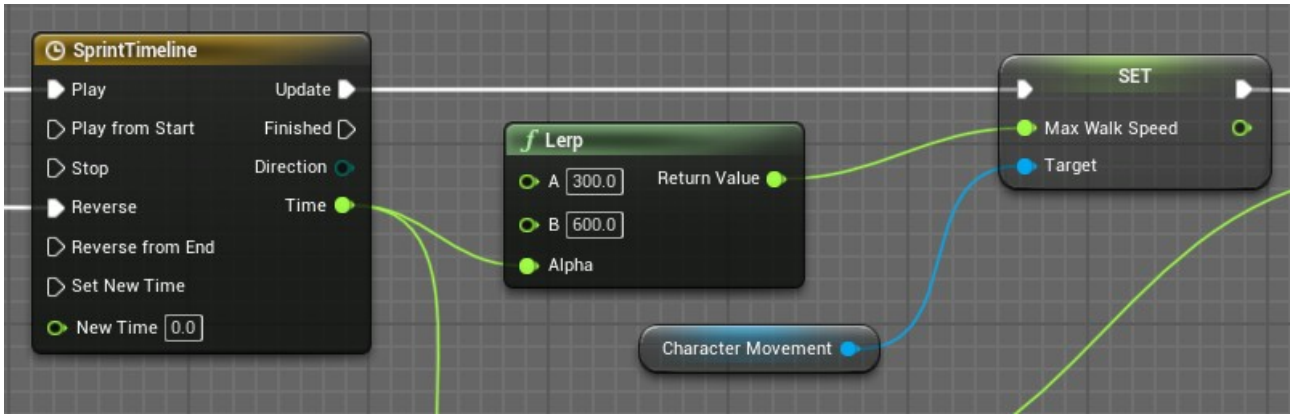


Figura 7.53: Interpolació entre la velocitat mínima i màxima a partir de la funció definida al Timeline

Per polir encara més el sistema de sprint, també s'ha interpolat de la mateixa manera el Field Of View de la càmera, de forma que quan el jugador esprinta s'amplia una mica la visió del jugador. La Figura 7.54 mostra el sistema d'esprintar complet, i es pot veure com quan el jugador interactua amb la tecla de córrer, si s'està movent (no te sentit esprintar quiet), s'activa aquest Timeline a partir del qual s'interpolen i assignen els dos valors mencionats. Finalment, quan el jugador deixa la tecla es reprodueix el Timeline del revés per recuperar els valors inicials mantenint les animacions definides.

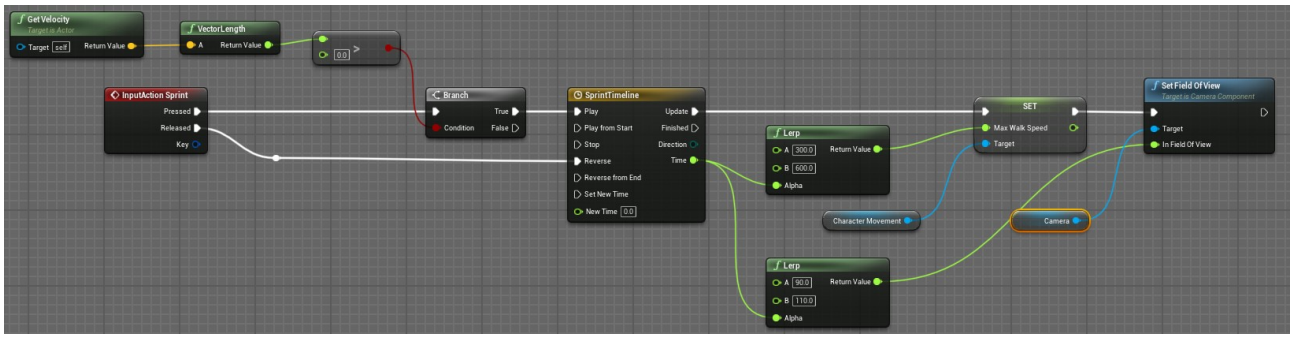


Figura 7.54: Implementació de l'esdeveniment InputActionSprint

7.3. GameMode

El GameMode és un tipus d'objecte que defineix les regles del joc. En aquest projecte s'ha utilitzat per portar un registre de tots els objectes que poden haver a l'escenari, com recursos, receptes, edificis...

Al començar la partida es creen tres Maps on s'emmagatzema una referencia a cada ítem, recepta i edifici. També s'implementen tots els mètodes necessaris per a poder treballar des dels diferents sistemes del joc amb aquestes dades. El GameMode s'ha implementat a la classe *TFGGameMode,* i es pot dividir en tres grans blocs:

Ítems: El següent fragment de codi mostra la definició dels atributs i mètodes encarregats d'emmagatzemar els ítems.

```
// Item references
UPROPERTY()
TMap<FString, FitemStruct> itemReferences;

UFUNCTION(BlueprintCallable)
void addItemReference(ABasicItem* item);

UFUNCTION(BlueprintCallable)
ABasicItem* GetItemReference(FString item);

UFUNCTION(BlueprintCallable)
void logItemReferences() const;
```

Receptes: El següent fragment de codi mostra la definició dels atributs i mètodes encarregats d'emmagatzemar les receptes.

```
// Recepies
UPROPERTY()
TMap<FString, FRecipeStruct> recipesSet;

UFUNCTION(BlueprintCallable)
void addRecipe(ACraftingItem* recipe);

UFUNCTION(BlueprintCallable)
TArray<FString> GetRecipesBlueprintType(int typeOfRecipes) const;

UFUNCTION(BlueprintCallable)
TMap<FString, int> GetRecipieItemsNeededByName(FString recipeName) ;

UFUNCTION(BlueprintCallable)
ACraftingItem* GetRecipeReferenceByName(FString recipeName);

UFUNCTION(BlueprintCallable)
TSubclassOf<ACraftingItem> GetRecipeClassByName(FString recipeName);

UFUNCTION(BlueprintCallable)
void logRecipeReferences() const;
```

Edificis: El següent fragment de codi mostra la definició dels atributs i mètodes encarregats d'emmagatzemar els edificis.

```
// Buildings
UPROPERTY()
TMap<FString, FBuildingStruct> buildingReferences;

UFUNCTION(BlueprintCallable)
void addBuildingReference(ABuildingItem* building);

UFUNCTION(BlueprintCallable)
ABuildingItem* GetBuildingReference(FString building);

UFUNCTION(BlueprintCallable)
void logBuildingReferences() const;

UFUNCTION(BlueprintCallable)
TArray<FString> GetBuildingsBlueprintType(int typeOfBuildings) const;

UFUNCTION(BlueprintCallable)
ABuildingItem* GetBuildingReferenceByName(FString buildingName);

UFUNCTION(BlueprintCallable)
TSubclassOf<ABuildingItem> GetBuildingClassByName(FString buildingName);

UFUNCTION(BlueprintCallable)
TMap<FString, int> GetBuildingItemsNeededByName(FString buildingName);

UFUNCTION(BlueprintCallable)
void SetIsBuilt(FString buildingName);

UFUNCTION(BlueprintCallable)
void IncreaseBuilt(FString buildingName);

UFUNCTION(BlueprintCallable)
bool GetIsBuilt(FString buildingName);

UFUNCTION(BlueprintCallable)
bool logBuildingsToFile(FString filePath) const;
```

7.4. Ítems

Els ítems són la base a partir de la qual es creen tots els elements amb els que el jugador pot interactuar. Com s'ha explicat a la part de disseny, hi ha diferents tipus d'ítems que tenen funcionalitats específiques. Per a implementar aquesta part s'ha utilitzat la herència combinada amb la metodologia utilitzada durant tot el treball de posar la lògica en C++ i els aspectes visuals en Blueprints. La Figura 7.55 mostra el diagrama de classes detallat de tots els ítems, on s'indiquen en verd les classes implementades en C++ i en blau les implementades en Blueprints. Es pot observar com els atributs i els mètodes de les classes implementades en C++ defineixen les bases del sistema, i les classes implementades en Blueprints afegeixen components com models o materials.

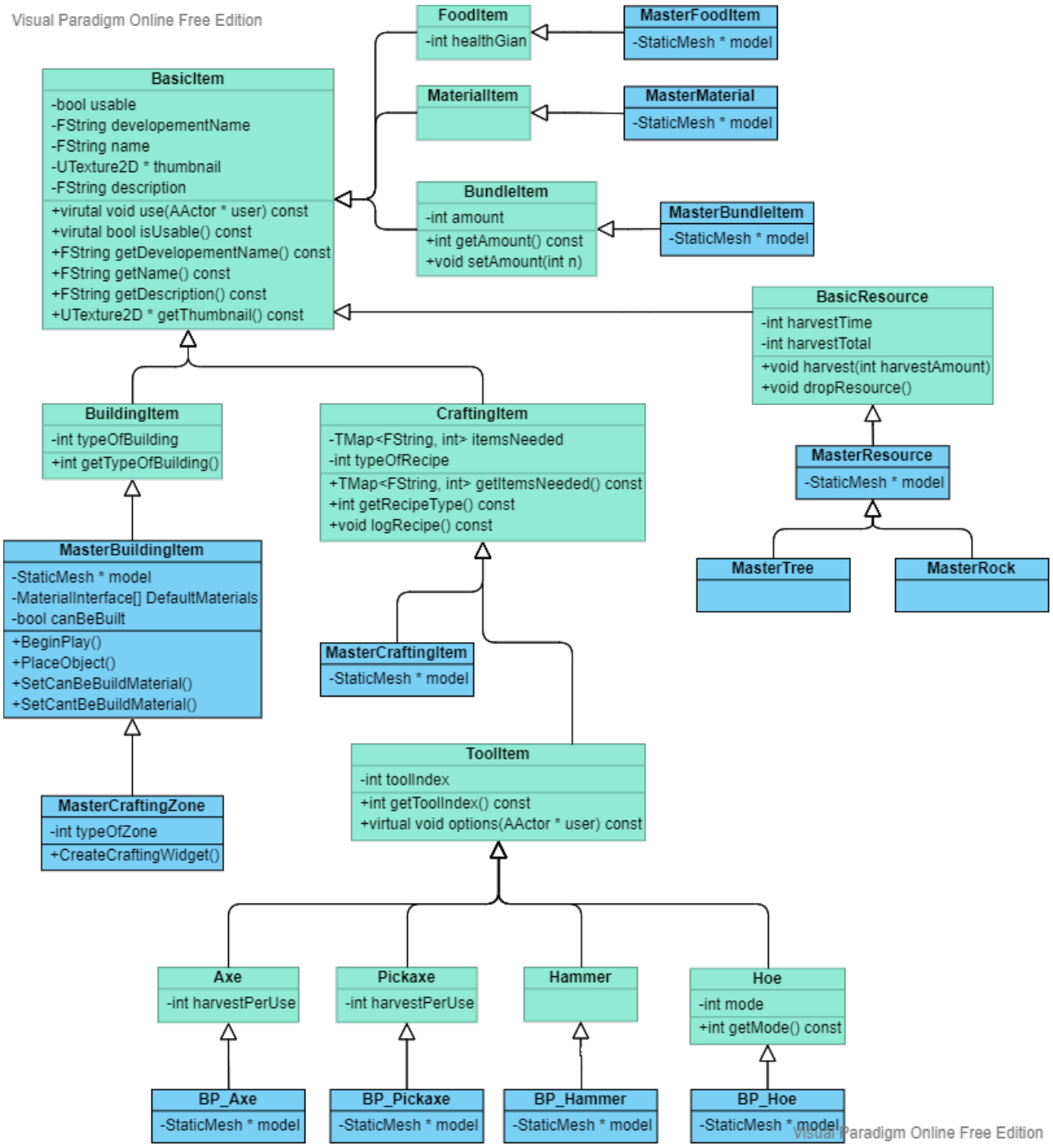


Figura 7.55: Diagrama de classes detallat amb els diferents tipus d'ítems

7.4.1. BasicItem

La classe *`BasicItem`* implementa el tipus més bàsic d'ítem, i a partir d'aquest tipus es genera tota l'estructura descrita al punt anterior. Aquí es defineixen tots els atributs i mètodes comuns entre tots els tipus d'ítems.

Un ítem bàsic està representat per dos noms, un que s'utilitza a nivell intern i un que és el que es mostra als jugadors, una breu descripció i una miniatura. Mitjançant un atribut booleà s'indica si l'ítem es pot utilitzar o no.

S'han definit dos mètodes virtuals, *`use()`* i *`isUsable()`*, que s'implementaran en les classes derivades segons les necessitats del tipus d'ítems. El mètode *`use()`* rep una referència a l'actor que utilitza l'ítem.

Al següent fragment de codi es pot veure la definició de la classe *`BasicItem`*.

```
UCLASS()
class PROJECTE_TFG_API ABasicItem : public AStaticMeshActor
{
	GENERATED_BODY()

protected:

	UPROPERTY()
	bool usable;

	UPROPERTY(EditAnywhere)
	FString developementName;

	UPROPERTY(EditAnywhere)
	FString name;

	UPROPERTY(EditAnywhere)
	UTexture2D* thumbnail;

	UPROPERTY(EditAnywhere)
	FString description;


public:
	ABasicItem();

	UFUNCTION(BlueprintCallable)
	virtual void use(AActor* user) const;

	UFUNCTION(BlueprintCallable)
	virtual bool isUsable() const;


	UFUNCTION(BlueprintCallable)
	FString GetDevelopementName() const;

	UFUNCTION(BlueprintCallable)
	FString GetName() const;

	UFUNCTION(BlueprintCallable)
	UTexture2D* GetThumbnail() const;

	UFUNCTION(BlueprintCallable)
	FString GetDescription() const;
};
```

Des de l'editor es poden modificar els paràmetres marcats com a *EditAnywhere*, com es pot veure a la Figura 7.56:

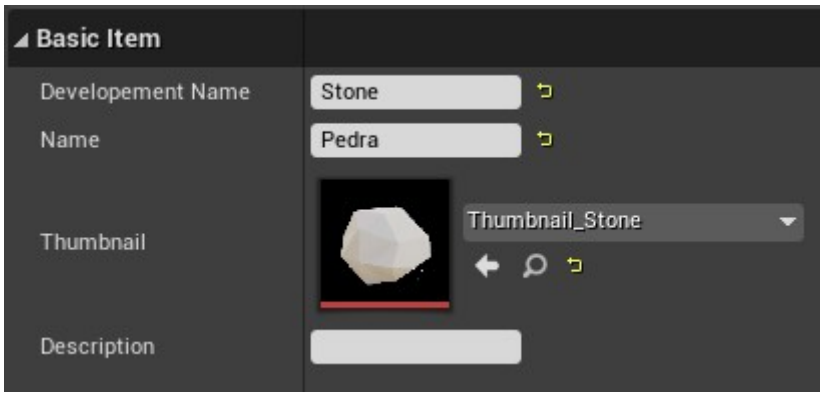


Figura 7.56: Atributs dels BasicItems des de l'editor

7.4.2. MaterialItem

La classe `MaterialItem` implementa els materials que el jugador pot trobar a l'escenari. Aquests tipus d'ítems no es poden utilitzar i no tenen cap funcionalitat extra, de forma que simplement s'ha modificar el valor de l'atribut `usable`. El següent fragment de codi mostra la definició de la classe `MaterialItem`.

```
UCLASS()
class PROJECTE_TFG_API AMaterialItem : public ABasicItem
{
	GENERATED_BODY()

	AMaterialItem();

};
```

Al següent fragment de codi s'observa com mitjançant el constructor de la classe `MaterialItem` es modifica l'atribut `usable` definit a la classe `BasicItem`.

```
AMaterialItem::AMaterialItem() {
	usable = false;
}
```

7.4.3. BundleItem

La classe *BundleItem* permet implementar ítems que al recollir-los donin un nombre de recursos definit. El nombre de recursos que donen es defineix mitjançant un enter que es pot modificar des de l'editor. El següent fragment de codi mostra la definició de la classe *BundleItem.*

```
UCLASS()
class PROJECTE_TFG_API ABundleItem : public ABasicItem
{
	GENERATED_BODY()

	UPROPERTY(EditAnywhere)
	int amount;

public:
	UFUNCTION(BlueprintCallable)
	int GetAmount() const;

	UFUNCTION(BlueprintCallable)
	void SetAmount(int n);

};
```

A la Figura 7.57 es poden observar com des de l'editor es poden modificar els atributs de *BasicItem* i de *BundleItem*.

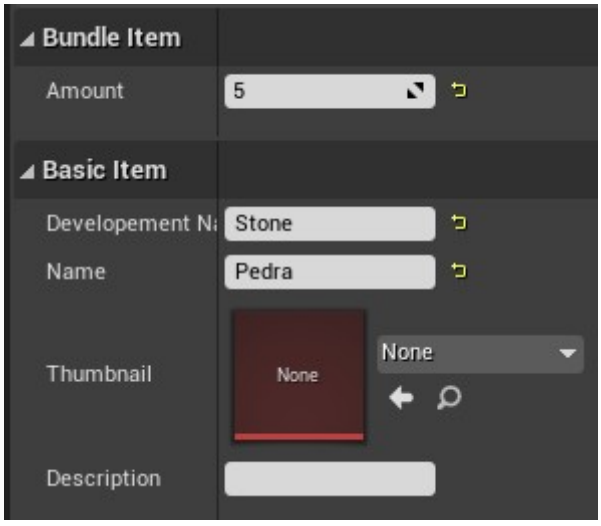


Figura 7.57: Atributs de BasicItem i BundleItem des de l'editor

7.4.4. ResourceItem

La classe *BasicResource* permet implementar ítems amb els quals es pot interactuar mitjançant eines i que donen materials al jugador. Aquest tipus d'ítems implementen un sistema de vida on es defineix un nombre de punts que s'han de treure per a poder obtenir els materials. El següent fragment de codi mostra la definició de la classe *BasicResource*.

```
UCLASS()
class PROJECTE_TFG_API ABasicResource : public  ABasicItem
{
	GENERATED_BODY()

public:
	// Sets default values for this actor's properties
	ABasicResource();

	UFUNCTION(BlueprintCallable)
	void harvest(int harvestAmmount);

	UFUNCTION(BlueprintImplementableEvent)
	void dropResource();

protected:

	UPROPERTY(EditAnywhere)
	int harvestTime;

	int harvestTotal;

};
```

El mètode *dropResource()*, implementat des de les Blueprints com a esdeveniment, fa aparèixer al món els ítems del tipus material que dona el recurs al jugador. La Figura 7.58 mostra la implementació d'aquest mètode des de la Blueprint *MasterRock.*

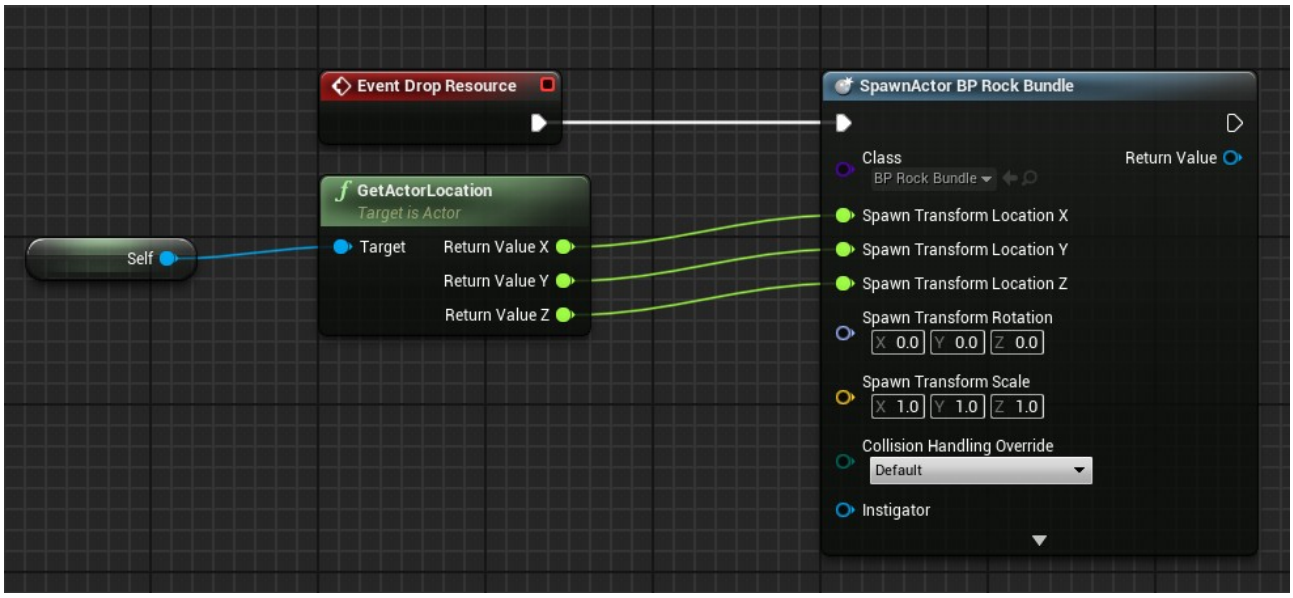


Figura 7.58: Implementació del mètode dropResource()

Els mètode *`harvest(int harvestAmount)`* s'encarrega de treure a l'ítem la quantitat de punts passada com a paràmetre, i també s'encarrega de cridar al mètode *`dropResource()`* i destruir el recurs en cas que la vida arribi a 0. El següent fragment de codi mostra la implementació d'aquest mètode:

```
void ABasicResource::harvest(int harvestAmmount) {

	int remainingHarvestTime = harvestTotal - harvestAmmount;
	if (remainingHarvestTime > 0) {
		harvestTotal = remainingHarvestTime;

	}
	else {
		this->dropResource();
		this->Destroy();
	}
}
```

Quan el jugador utilitza alguna d'aquestes eines s'ha de detectar si està apuntant a algun objecte amb el que puguin interactuar. Mitjançant la funció *`TraceResource()`* es detecta si el jugador apunta a algun objecte amb el que interactuar, i si el troba guarda una referència a la variable *`ActorHit`*. Quan s'executa el codi d'utilitzar les eines es dispara l'esdeveniment del jugador *`InteractWithResource(int harvestAmout)`*. Aquest esdeveniment comprova que l'eina utilitzada i el tipus de recurs siguin compatibles (no es pot tallar un arbre amb un pic, per exemple), i mitjançant la funció *`harvest()`*, treu al recurs la quantitat de punts de recol·lecció passada a l'esdeveniment. Finalment, segons el tipus de recurs, reprodueix un so a la posició del jugador, per a donar feedback

sobre si s'ha utilitzat o no l'eina. La Figura 7.59 mostra la implementació de l'esdeveniment *InteractWithResource()*.

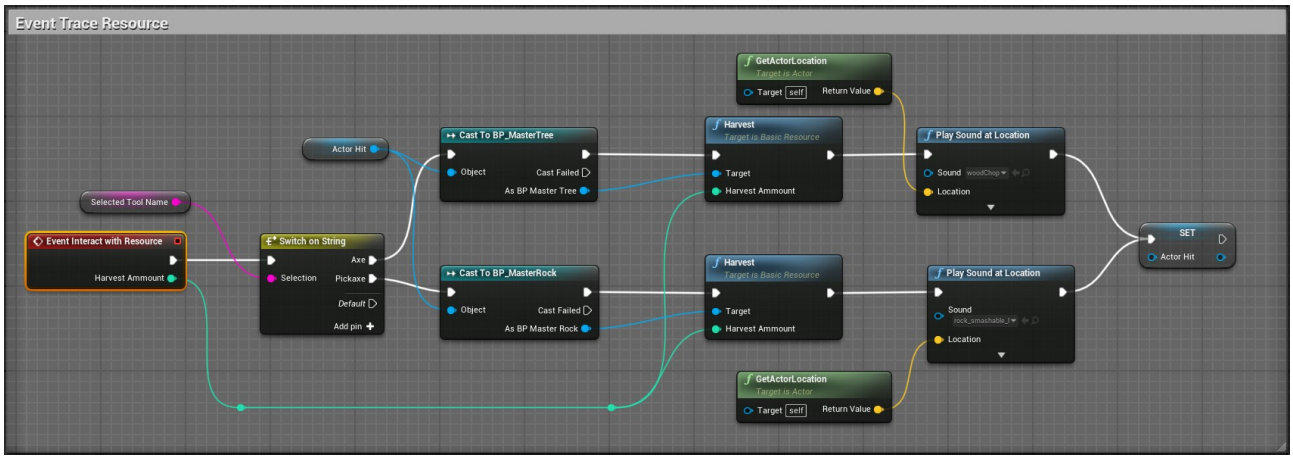


Figura 7.59: Implementació de l'esdeveniment interactWithResource

7.4.5. FoodItem

La classe *FoodItem* implementa els aliments que el jugador pot menjar. Els ítems d'aquest tipus es poden utilitzar, i al fer-ho incrementen la vida del jugador.

Aquest tipus d'ítems tenen un atribut extra que indica la quantitat de vida que recuperen al jugador, i per a poder modificar-ho de forma dinàmica des de l'editor, s'ha marcat amb la macro *EditAnywhere*. Com que es poden utilitzar s'ha de sobreescriure el mètode *use()*. El següent fragment de codi mostra la definició de la classe *FoodItem*.

```
UCLASS()
class PROJECTE_TFG_API AFoodItem : public ABasicItem
{
	GENERATED_BODY()

protected:
	UPROPERTY(EditAnywhere)
	int healthGain;

public:
	virtual void use(AActor* user) const override;
};
```

La implementació del mètode *use()* ha d'incrementar la vida del jugador tants punts com indiqui l'atribut protected *healthGain*. Com que la gestió de la vida es fa des del *HealthComponent*, que es veurà més endavant, s'ha de comprovar si l'actor que vol utilitzar l'ítem té un component d'aquest tipus. Si la referència a l'actor és vàlida i aquest té un component del tipus *HealthComponent* s'incrementa la vida cridant al mètode del

component *gainHealth()*. El següent fragment de codi mostra la implementació del mètode *use()*.

```
void AFoodItem::use(AActor* user) const {
  if (user){
      UHealthComponent* userHealthComponent = user->FindComponentByClass<UHealthComponent>();
      if (userHealthComponent) {
         userHealthComponent->gainHealth(healthGain);
      }
   }
}
```

Des de l'editor es poden modificar tant els paràmetres definits a *BasicItem* com els definits a *FoodItem*. La Figura 7.60 mostra ítem del tipus *FoodItem* des de l'editor:

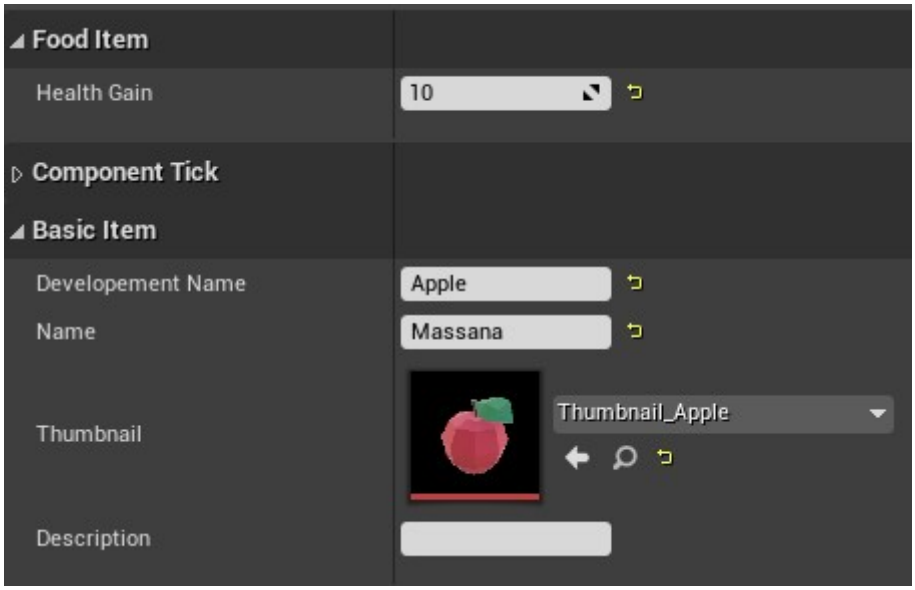


Figura 7.60: Atributs de BasicItem i FoodItem des de l'editor

7.4.6. CraftingItem

La classe *CraftingItem* implementa els ítems que es poden fabricar. Com es pot veure a la Figura 7.61 hi ha tres tipus de CraftingItems, els *MasterCraftingItem*, implementats en Blueprints, els *ToolItem*, dels quals es parlarà en el punt 7.3.5.1, i els *BuildingItem*, dels quals es parlarà en el punt 7.3.5.2.

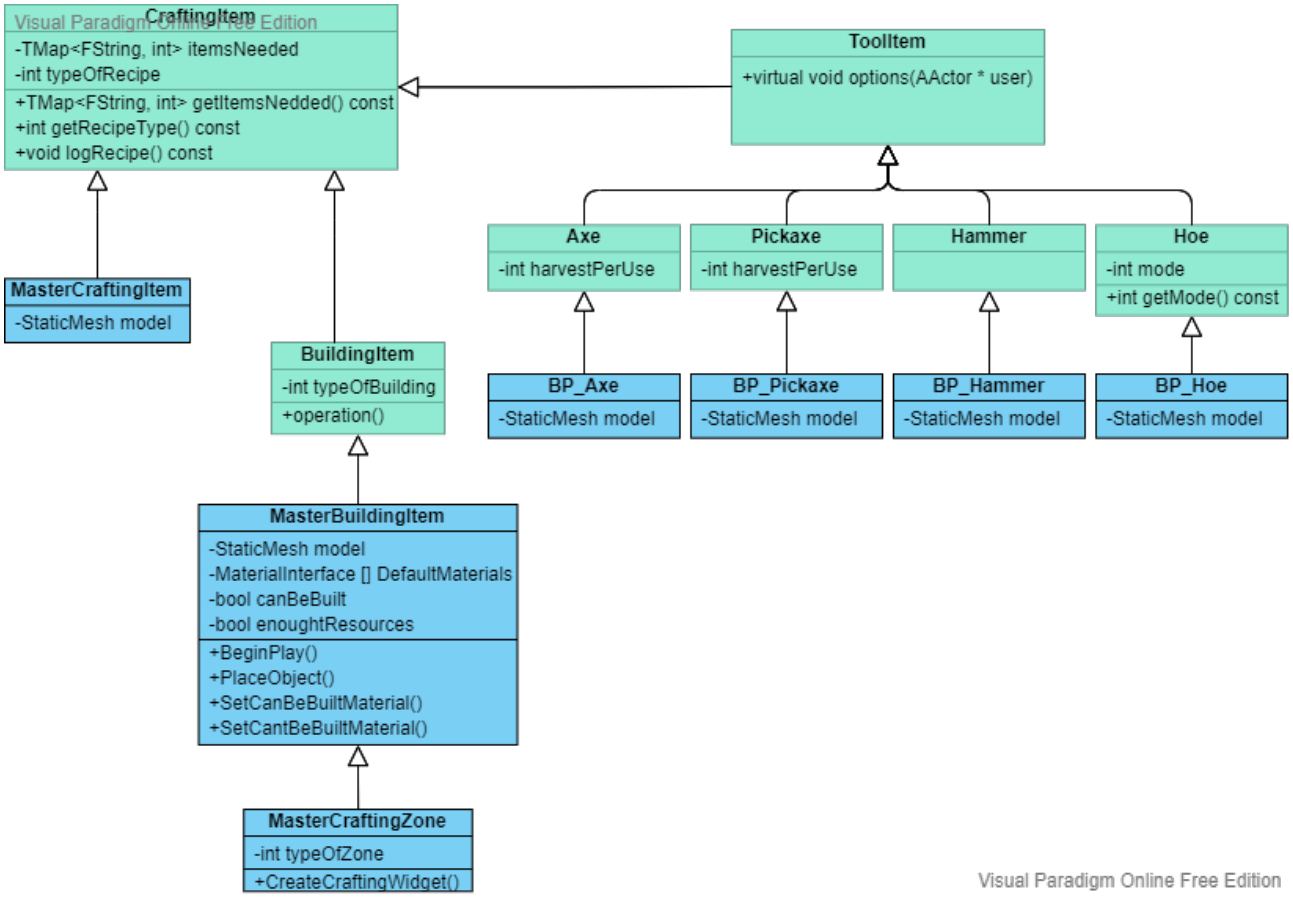


Figura 7.61: Diagrama de classes amb els tipus de CraftingItem

Els objectes d'aquest tipus es creen a partir de receptes, que són relacions d'ítems i quantitats. Aquestes receptes estan formades per un *Map* on la clau és el nom de l'ítem i el valor és un enter que indica la quantitat d'objectes necessaris per a crear el nou element de la recepta. Mitjançant un enter s'indica a quina categoria de receptes pertany l'ítem. El següent fragment de codi mostra la definició de la classe `CraftingItem`.

```
UCLASS()
class PROJECTE_TFG_API ACraftingItem : public ABasicItem
{
	GENERATED_BODY()


protected:
	UPROPERTY(EditAnywhere)
	TMap<FString, int> itemsNeeded;

	UPROPERTY(EditAnywhere)
	int typeOfRecipe;

public:
	UFUNCTION()
	void logRecipe() const;


	UFUNCTION(BlueprintCallable)
	TMap<FString, int> GetItemsNeeded() const;

	UFUNCTION(BlueprintCallable)
	int GetRecipeType() const;

};
```

La Figura 7.62 mostra les propietats exposades a les Blueprints d'un ítem d'aquest tipus.

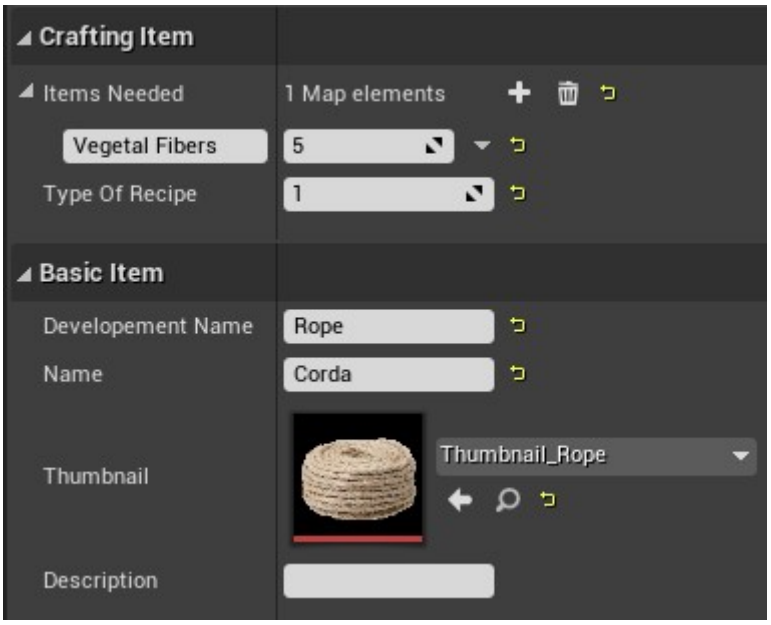


Figura 7.62: Atributs de BasicItem i CraftingItem des de l'editor

7.4.6.1. Toolitem

Les eines són un tipus d'ítem que es pot craftejar i utilitzar, i cada eina ha d'estar programada per a fer unes tasques diferents. El següent fragment de codi mostra la definició de la classe *`ToolItem`*.

```
UCLASS()
class PROJECTE_TFG_API AToolItem : public ACraftingItem
{
	GENERATED_BODY()

public:
	AToolItem();

	virtual void use(AActor* user) const override;
	virtual void Options(AActor* user) ;
};
```

7.4.6.1.1. Destral i pic

La destral i el pic són eines que permeten interactuar amb elements de l'escenari per a obtenir ítems. A continuació es mostrarà la implementació de la classe *`Axe`*, la classe *`Pickaxe`* no es mostrarà ja que és molt semblant. Un atribut enter que es pot modificar des de l'editor indica la quantitat de punts de recol·lecció que treu la eina quan s'utilitza contra un arbre o arbust. El següent fragment de codi mostra la definició de la classe *`Axe`*.

```
UCLASS()
class PROJECTE_TFG_API AAxeTool : public AToolItem
{
public:
	GENERATED_BODY()

protected:

	UPROPERTY(EditAnywhere)
	int harvestPerUse;

public:
	virtual void use(AActor* user) const override;
};
```

El mètode *`use()`* accedeix a la referència del *`Character`* del jugador a través de la classe d'utilitats *`UgameplayStatics`* i dispara l'esdeveniment *`interactWithResource()`* vist al punt 7.4.4. El següent fragment de codi mostra la implementació del mètode *`use()`*.

```
void AAxeTool::use(AActor* user) const {

	AProjecte_TFGCharacter* myCharacter =
	Cast<AProjecte_TFGCharacter>(UGameplayStatics::GetPlayerCharacter(GetWorld(), 0));

	myCharacter->interactWithResource(harvestPerUse);

}
```

7.4.6.1.2. Martell

El martell és l'eina que permet construir a l'escenari. La classe *Hammer* només implementa el mètode *use()*, que accedeix al jugador com fan la destral i el pic, però llença l'esdeveniment *build()* vist al punt 7.2.2.5. El següent fragment de codi mostra la implementació d'aquest mètode.

```
void AHammer::use(AActor* user) const {

	AProjecte_TFGCharacter* myCharacter =
	Cast<AProjecte_TFGCharacter>(UGameplayStatics::GetPlayerCharacter(GetWorld(), 0));

	myCharacter->build();

}
```

7.4.6.1.3. Aixada

L'aixada és una eina que permet crear zones cultivables i utilitzar-les per a plantar. Com que ha de fer dues tasques diferents, s'ha creat un atribut *mode* que indica si l'eina està treballant en mode plantar o en mode llaurar. El següent fragment de codi mostra la definició de la classe *Hoe*.

```
UCLASS()
class PROJECTE_TFG_API AHoe : public AToolItem
{
public:
	GENERATED_BODY()

protected:

	UPROPERTY()
	int mode;

public:
	AHoe();

	virtual void use(AActor* user) const override;

	virtual void Options(AActor* user) override;

	UFUNCTION(BlueprintCallable)
	int GetMode() const;
};
```

La classe *Hoe* sobreescriu dos mètodes, *use()*, que està definit a *BasicItem* i dispara l'esdeveniment corresponent segons el mode en el que està l'eina, i *options()*, que està definit a *ToolItem* i s'encarrega de canviar el mode de l'eina.

Si l'eina està en mode plantar el mètode *use()* accedeix a la referència del jugador, tal i com s'ha vist a les altres eines, i activa l'esdeveniment *Plant()* explicat al punt 7.2.2.5. Si està en mode llaurar accedeix a la referència del *GameMode* mitjançant la funció del motor *GetAuthGameMode()* i dispara l'esdeveniment *DrawAtLocation()*.

Per a integrar les zones cultivables amb la geometria del terreny s'ha decidit treballar amb textures que es puguin modificar dinàmicament en temps d'execució. Aquest tipus de textures s'implementen dintre d'Unreal Engine 4 com a *RenderTargets*. La idea darrere d'aquesta solució és que, quan el jugador utilitzi l'aixada en mode llaurar, es dibuixi dintre d'un *RenderTarget* la zona que s'ha convertit en cultivable, per poder utilitzar aquesta informació per aplicar textures de forma dinàmica des del *shader* del terreny. La Figura 7.63 mostra un exemple del funcionament d'aquesta idea, i es pot veure com la zona blanca del *RenderTarget* coincideix amb la zona seleccionada com a cultivable al terreny. Per entendre el funcionament de l'aixada en aquest punt s'explicarà com dibuixar a sobre del *RenderTarget*, i més endavant, al Punt 7.8, s'explicarà com utilitzar aquesta textura des del material del terreny per modificar-lo de forma dinàmica.

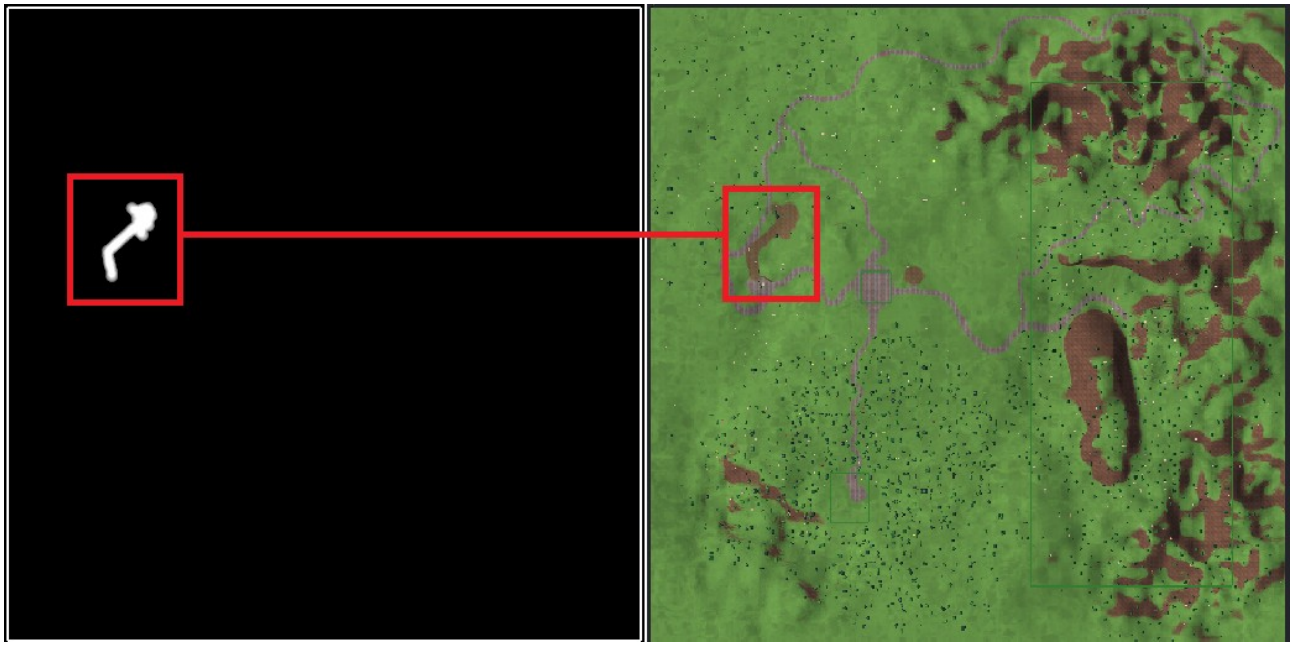


Figura 7.63: RenderTarget i terreny amb les textures aplicades

Per poder pintar a sobre d'un *RenderTarget* s'ha de crear un material que actuï com a pinzell i que pugui dibuixar a la posició on es troba el jugador. Per aquest joc s'ha implementat un pinzell circular al voltant del jugador i s'han parametritzat els valors del radi i de la duresa de dibuixat. El material creat ha d'estar configurat amb *blending mode* ADDITIVE de manera que al dibuixar sobre el *RenderTarget* es mantinguin les zones seleccionades. L'editor de materials de Unreal Engine 4 només permet treballar amb colors, que són vectors de fins a quatre dimensions que indiquen el valor dels components roig (R), verd (G), blau (B) i la transparència (alfa). Per tant si volem passar la posició del jugador per a projectar la zona cultivable al seu voltant, hem de convertir el vector que emmagatzema la posició en coordenades cartesianes (X, Y i Z) a un color, on el canal roig representarà la X, el canal verd la Y i el canal blau la Z. Com que les textures són imatges bidimensionals es pot descartar el valor de la component Z i el valor de la transparència. La Figura 7.64 mostra el paràmetre `ForcePosition`, que rep un color amb la posició del jugador i mitjançant una màscara elimina els canals Blau i Transparència, convertint el vector inicial en un vector de dues dimensions amb les coordenades X i Y de la posició del jugador a l'escenari.

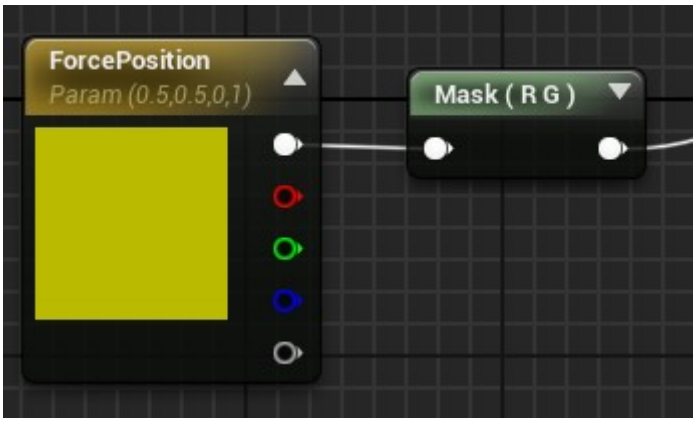


Figura 7.64: Posició del jugador parametritzada

A partir d'aquest vector de dues dimensions obtingut es pot calcular la distància Euclidiana a les coordenades de la textura UV per obtenir un cercle negre a la posició del jugador respecte la textura del material. La Figura 7.65 mostra la implementació del càlcul de la distància i el resultat obtingut projectat a un pla.

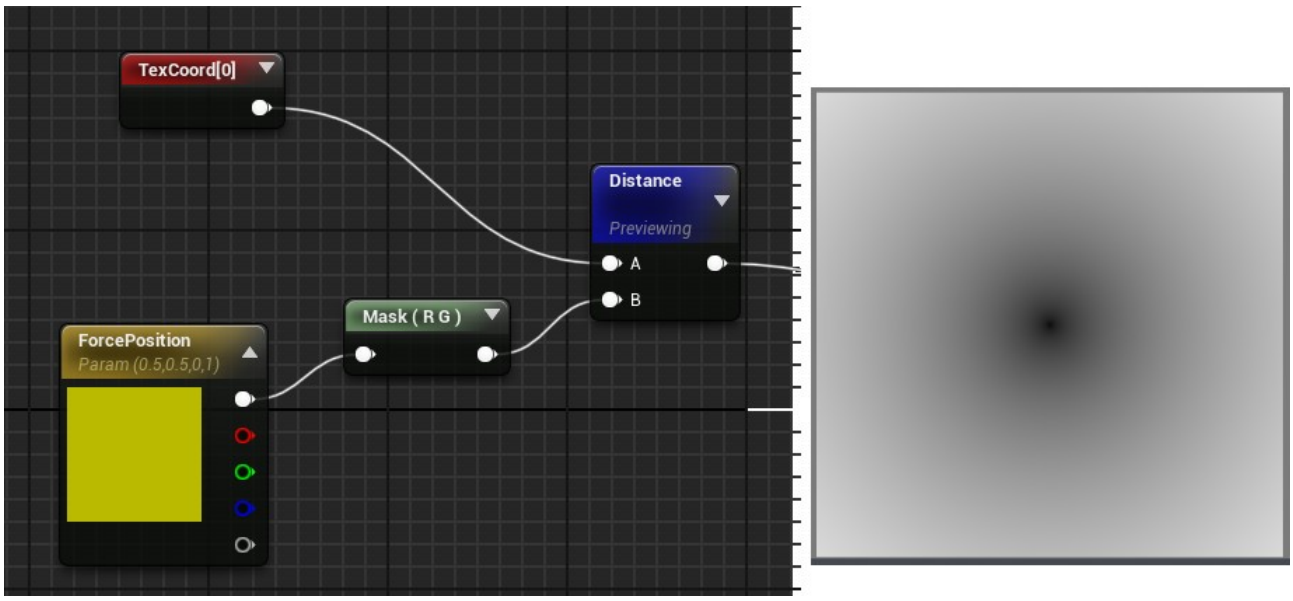


Figura 7.65: Generació d'un punt a la posició del jugador respecte la textura

Per a tenir més control sobre la mida del cercle que es projectarà, s'ha afegit un paràmetre anomenat *`ForceSize`,* que controla el radi del cercle generat. Per aplicar aquest paràmetre s'ha d'invertir la textura generada, ja que per poder utilitzar el pinzell com una màscara el centre ha de ser blanc i les vores negres, i restar-li la inversa del paràmetre, per obtenir un pinzell circular de la mida *`ForceSize`*. Finalment s'ha de saturar el resultat per a deixar només valors de 0 o 1 dintre de la textura. La Figura 7.66 mostra com s'aplica el paràmetre de la força i el resultat.

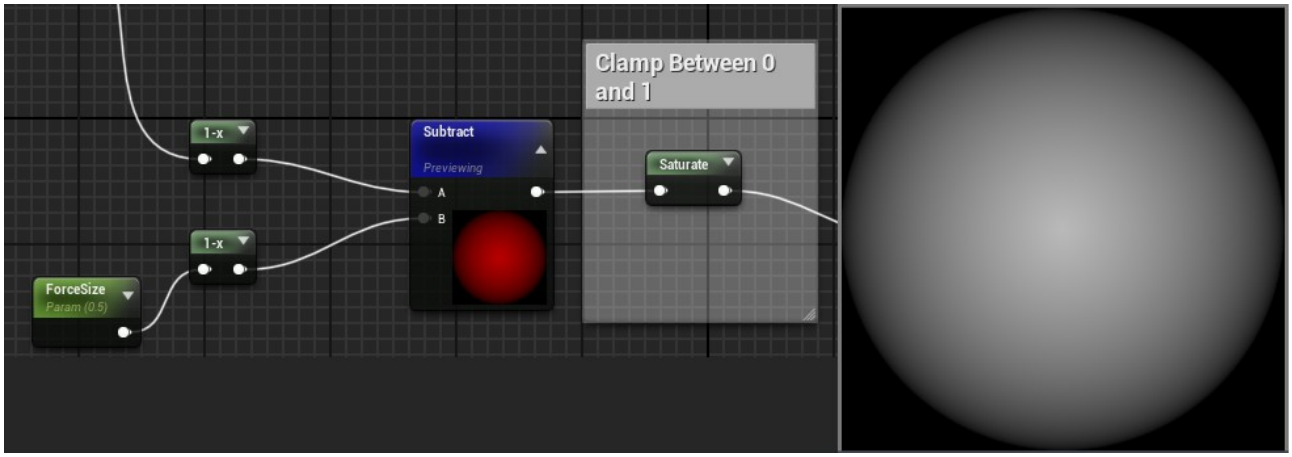


Figura 7.66: Incrementar la mida del pinzell a partir d'un paràmetre

Per acabar el material del pinzell, s'ha definit un paràmetre `ForceStrength` que defineix la duresa amb la que el pinzell interactuara amb el *RenderTarget*. Aquest paràmetre s'ha de multiplicar pel resultat obtingut al pas anterior, i es retorna al material com a *Emissive Color*. La Figura 7.67 mostra la implementació de tot el shader.

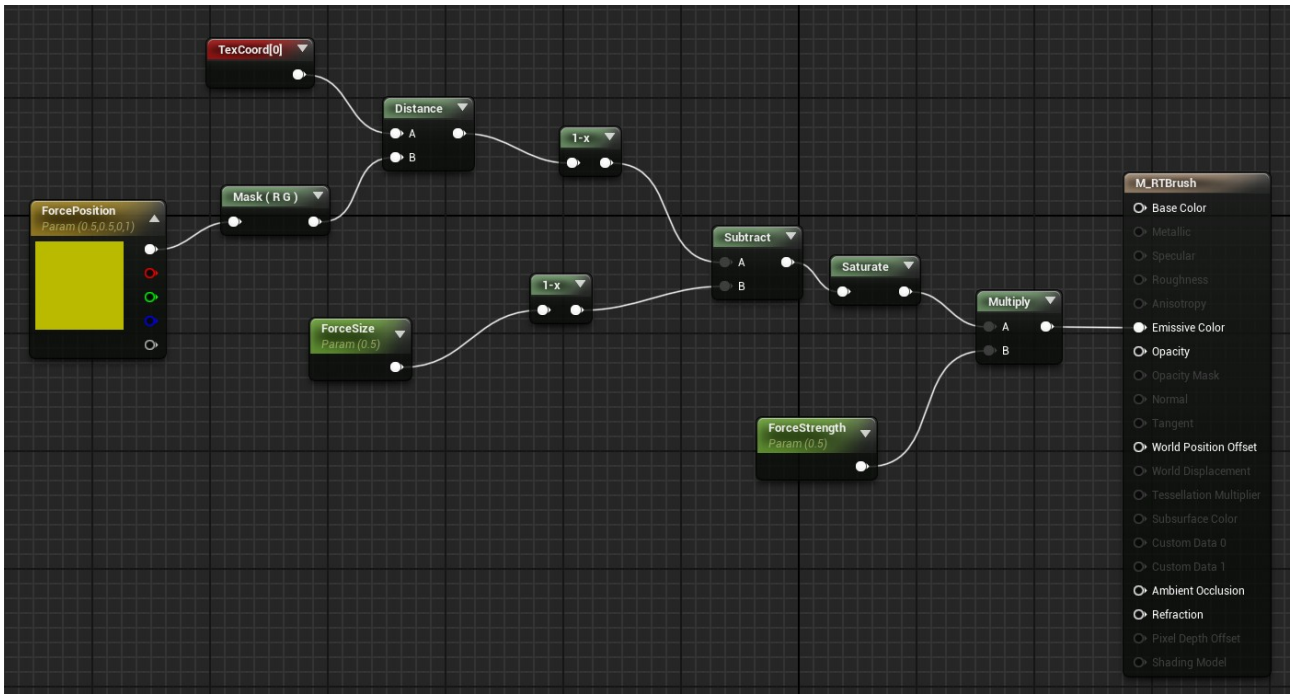


Figura 7.67: Implementació del pinzell

Treballar amb un material parametritzat permet modificar-lo en temps d'execució, com s'ha explicat al Punt 4. L'esdeveniment `DrawAtLocation(),` que crida l'aixada quan el jugador la utilitza, simplement dona valor a aquests paràmetres i mitjançant la funció del motor `DrawMaterialToRenderTarget()` projecta el pinzell configurat a sobre del RenderTarget. La Figura 7.68 mostra la implementació d'aquest esdeveniment, on es pot

veure com es passa al material un color generat a partir de la posició del jugador normalitzada entre la mida del mapa i com s'indica la mida de l'efecte a partir d'una variable per poder fer proves ràpides amb diferents valors.

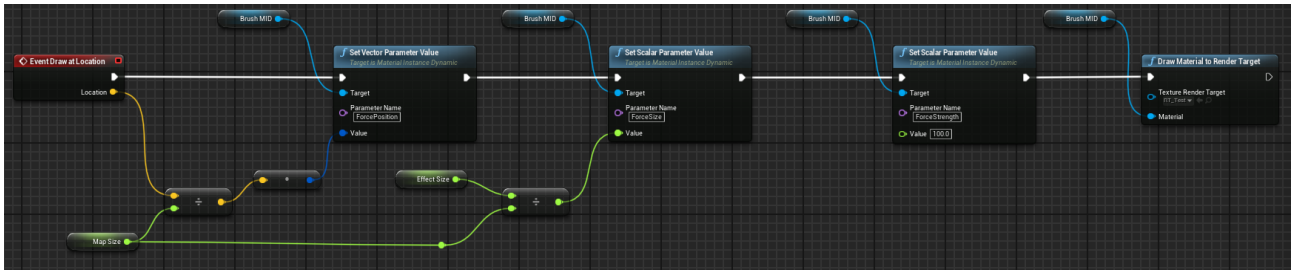


Figura 7.68: Implementació de l'esdeveniment DrawAtLocation()

7.4.6.2. BuildingItem

La classe *`BuildingItem`* implementa els ítems que es poden construir. Aquest tipus d'ítems no es poden utilitzar, i mitjançant un atribut enter es defineix el tipus d'element a construir. El següent fragment de codi mostra la definició de la classe *`BuildingItem`*.

```
UCLASS()
class PROJECTE_TFG_API ABuildingItem : public  ACraftingItem
{
	GENERATED_BODY()

	UPROPERTY(EditAnywhere)
	int typeOfBuilding;

public:
	ABuildingItem();

	UFUNCTION(BlueprintCallable)
	int GetTypeOfBuilding() const;
};
```

A la Figura 7.69 es poden observar els atributs modificables a les instàncies dels objectes d'aquest tipus.

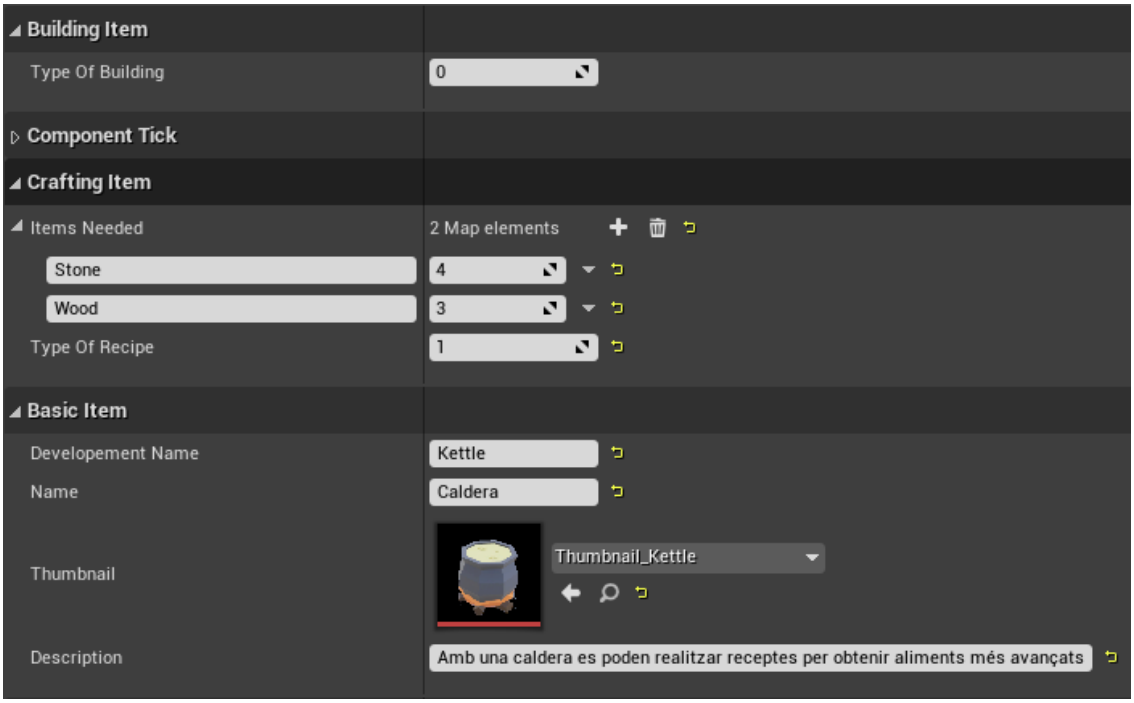


Figura 7.69: Atributs de BasicItem, CraftingItem i BuildingItem des de l'editor

Quan el jugador vol col·locar un element, aquest es veu roig o verd segons si es pot col·locar o no, com es pot observar a la Figura 7.70.

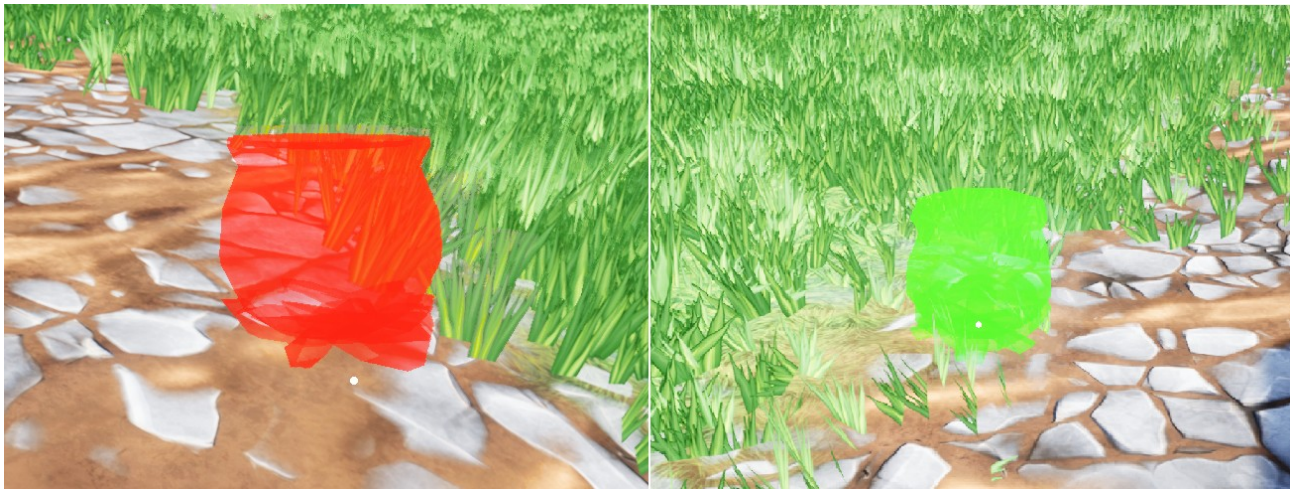


Figura 7.70: Canvi de materials segons si l'element es pot construir o no

Al tractar-se de modificar l'aspecte visual dels objectes la gestió del canvi de materials s'ha implementat a la Blueprint `MasterBuildingItem`. Les funcionalitats necessàries s'han implementat com a esdeveniments personalitzats.

BeginPlay

Quan es crea un objecte s'ha de guardar el material inicial, per poder-lo aplicar quan el jugador col·loqui la construcció a l'escenari. Un cop guardat s'ha de fer un loop per tots els

materials i canviar-los pel material de previsualització, i finalment s'han de desactivar les col·lisions. La Figura 7.71 mostra la implementació d'aquest esdeveniment.

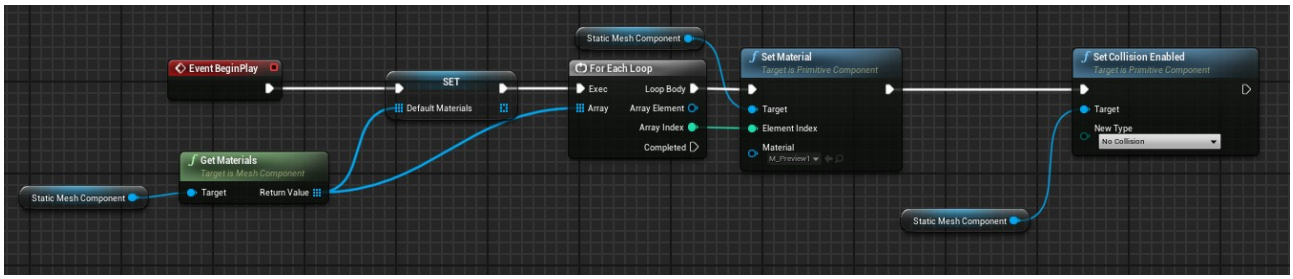


Figura 7.71: Implementació de l'esdeveniment BeginPlay

SetCanBuildMaterial i SetCantBuildMaterial

Aquests esdeveniments canvien tots els materials pel material verd o pel material roig. La Figura 7.72 mostra la implementació d'aquests dos esdeveniments.

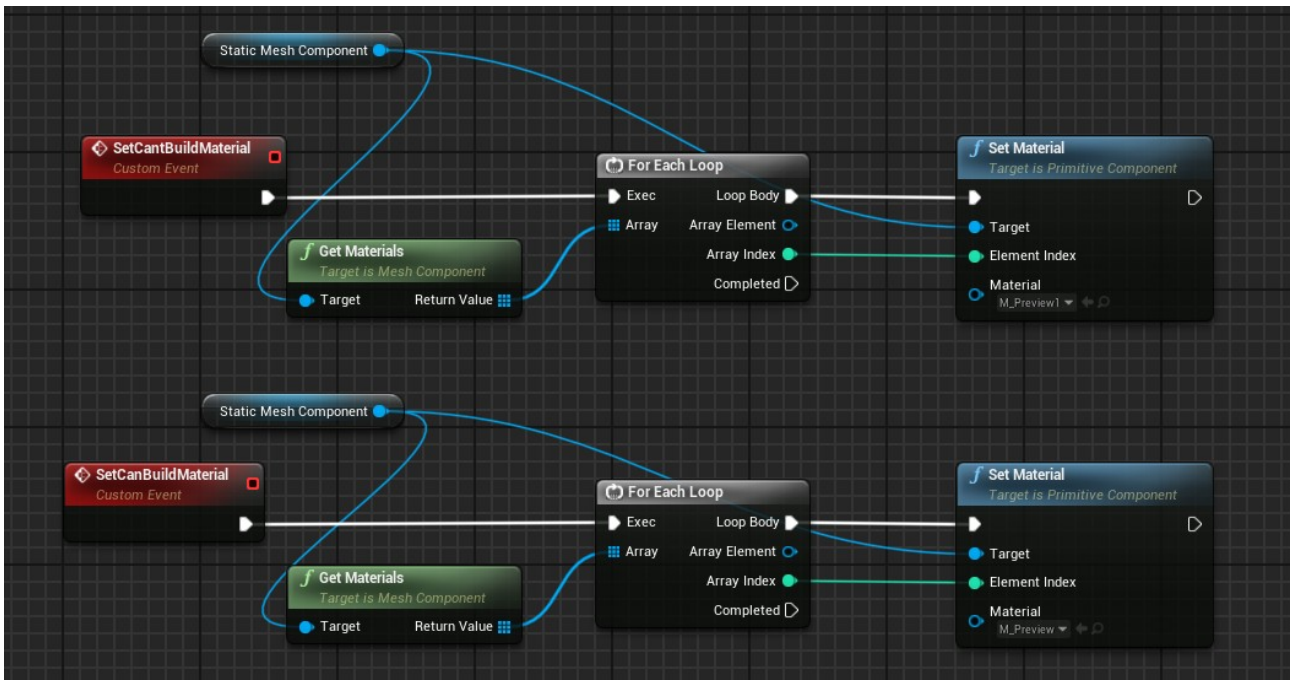


Figura 7.72: Implementació dels esdeveniments SetCanBuildMaterial i SetCantBuildMaterial

PlaceObject

Quan el jugador col·loca l'edifici a l'escenari s'ha de recórrer el vector on s'han guardat els materials inicials de l'objecte i tornar-los a assignar. La Figura 7.73 mostra la implementació de l'esdeveniment `PlaceObject`.

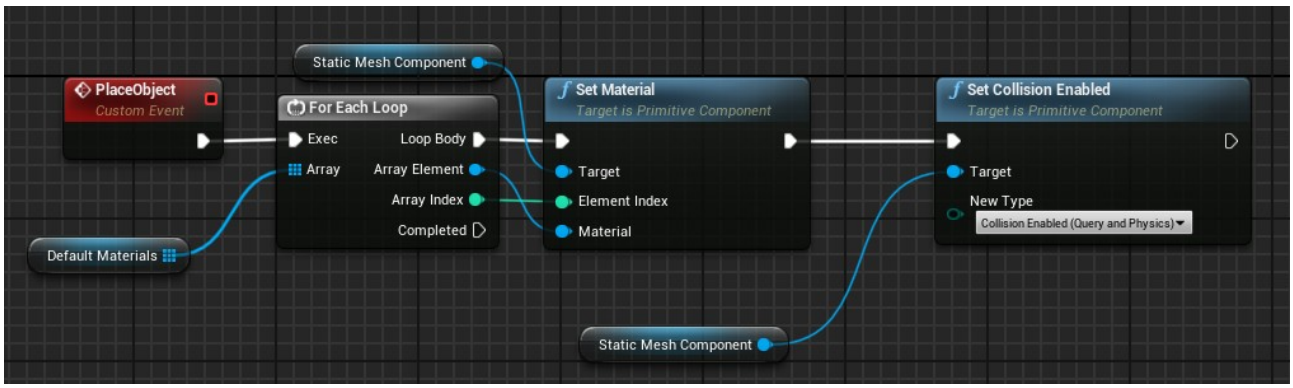


Figura 7.73: Implementació de l'esdeveniment PlaceObject

7.5. Inventari i crafteig

El sistema d'inventari permet emmagatzemar ítems. S'ha implementat mitjançant components, permetent que cada Actor del joc pugui emmagatzemar ítems utilitzant aquesta mateixa classe base. El propi inventari no integra el sistema de crafteig, però defineix una estructura perquè es pugi implementar des de qualsevol element del joc.

El jugador incorpora dos inventaris, un per a emmagatzemar ítems i l'altre per a emmagatzemar eines. Com que hi ha petites diferències a l'hora de gestionar els ítems i les eines s'ha decidit implementar cada tipus d'inventari com a un component diferent. La classe `InventoryComponent` implementa en C++ l'inventari d'ítems. Aquest component permet afegir, treure i utilitzar ítems, i a més incorpora funcionalitats relacionades amb el crafteig. La classe `ToolsInventoryComponent` implementa, també en C++, l'inventari d'eines. Aquest component gestiona funcionalitats específiques de les eines com ara gestionar el mode de treball en el què es troben.

7.5.1. Inventari d'ítems

Al dissenyar l'estructura de dades de l'inventari s'han hagut de tenir en compte un parell de consideracions per adaptar-lo al funcionament plantejat al disseny de mecàniques:

- S'havien de poder guardar diferents ítems d'un mateix tipus
- L'inventari havia de tenir accés a les instàncies dels ítems per poder utilitzar els mètodes implementats

L'estructura de dades s'ha implementat mitjançant un *Map*, ja que permet fer cerques molt eficientment, i durant la partida fa falta accedir moltes vegades a la informació de l'inventari. Per a complir les consideracions de disseny s'ha decidit crear una estructura

que guarda una referència a un objecte del tipus *BasicItem* i un enter que indica la quantitat, i aquesta estructura guardar-la en un Map utilitzant com a clau una String amb el nom de l'ítem. El següent fragment de codi mostra la definició de l'estructura *FitemInfo.*

```
USTRUCT()
struct FitemInfo {
	GENERATED_BODY()

	UPROPERTY()
	ABasicItem* itemRef;
	UPROPERTY()
	int32 itemCount;
	FitemInfo() {
		itemRef = nullptr;
		itemCount = 0;
	}
};
```

El següent fragment de codi mostra la definició de la classe *InventoryComponent.*

```
class PROJECTE_TFG_API UInventoryComponent : public UActorComponent
{
public:
	UInventoryComponent();

	UFUNCTION(BlueprintCallable, Category = "Inventory")
	void addToInventory(ABasicItem* item);

	UFUNCTION(BlueprintCallable, Category = "Inventory")
	void deleteFromInventory(FString item, int amount = 1);

	UFUNCTION(BlueprintCallable, Category = "Inventory")
	int useItem(FString item);

	UFUNCTION(BlueprintCallable, Category = "Inventory")
	int32 GetNElements() const;

	UFUNCTION(BlueprintCallable, Category = "Inventory")
	TMap<FString, int32> GetInventoryBlueprintType() const;

	UFUNCTION(BlueprintCallable, Category = "Inventory")
	UTexture2D* GetItemThumbnail(FString item);

	UFUNCTION(BlueprintCallable, Category = "Inventory")
	bool AreEnoughtItemsByName(FString itemName, int neededItems);

	UFUNCTION(BlueprintCallable, Category = "Inventory")
	int GetItemCountByName(FString itemName) ;

	UFUNCTION(BlueprintCallable, Category = "Inventory")
	bool IsItemUsable(FString itemName);

	UFUNCTION(BlueprintCallable, Category = "Inventory")
	void CraftItem(TMap<FString, int> neededItems);

	UFUNCTION(BlueprintCallable, Category = "Inventory")
	bool logInventoryToFile(FString filePath) const;
protected:
	UPROPERTY()
	TMap<FString, FitemInfo> currentInventory;
	AActor* Owner;
};
```

A continuació s'explicarà el funcionament dels mètodes més importants d'aquesta classe.

void addToInventory(ABasicItem* item): Aquest mètode afegeix l'ítem passat com a paràmetre a l'inventari. Per afegir un ítem s'ha de comprovar si és de tipus *`ItemBundle`* o no, ja que segons el tipus variarà la quantitat d'ítems a afegir. Si l'ítem existeix a l'inventari s'incrementa el comptador, i si no existeix es crea una nova estructura *`FitemInfo`* i s'afegeix l'ítem a l'inventari.

```
void UInventoryComponent::addToInventory(ABasicItem* item) {
	ABasicItem* itemRef = nullptr;
	AProjecte_TFGGameMode* GameMode = (AProjecte_TFGGameMode*)GetWorld()->GetAuthGameMode();
	// Check if item is Bundle and get item reference
	ABundleItem* ref = Cast<ABundleItem>(item);
	if (ref != nullptr) {
		if (item != nullptr) {
			itemRef = GameMode->GetItemReference(item->GetDevelopementName());
		}
	}
	else {
		itemRef = item;
	}
	// Find if the item is already at the inventory
	FitemInfo* aux = nullptr;
	if (itemRef != nullptr) {
		aux = currentInventory.Find(itemRef->GetDevelopementName());
	}

	// If the item has been found increase the counter
	if (aux != nullptr) {
		if (ref != nullptr) {
			aux->itemCount += ref->GetAmount();
		}
		else {
			aux->itemCount++;
		}
	}
	// If the item has NOT been found add it to the inventory
	else {
		if (ref != nullptr && itemRef != nullptr) {
			FitemInfo newItem(itemRef, ref->GetAmount(), itemRef->GetClass());
			currentInventory.Add(itemRef->GetDevelopementName(), newItem);
		}
		else if (itemRef != nullptr) {
			FitemInfo newItem(itemRef, 1, itemRef->GetClass());
			currentInventory.Add(itemRef->GetDevelopementName(), newItem);
		}
	}
}
```

void deleteFromInventory(FString item, int amount): Aquest mètode comprova al Map si existeix alguna clau igual que la passada per paràmetre, i en cas de que existeixi treu la quantitat d'ítems indicada. En cas que el comptador d'ítems es quedi a 0 elimina l'entrada de l'ítem del Map.

```
void UInventoryComponent::deleteFromInventory(FString item, int amount)  {
	FitemInfo* aux = currentInventory.Find(item);
	if (aux != nullptr) {
		if (aux->itemCount > amount) {
			(aux->itemCount)-= amount;
		}
		else {
			currentInventory.Remove(item);
		}
	}
}
```

int useItem(FString item): Aquest mètode busca si existeix l'ítem passat per paràmetre, i en cas que sigui utilitzable accedeix a la referència de l'ítem i executa el mètode *use()*. Un cop utilitzat l'ítem el treu de l'inventari i retorna la quantitat d'ítems del tipus restant. Si l'ítem no es pot utilitzar retorna -2 i si hi ha algun altre error retorna -1.

```
int UInventoryComponent::useItem(FString item) {
	FitemInfo* aux = currentInventory.Find(item);
	if (aux != nullptr) {
		if (Owner) {
			if (aux->itemCount > 0) {
				if (aux->itemRef->isUsable()) {
					aux->itemRef->use(Owner);
					deleteFromInventory(item);
					return aux->itemCount;
				}
				else {
					return -2;
				}
			}
			else {
				return -1;
			}
		}
	}
	return -1;
}
```

TMap<Fstring, int32> getInventoryBlueprintType() const: Les Blueprints no poden treballar amb estructures personalitzades creades en C++, ja que no reconeixen el tipus. Aquest mètode recorre l'inventari i el guarda en un nou mapa on la clau són Strings i el valor enters, i que al ser tipus bàsics són reconeguts per les Blueprints. La principal aplicació d'aquest mètode és poder obtenir la informació de l'inventari des de les Widget Blueprints que implementen les interfícies d'usuari.

```
TMap<FString, int32> UInventoryComponent::GetInventoryBlueprintType() const {
	TMap<FString, int32> inventoryBP;
	for (auto It = currentInventory.CreateConstIterator(); It; ++It)
	{
		inventoryBP.Add(*It.Key(), It.Value().itemCount);
	}

	return inventoryBP;
}
```

void CraftItem(TMap <Fstring, int> neededItems): Aquest mètode rep un Map que relaciona noms d'ítems amb quantitats necessàries, i els treu de l'inventari. No hi ha cap control de quantitats ja que aquest mètode està pensat per ser utilitzat des de les interfícies, on ja s'ha comprovat si hi ha suficients ítems o no.

```
void UInventoryComponent::CraftItem(TMap<FString, int> neededItems) {
	FitemInfo* aux = nullptr;
	for (auto It = neededItems.CreateConstIterator(); It; ++It)
	{
		aux = currentInventory.Find(It.Key());
		if (aux != nullptr) {
			deleteFromInventory(It.Key(), It.Value());
		}
	}
}
```

7.5.1.1. Integració amb el jugador

Quan el jugador agafa ítems aquests es guarden a l'inventari, i totes les tasques de gestió de recursos, com utilitzar o fabricar ítems, es gestionen des de les interfícies d'usuari. Cada cop que el jugador obre l'inventari es construeix la interfície a partir de dos *Widget Blueprints*: el *W_Inventory*, que gestiona la visualització dels ítems, i el *W_InventoryEntry*, que mostra la informació del nom, miniatura i quantitat de cada ítem, i incorpora un botó per a utilitzar l'ítem.

La *Widget Buleprint W_Inventory* s'ha creat a partir d'una *ScrollBox*, que és un tipus de Widget que sempre ocupa el mateix espai, i si ha de mostrar més informació de la que cap, de forma automàtica permet fer scroll per a mostrar-ho tot, i un *UniformGridPanel*, que és un tipus de Widget que divideix l'espai a parts iguals entre tots els seus fills. A aquesta graella s'afegeixen *Widgets* del tipus *W_InventoryEntry*. La Figura 7.74 mostra la part gràfica de les dues *Widget Blueprints* i la jerarquia de *Widgets* que les formen.

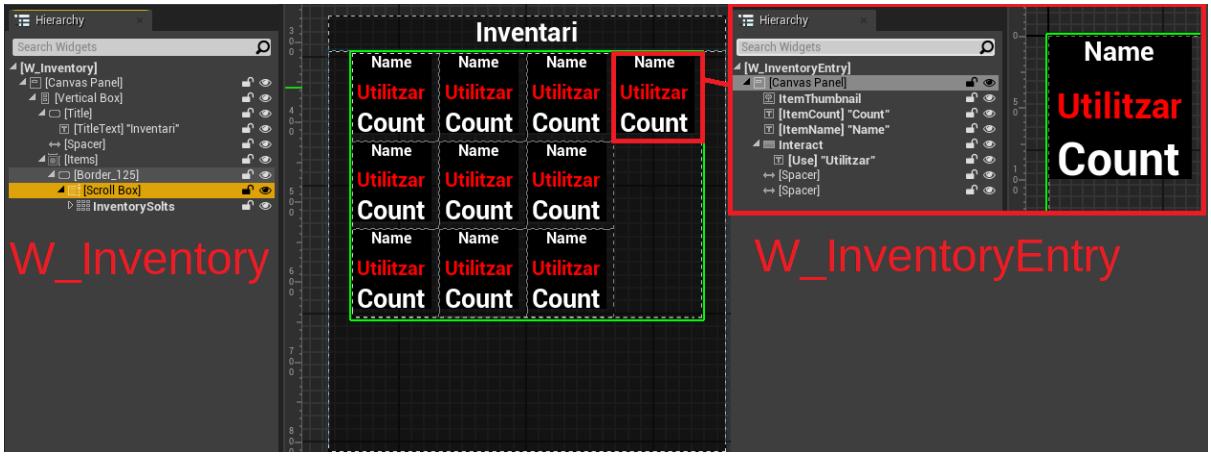


Figura 7.74: Widgets que conformen la interfície de l'inventari

La Blueprint *W_InventoryEntry* s'encarrega de mostrar la informació de cada ítem, i mitjançant un botó permet utilitzar-lo. Com que hi ha ítems que no es poden utilitzar, aquest botó només es mostra si l'ítem seleccionat és utilitzable. Per saber-ho, quan el jugador fa *hover* a sobre d'un d'aquests *Widgets* s'ha de comprovar si l'ítem és utilitzable, i mostrar un missatge al jugador indicant que pot o no utilitzar l'ítem. La Figura 7.75 mostra la implementació dels esdeveniments *OnHovered* i *OnUnhovered* a sobre del botó *Interact*.

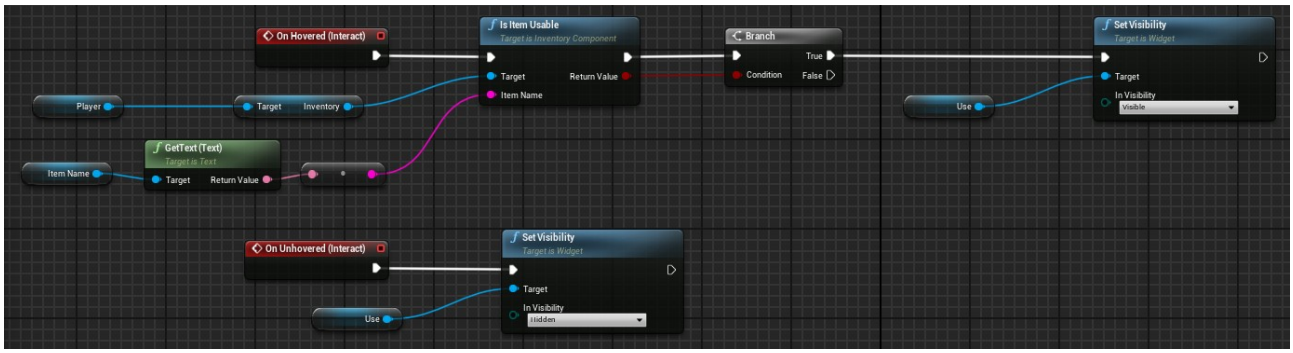


Figura 7.75: Implementació dels esdeveniments OnHovered i OnUnhovered del botó interact

Quan el jugador clica el botó *Interact* s'ha d'utilitzar l'ítem i actualitzar el comptador i la vida del jugador. La funció *UpdateEntryData()* actualitza el valor de la interfície i treu el *Widget* de la interfície en cas que s'hagi utilitzat l'últim ítem emmagatzemat. La Figura 7.76 mostra la implementació de la funció *UpdateEntryData().*

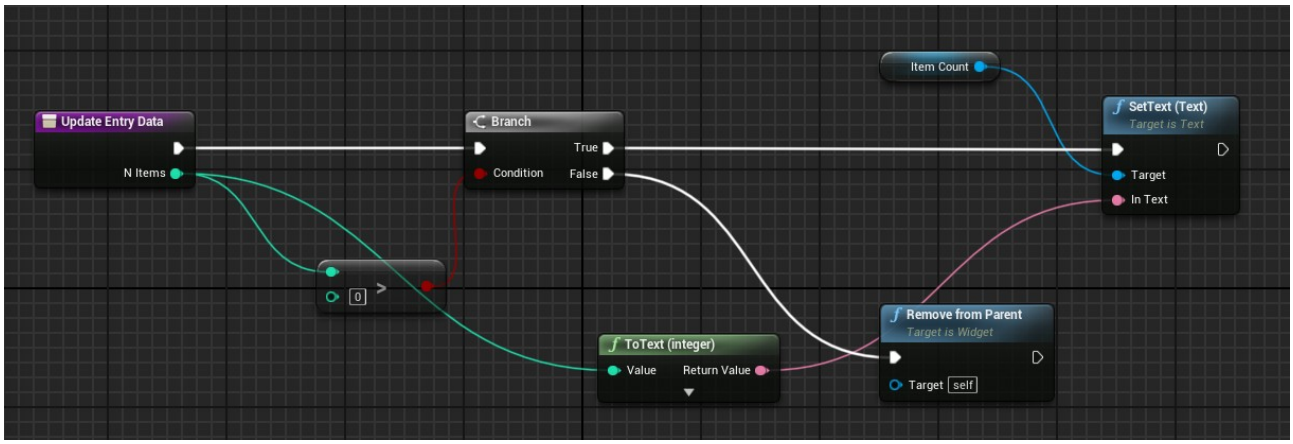


Figura 7.76: Implementació de la funció UpdateEntryData

L'esdeveniment *OnClicked* del botó Interact s'encarrega de cridar al mètode *useItem()* de l'inventari amb el nom de l'ítem seleccionat, i, mitjançant la funció *UpdateEntryData(),* actualitzar el comptador de l'ítem. Finalment, com que alguns tipus d'ítems modifiquen la salut del personatge, s'utilitza la funció *UpdateHealth()* per

actualitzar la barra de vida del jugador amb el possible nou valor de la vida. La Figura 7.77 mostra la implementació de l'esdeveniment *`OnCliked`.*

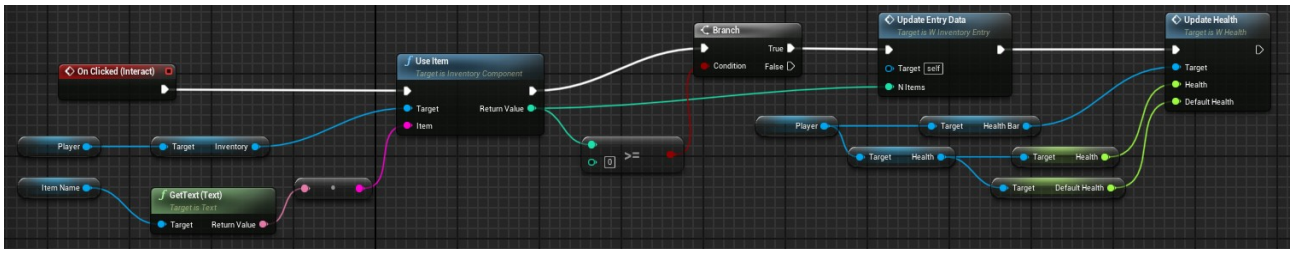


Figura 7.77: Implementació de l'esdeveniment OnClicked

A partir d'aquests Widgets la Blueprint *`W_Inventory`* s'encarrega de recórrer l'inventari, crear una entrada per a cada ítem i posar-la a la graella. Mitjançant la funció *`GetInventoryBlueprintType()`* s'obté un *Map* de Strings i enters que es pot utilitzar des d'aquesta Blueprint per a generar la interfície. Per a recórrer aquest Map s'ha utilitzat la funció del motor *`Keys()`,* que retorna un *Array* amb totes les claus del mapa. A partir d'aquest Array, de l'índex de cada element i una variable *`SlotsPerRow`,* que indica quants ítems es poden mostrar per fila, es calcula la posició de la graella on va cada ítem. Finalment assignen els valors de l'ítem als diferents camps del Widget *`W_InventoryEntry`.* La Figura 7.78 mostra la implementació de la funció *`GenerateSlotWidgets()`.*

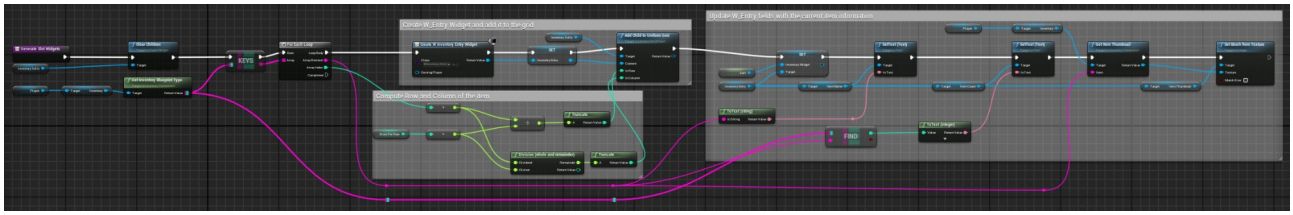


Figura 7.78: Implementació de la funció GenerateSlotWidgets

7.5.2. Inventari d'eines

L'inventari d'eines implementa les funcionalitats bàsiques d'un sistema de gestió d'inventari: afegir o utilitzar ítems, i també implementa funcionalitats per a canviar el mode de treball de les eines. Com que les eines perduren de forma infinita amb el temps, i com que el jugador només pot tenir una eina de cada tipus a l'inventari, s'ha decidit utilitzar un inventari específic per a gestionar només les eines. Aquest sistema s'ha implementat mitjançant components a la classe *`ToolsInventoryComponent`.* El següent fragment de codi mostra la definició d'aquesta classe.

```
class PROJECTE_TFG_API UToolsInventoryComponent : public UActorComponent{
	GENERATED_BODY()

public:
	// Sets default values for this component's properties
	UToolsInventoryComponent();

	UFUNCTION(BlueprintCallable)
	void addToToolsInventory(AToolItem* tool);

	UFUNCTION(BlueprintCallable)
	TArray<FString> GetInventoryBlueprintType() const;

	UFUNCTION(BlueprintCallable)
	void useTool(FString toolName) ;

	UFUNCTION(BlueprintCallable)
	void toolOptions(FString toolName) ;

	UFUNCTION(BlueprintCallable)
	bool findTool(FString toolName) ;

	UFUNCTION(BlueprintCallable)
	AToolItem* getToolRef(FString toolName);

	UFUNCTION(BlueprintCallable)
	bool logToolsInventoryToFile(FString filePath) const;

protected:
	UPROPERTY()
	TMap<FString, FToolInfo> toolsInventory;
	AActor* Owner;
	int nTools = 4;
};
```

Al no haver de gestionar les quantitats d'ítems, la implementació de mètodes com *`use()`* s'ha simplificat molt, ja que no és necessari haver de controlar si l'ítem és *`ItemBundle`*, modificar la quantitat, etc. En aquest inventari només cal comprovar que no existeixi i inserir-lo al Map. El mètode *`toolOptions(FString toolName)`* és l'únic mètode exclusiu d'aquest inventari, i s'encarrega d'accedir a la referència de la eina indicada i executar el mètode *`options()`* que canvia el mode de treball de la eina. El següent fragment de codi mostra la implementació d'aquest mètode.

```
void UToolsInventoryComponent::toolOptions(FString toolName)  {
	FToolInfo* aux = toolsInventory.Find(toolName);
	if (aux != nullptr) {
		if (Owner) {
			aux->toolRef->Options(Owner);
		}
	}
}
```

La integració d'aquest inventari amb el jugador es fa igual que amb l'inventari d'ítems, només que s'ha implementat una nova *Widget Blueprint* encarregada de mostrar només les eines. La Figura 7.79 mostra el disseny d'aquesta Blueprint.

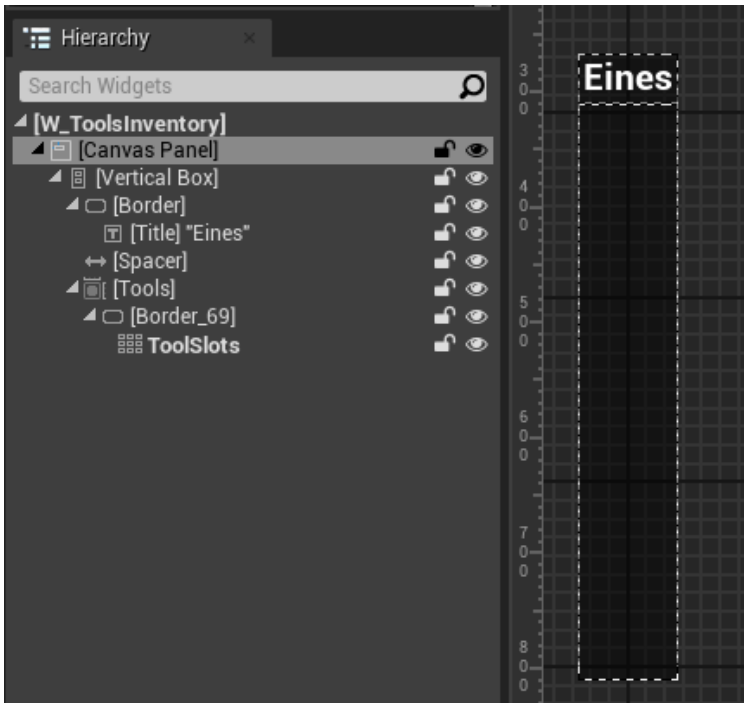


Figura 7.79: Disseny de la interfície de l'inventari d'eines

7.5.3. Integració del sistema de crafteig

El sistema de crafteig està format per un conjunt d'interfícies que connecten els ítems que es poden fabricar amb l'inventari mitjançant receptes. Tot el sistema s'ha construït a partir de tres *Widget Blueprints*:

- *`W_BasicCrafting`:* S'encarrega de mostrar la llista dels ítems que es poden fabricar i la informació necessària per a crear-los.

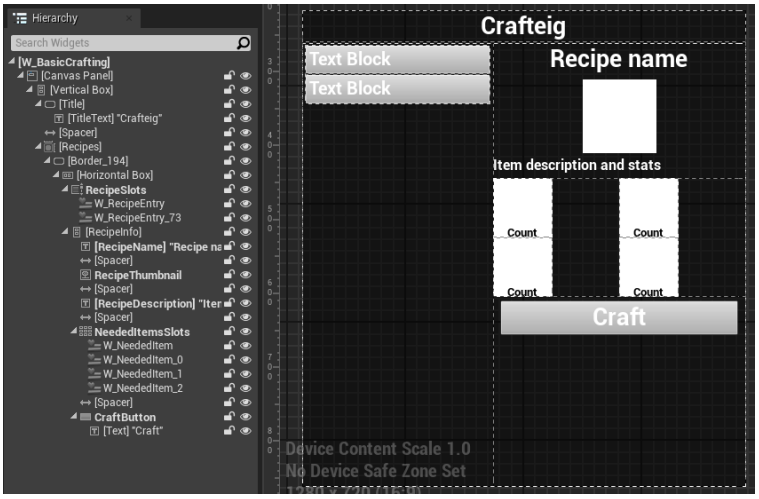


Figura 7.80: Jerarquia i disseny del Widget W_BasicCrafting

- *W_RecipeEntry:* Representa una recepta concreta i s'encarrega d'omplir els camps de la interfície *W_BasicCrafting* amb la informació de la recepta.

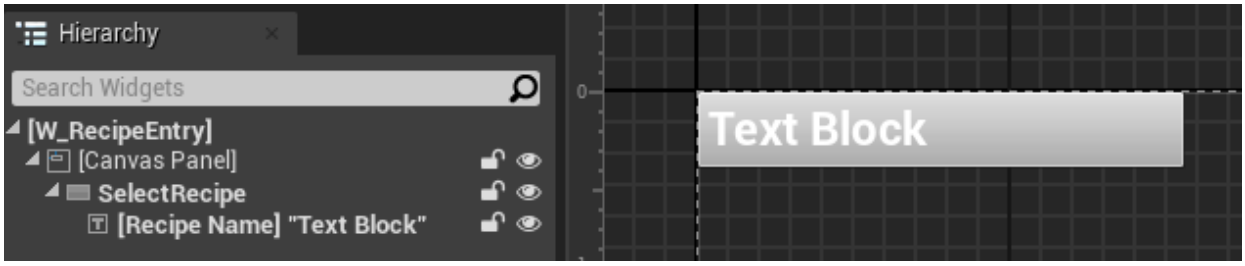


Figura 7.81: Jerarquia i disseny del Widget W_RecipeEntry

- *W_NeededItem:* S'encarrega de mostrar el nom, la miniatura i la quantitat necessitaria d'un recurs.

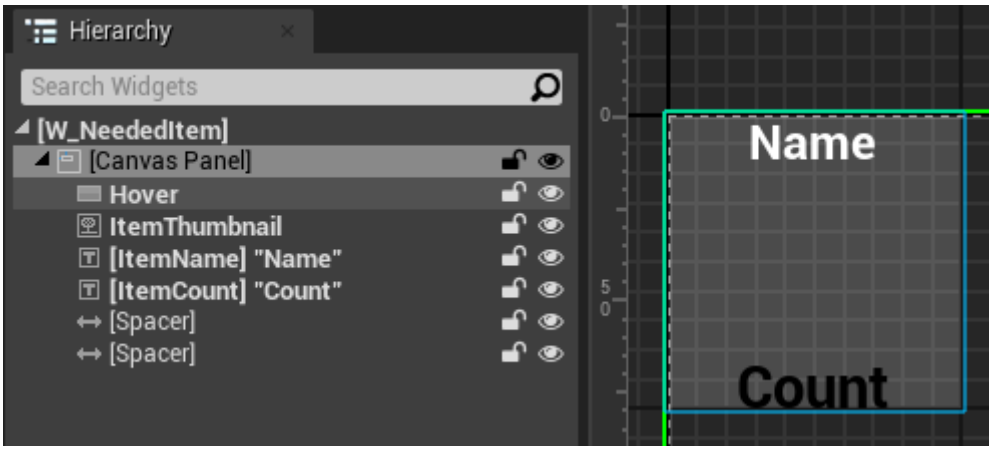


Figura 7.82: Jerarquia i disseny del Widget W_NeededItem

El Widget *W_BasicCrafting* s'encarrega, mitjançant la funció *GenerateRecipeWidgets(int recipesToShow),* de recórrer l'estructura del GameMode que guarda totes les receptes, i genera un Widget del tipus *W_RecipeEntry* amb el nom de cada recepta. El paràmetre *recipesToShow* és un enter que indica quin tipus de receptes es mostraran. Quan acaba de recórrer tota l'estructura mostra la informació de l'última recepta trobada. A la Figura 7.83 es pot observar la implementació d'aquesta funció.

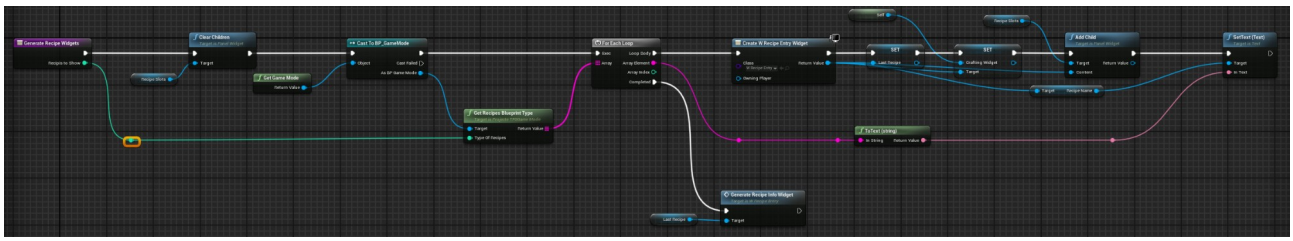


Figura 7.83: Implementació de la funció GenerateRecipeWidgets()

El Widget *W_RecipeEntry* està format per un botó que al clicar-lo mostra la informació de la recepta que representa al Widget *W_BasicCrafting.* La funció *GenerateRecipeInfoWidget()* actualitza la informació, i aprofitant que ha d'accedir a l'inventari per a mostrar les dades dels ítems, emmagatzema en un flag anomenat *CanBeCrafted* si a l'inventari hi ha suficients recursos per a poder fabricar la recepta. Aquesta funció es pot dividir en quatre parts importants:

Inicialitzar les variables necessàries: Abans de generar la informació de la recepta fa falta eliminar els ítems necessaris de la recepta anterior, obtenir la referència de la recepta seleccionada des del GameMode i reiniciar el flag *CanBeCrafted.* La Figura 7.84 mostra la inicialització de les variables.

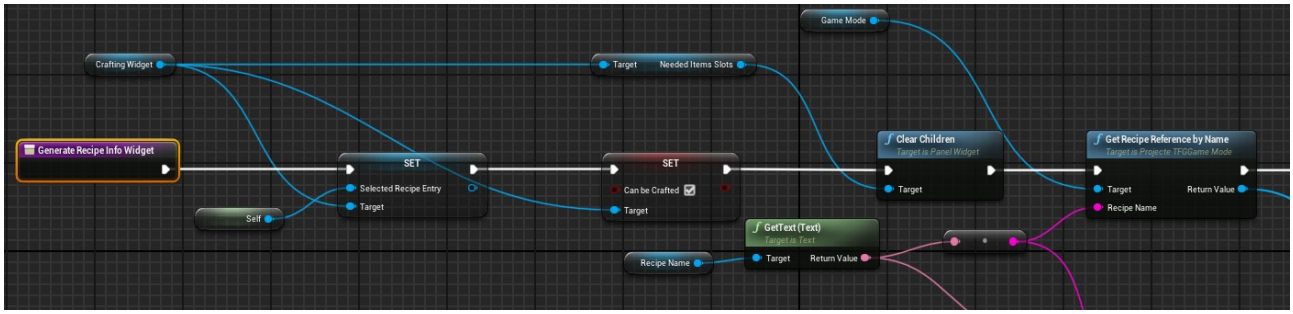


Figura 7.84: Inicialització de les variables

Actualitzar la informació de la recepta (nom, descripció i miniatura): Un cop inicialitzades les dades s'ha d'actualitzar la informació bàsica de la recepta. La Figura 7.85 mostra com s'actualitzen els camps *RecipeName, RecipeThumbnail* i *RecipeDescription.*

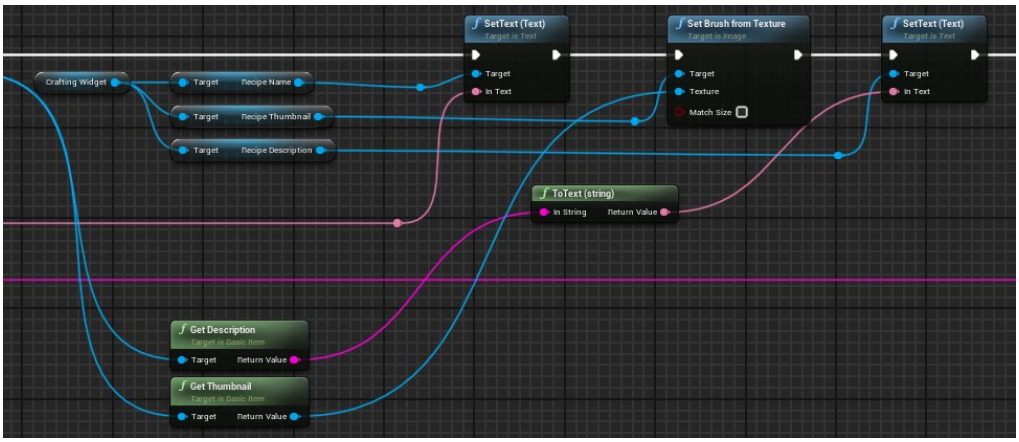


Figura 7.85: Actualització del nom, descripció i miniatura de la recepta

Recórrer els ítems necessaris i crear els Widgets *W_NeededItem:* Les receptes són un *Map* que relaciona els noms dels ítems amb la quantitat necessària per a fabricar els objectes. De forma similar a la UI de l'inventari es generen els *Widgets* amb la informació dels ítems i es posen en una graella. La Figura 7.86 mostra el codi que recorre els ítems necessaris de la recepta i els afegeix a la graella.

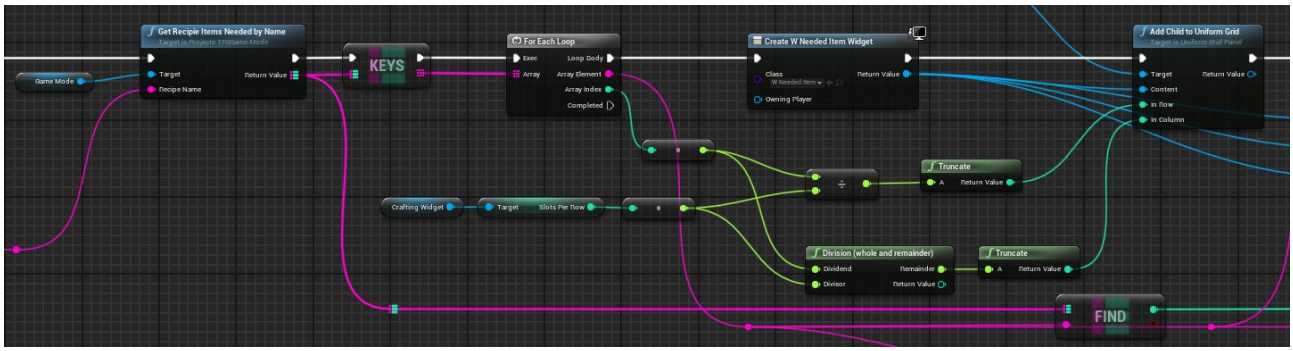


Figura 7.86: Mostrar els ítems necessaris de la recepta

Actualitzar la informació dels ítems i controlar si es pot craftejar o no: Els Widgets `W_NeededItems` creats s'han d'omplir amb la informació dels ítems. En aquest punt es controla, mitjançant la funció de l'inventari `AreEnoughItemsByName()`, si a l'inventari hi ha la quantitat requerida d'ítems. El text que indica el nombre d'ítems necessari es mostra roig o negre per indicar si el jugador té la quantitat necessària. En cas que falti algun ítem el flag `CanBeCrafted` es marca com a false. La Figura 7.87 mostra el codi que controla si la recepta es pot craftejar.

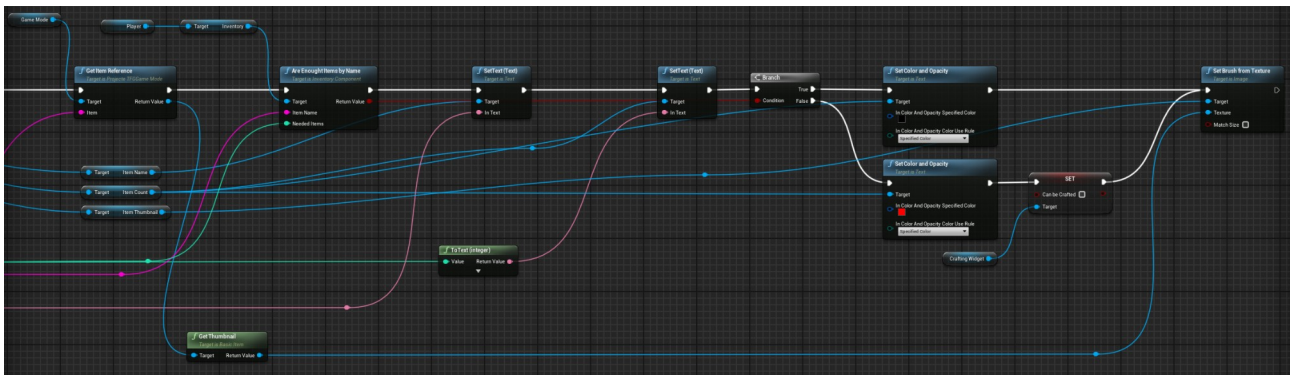


Figura 7.87: Actualitzar informació dels ítems i controlar si es pot craftejar

La interfície `W_BasicCrafting` implementa un botó que al clicar-lo fabrica, si pot, l'objecte indicat a la recepta seleccionada. L'esdeveniment `OnClicked` del botó Craftejar comprova el flag `CanBeCrafted` de la recepta seleccionada, i si és true, mitjançant la funció `CraftItem()` de l'inventari, treu els ítems utilitzats de l'inventari i instancia l'objecte indicat a la recepta a partir de la referència de la classe. Finalment actualitza

l'inventari per a mostrar el nou estat. La Figura 7.88 mostra l'esdeveniment *`OnCliked`* del botó Craftejar.

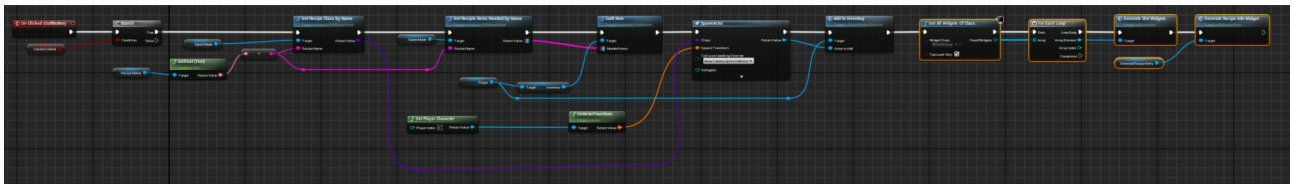


Figura 7.88: Implementació de l'esdeveniment OnCliked del botó Craftejar

7.6. Vida

Seguint amb la idea de fer un projecte sòlid i modular, s'ha decidit implementar el sistema de salut del jugador mitjançant components, permetent que cada Actor del joc pugui implementar un sistema de vida de forma molt ràpida, simplement incorporant aquest component.

El component de vida està format per dos atributs, un que indica la salut màxima de l'Actor i un altre que indica la salut actual. Mitjançant dos mètodes, *`takeDamage(float damage)`* i *`gainHealth(float health)`* es modifica el valor de la vida actual. El següent fragment de codi mostra la definició de la classe *`HealthComponent`*.

```
UCLASS( ClassGroup=(Custom), meta=(BlueprintSpawnableComponent) )
class PROJECTE_TFG_API UHealthComponent : public UActorComponent
{
	GENERATED_BODY()

public:
	// Sets default values for this component's properties
	UHealthComponent();

	UFUNCTION()
	void takeDamage(float damage);

	UFUNCTION()
	void gainHealth(float health);

protected:
	// Reference to the Actor that owns the component
	AActor* Owner;

	UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Health")
	float DefaultHealth;

	UPROPERTY(BlueprintReadOnly)
	float Health;

};
```

La implementació dels mètodes *takeDamage(float damage)* i *gainHealth(float health)* és molt simple, ja que només sumen o resten el valor dels paràmetres a la vida actual del component, i fan un *clamp* perquè el valor no baixi de 0.0 o superi 100.0. El següent fragment de codi mostra la implementació d'aquests dos mètodes.

```
void UHealthComponent::takeDamage(float damage)
{
	float newHealth = Health - damage;
	Health = FMath::Clamp(newHealth, 0.0f, 100.0f);
}

void UHealthComponent::gainHealth(float healthGain)
{
	float newHealth = Health + healthGain;
	Health = FMath::Clamp(newHealth, 0.0f, 100.0f);
}
```

7.7. Objectius

S'ha implementat un senzill sistema que permet definir petits objectius per a guiar al jugador a través de les mecàniques sense necessitat d'un tutorial. Per fer el sistema modular s'ha implementat una estructura formada per objectius i missions, on les missions són un contenidor de petites tasques que s'han de complir. La Figura 7.89 mostra l'estructura de classes d'aquest sistema.

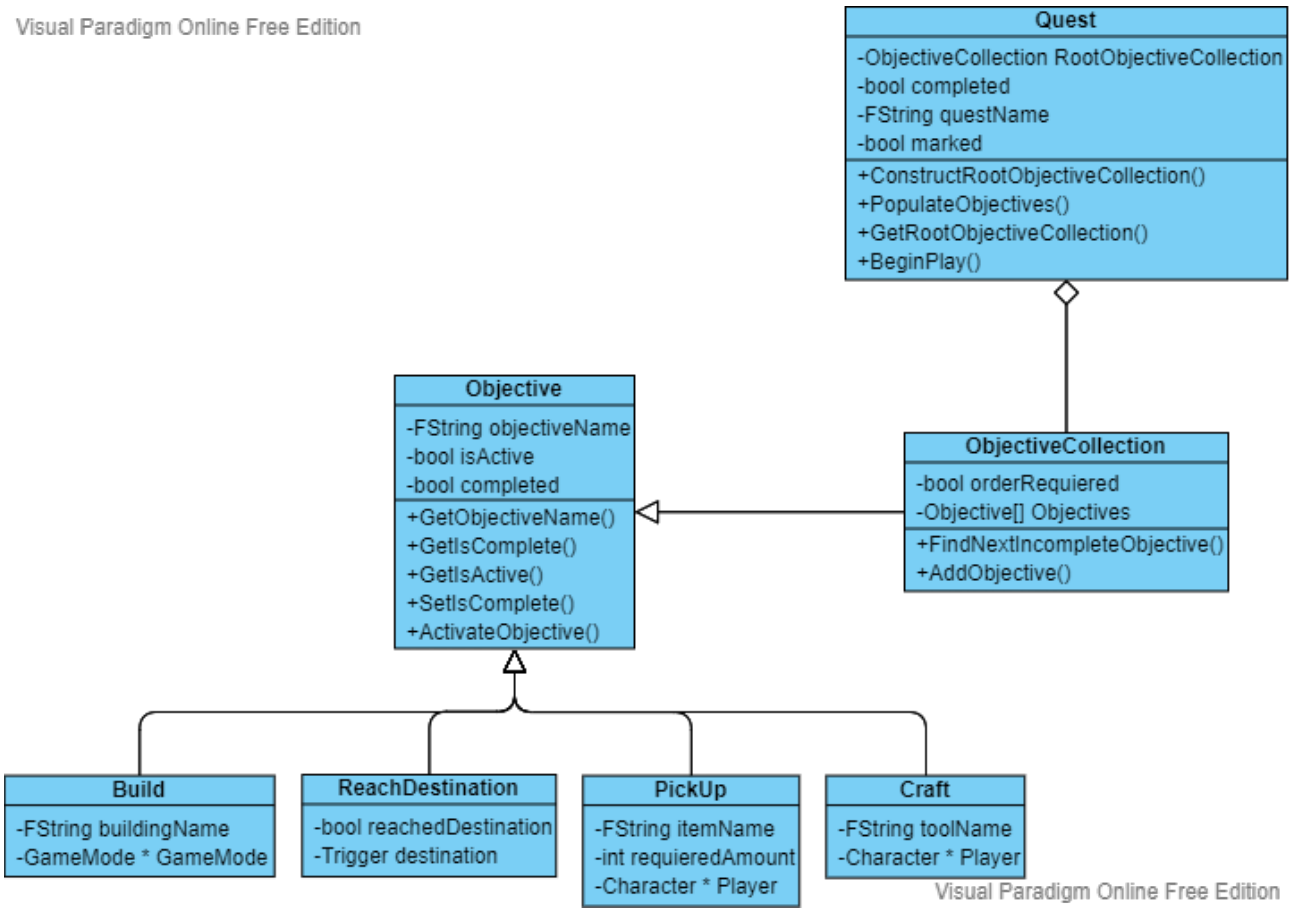


Figura 7.89: Estructura de classes del sistema d'objectius

Els objectius estan formats per un nom i un parell de variables booleanes que determinen si els objectius estan activats o si estan completats. Per a disposar d'un sistema més modular s'han afegit subclasses que representen diferents tipus d'objectius, com per exemple construir alguna cosa o anar a algun punt del mapa. La classe `ObjectiveCollection` implementa un conjunt d'objectius que s'han de completar en ordre o no. Finalment la classe `Quest` conté una col·lecció d'objectius.

Cada tipus d'objectiu sobreescriu el mètode `GetIsComplete()` per a adaptar-lo a les seves necessitats.

- Construir: Accedeix, mitjançant el GameMode, a una estructura de dades que guarda tots els edificis que s'han construït.

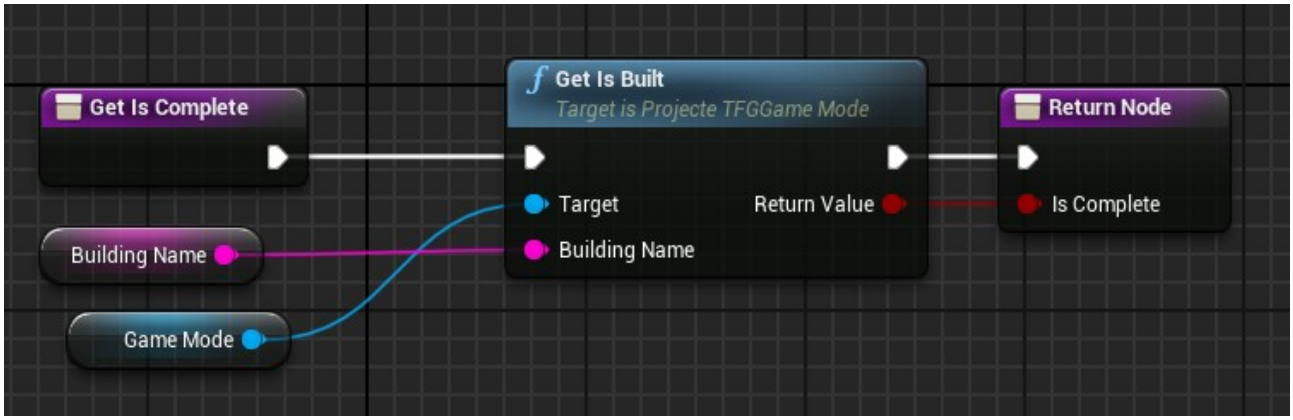


Figura 7.90: Implementació del mètode GetIsComplete per l'objectiu Build

- Arribar a una destinació: Mitjançant una referència al *trigger* que marca el punt on ha d'anar el jugador controla l'esdeveniment `OnBeginOverlap` per saber si el jugador ha entrat a la zona marcada o no.

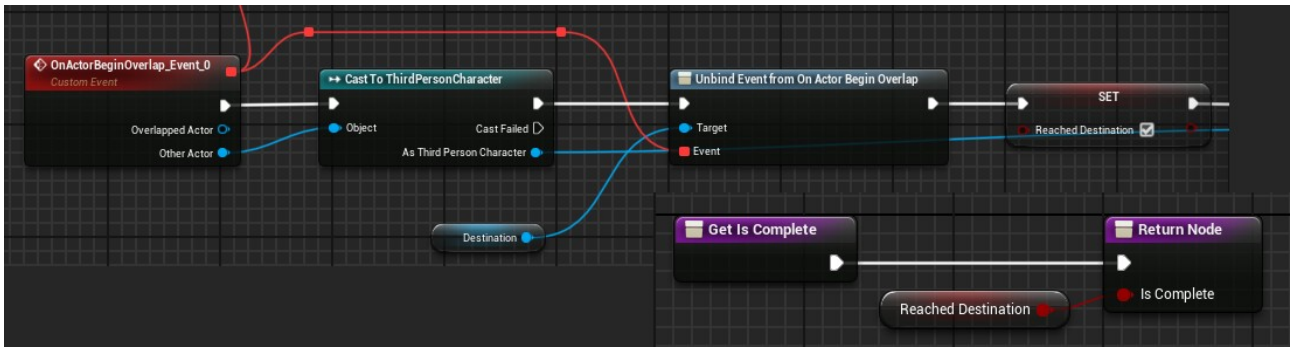


Figura 7.91: Implementació del mètode GetIsComplete i de l'esdeveniment ActorBeginOverlap de l'objectiu ReachDestination

- Agafar ítems: Accedeix a l'inventari i comprova si el jugador té el nombre d'ítems del tipus indicat.

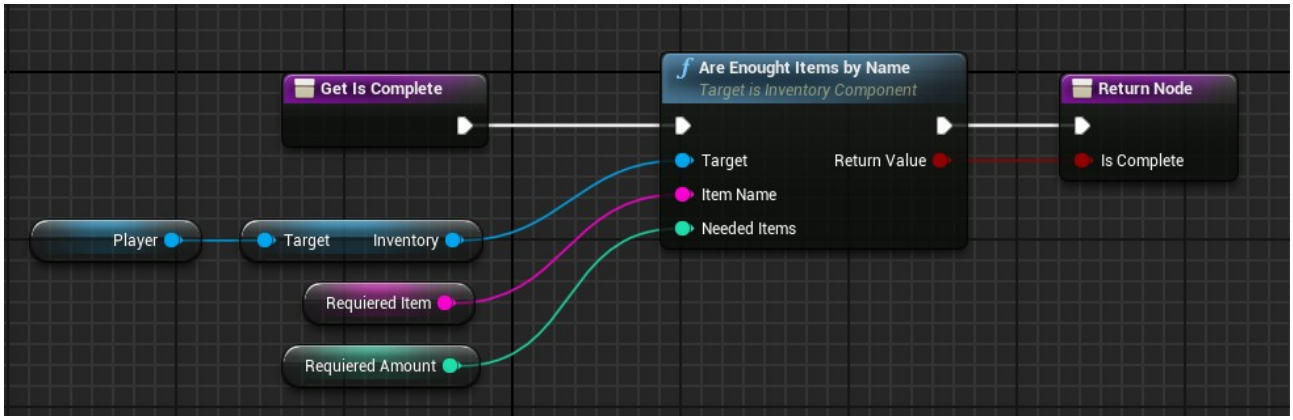


Figura 7.92: Implementació del mètode GetIsComplete per l'objectiu PickUp

- Fabricar eines: Accedeix a l'inventari d'eines i comprova que el jugador tingui l'eina indicada.

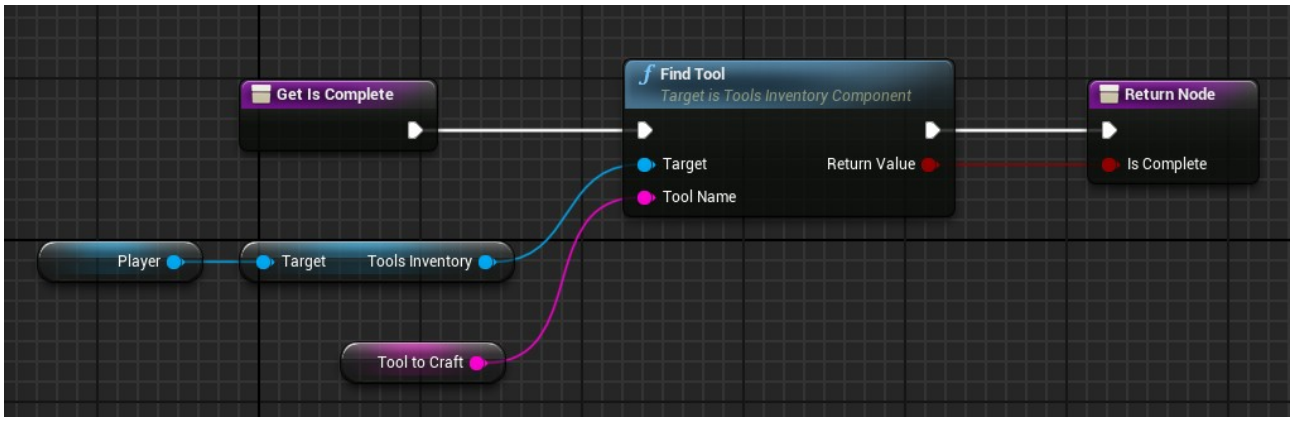


Figura 7.93: Implementació del mètode GetIsComplete per l'objectiu CraftTool

- Col·lecció d'objectius: Recorre tots els objectius de la col·lecció i retorna cert o fals segons si s'han completat tots o no.

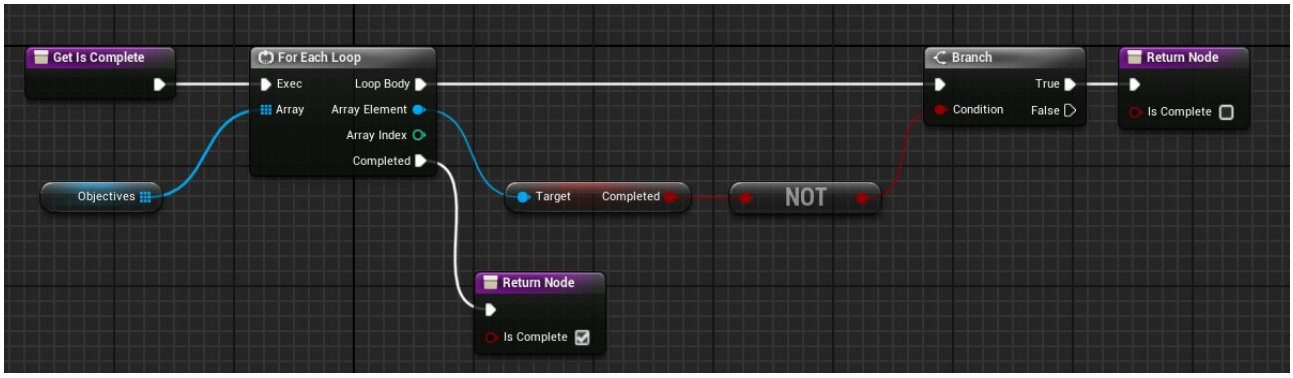


Figura 7.94: Implementació del mètode GetItsComplete per l'objectiu ObjectiveCollection

La classe `Quest` implementa les missions, que són conjunts d'objectius. Per fer-ho cada missió ha de sobreescriure els mètodes `ConstructRootObjectiveCollection()`, que crea l'arrel de la missió, i `PopulateObjectives()`, que s'encarrega de crear els subobjectius. Les Figures 7.95 i 7.96 mostren la implementació d'aquests dos mètodes en una de les missions del joc.

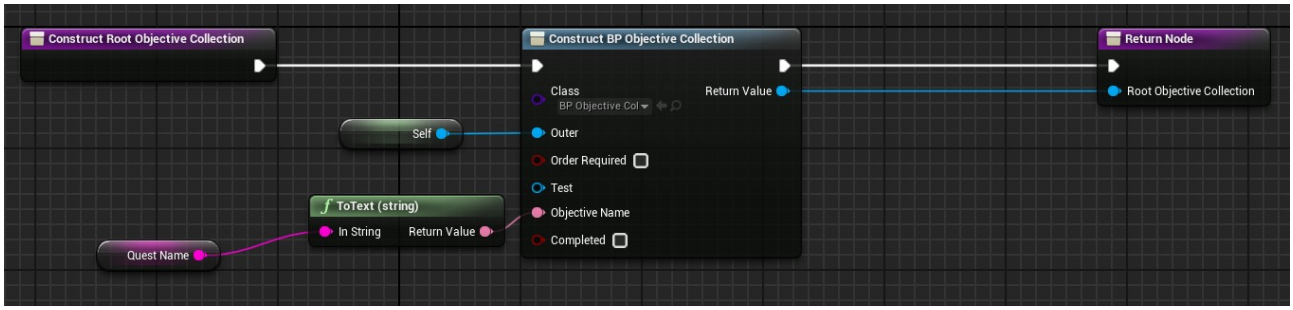


Figura 7.95: Implementació del mètode ConstructRootObjectiveColleciton en una missió del joc

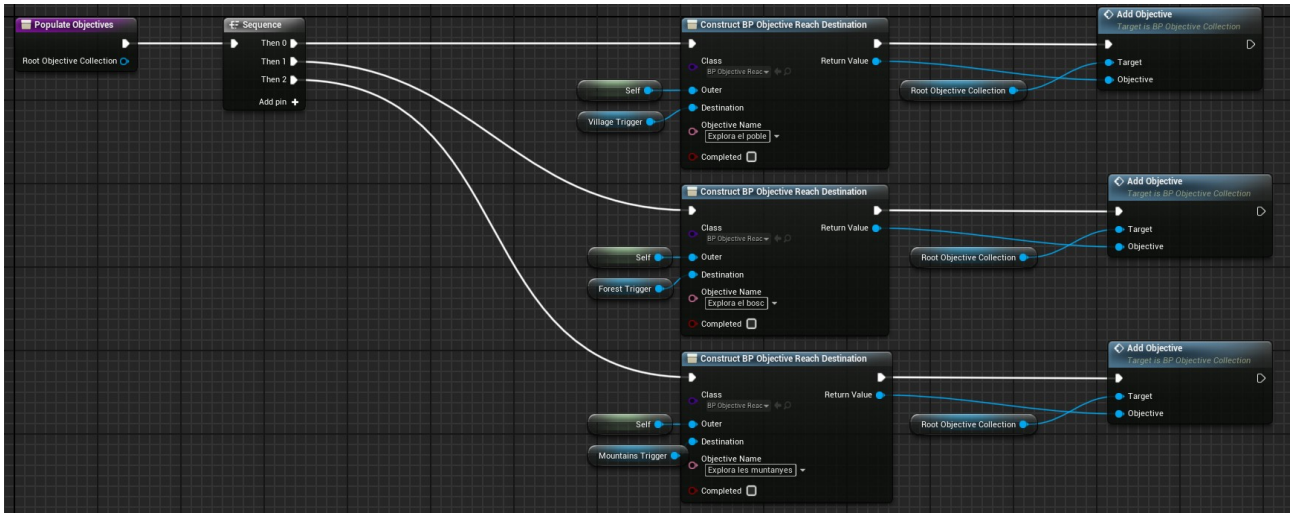


Figura 7.96: Implementació del mètode PopulateObjectives en una missió del joc

Per a crear una nova missió s'ha d'instanciar un objecte de la classe `Quest` i col·locar-lo físicament a l'escenari. Quan el jugador comença la partida el `Character` busca tots els Actors del tipus `Quest` i els emmagatzema en un vector. La Figura 7.97 mostra la part de l'esdeveniment `BeginPlay` on es guarden totes les missions.

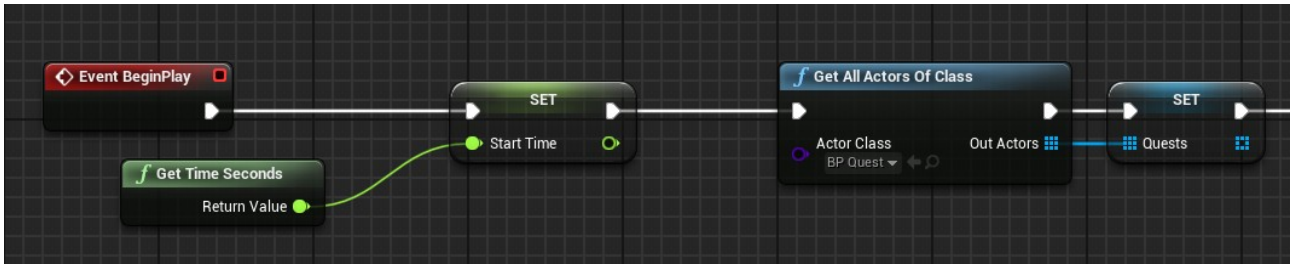


Figura 7.97: Fragment de l'esdeveniment del ThirdPersonCharacter que guarda totes les missions de l'escenari

Per controlar els objectius que va completant el jugador s'han implementat dos mètodes: `CheckForCompletedQuests()`, que revisa totes les missions i marca com a completades aquelles que tenen tots els objectius completats, i `GetRemainingQuests()`, que comprova que si s'han completat tots els objectius i per

tant ha d'acabar el joc. Les Figures 7.98 i 7.99 mostren la implementació de les funcions `CheckForCompletedQuests()` i `GetRemainingQuests()`.

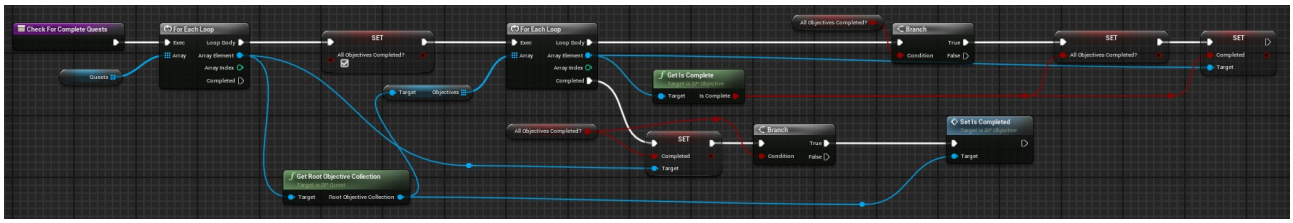


Figura 7.98: Implementació de la funció CheckForCompletedQuests

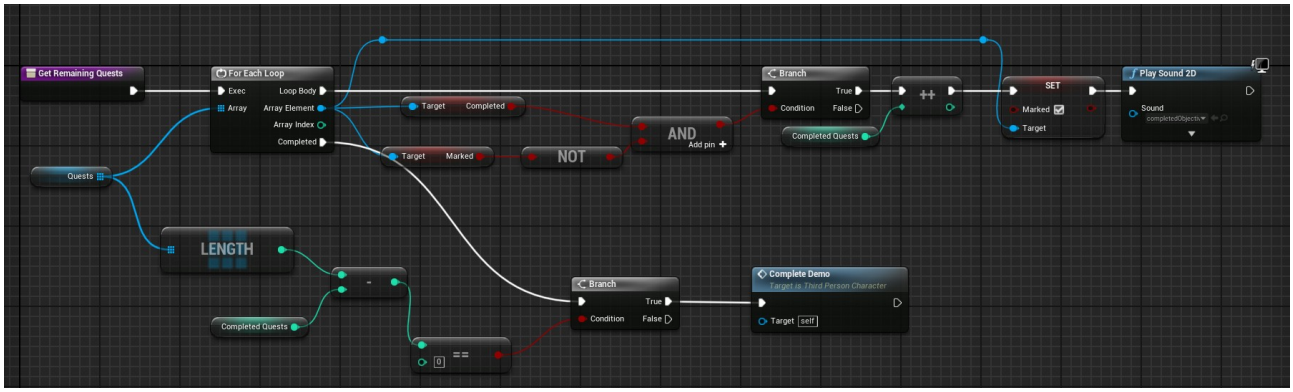


Figura 7.99: Implementació de la funció GetRemainingQuests

7.8. Escenari

L'escenari s'ha implementat mitjançant l'eina de generació de terrenys d'Unreal Engine 4. Un cop creat el terreny amb les dimensions desitjades s'ha esculpit el relleu combinant les eines de soroll (Noise) i de suavitzat (Smooth). La Figura 7.100 mostra el terreny esculpit.

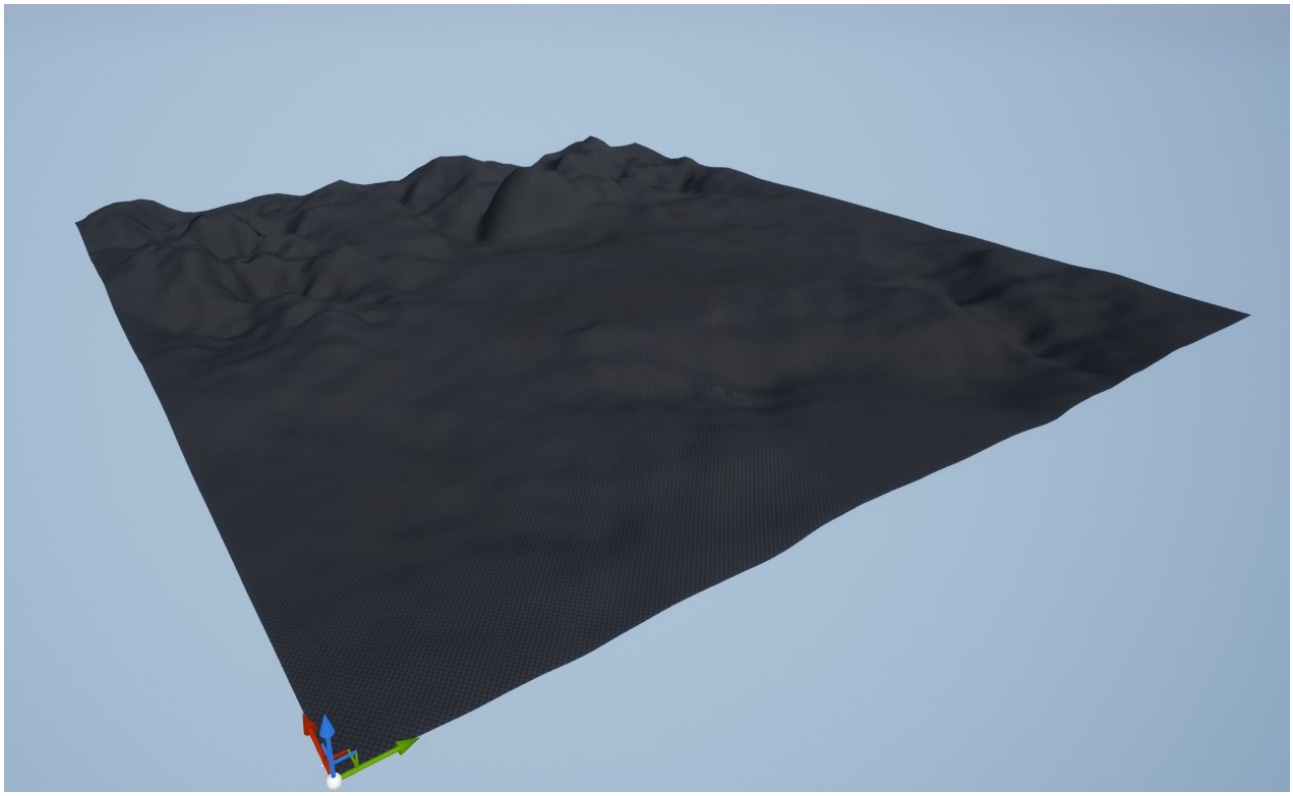


Figura 7.100: Escenari esculpit

A la vida real els escenaris no estan formats només per una única textura, sinó que estan formats a partir de diferents superfícies. Per donar un aspecte més realista al terreny s'ha decidit aplicar diferents textures, concretament una de gespa per a les zones planes, una de roca per a les zones amb molta pendent i un empedrat per a definir camins. Aquestes textures es poden combinar de diferents formes, la més senzilla i ràpida és utilitzant el mode pintar de l'eina de terrenys, que permet aplicar diferents textures de forma manual a sobre del terreny. Aquesta solució és útil en alguns casos, com per exemple a l'hora de dissenyar un camí que uneixi dues zones del mapa, però no ofereix gaires bons resultats aplicada a terrenys grans, ja que manualment s'han d'anar aplicant les diferents textures a cada zona de l'escenari. La solució òptima seria poder aplicar un material i que aquest adaptés la textura segons diferents paràmetres de l'escenari, com l'altura o la pendent de cada punt, permetent texturitzar de forma immediata terrenys de qualsevol mida.

7.8.1. Repetició de textures

Un dels problemes més típics i que treuen més realisme d'un terreny es el *tiling,* o la repetició de les textures. Aquest problema apareix perquè les textures tenen una mida inferior al terreny, i el que fa el motor es repetir-les fins a omplir tot el model. Encara que s'utilitzin textures pensades per a poder ser repetides, aplicar-les en un terreny suposa

moltíssimes repeticions, i acaba donant una sensació estranya al jugador. A la Figura 7.101 es pot observar com si s'aplica directament una textura al terreny generat les zones apareix *tiling* a les zones més llunyanes de la càmera.

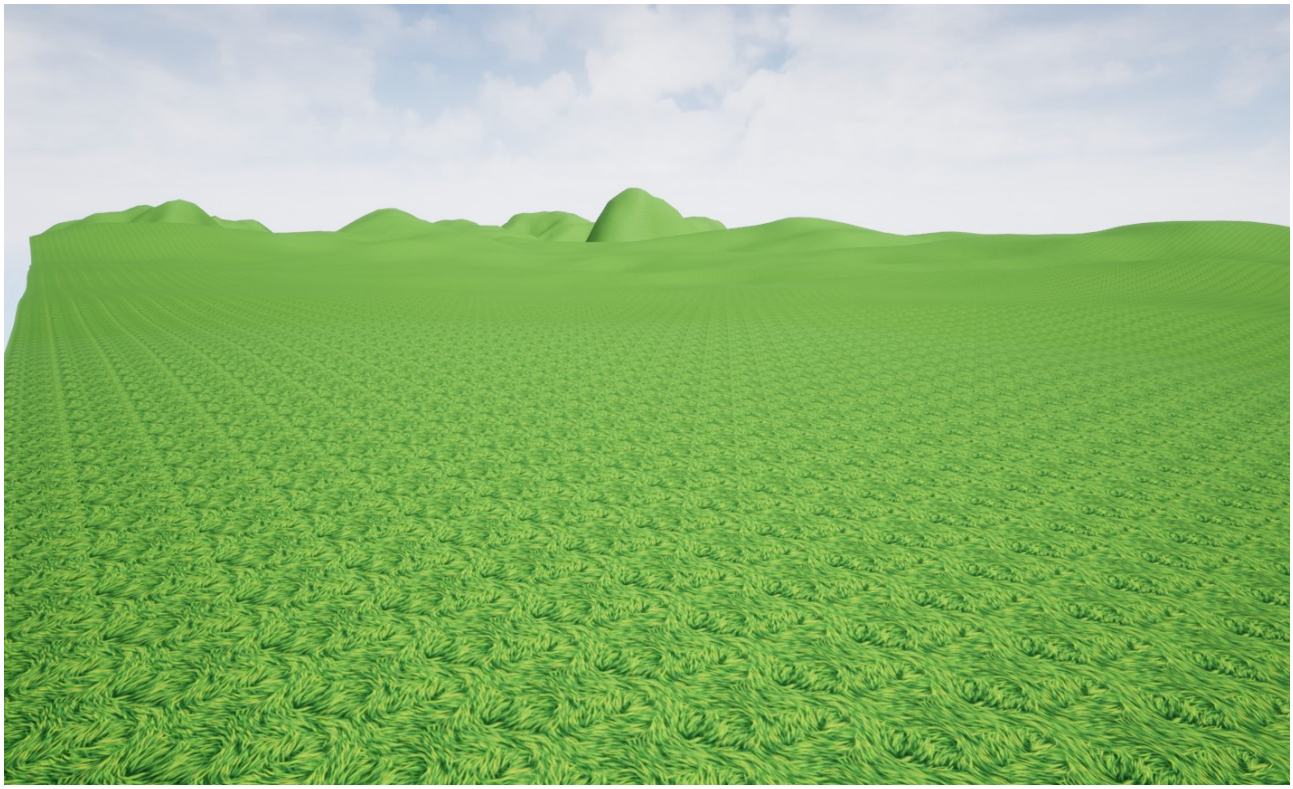


Figura 7.101: Exemple de tiling

Per solucionar aquest problema de repetició una de les tècniques més utilitzades és la d'utilitzar una mateixa textura en una mida gran i en una mida petita, i canviar entre mides segons la distància de la càmera, permetent que les zones allunyades mostrin una textura gran i les zones properes, per no perdre detall mostrin la textura petita. Aquesta tècnica s'anomena *distance tiling.*

Per a generar les textures de diferents mides s'han creat dos paràmetres anomenats `SizeNear` i `SizeFar`, per a poder canviar les mides des de les instàncies del material sense haver-lo de recompilar. Aquests paràmetres es multipliquen per les coordenades del terreny i s'utilitzen com a UV de les textures. Al multiplicar les coordenades per aquests paràmetres es pot modificar l'*UV tiling* per a generar textures de diferent mida. La Figura 7.102 mostra la implementació d'aquesta part del shader, on s'apliquen aquests paràmetres a la mateixa textura de gespa, i al seu mapa de normals.

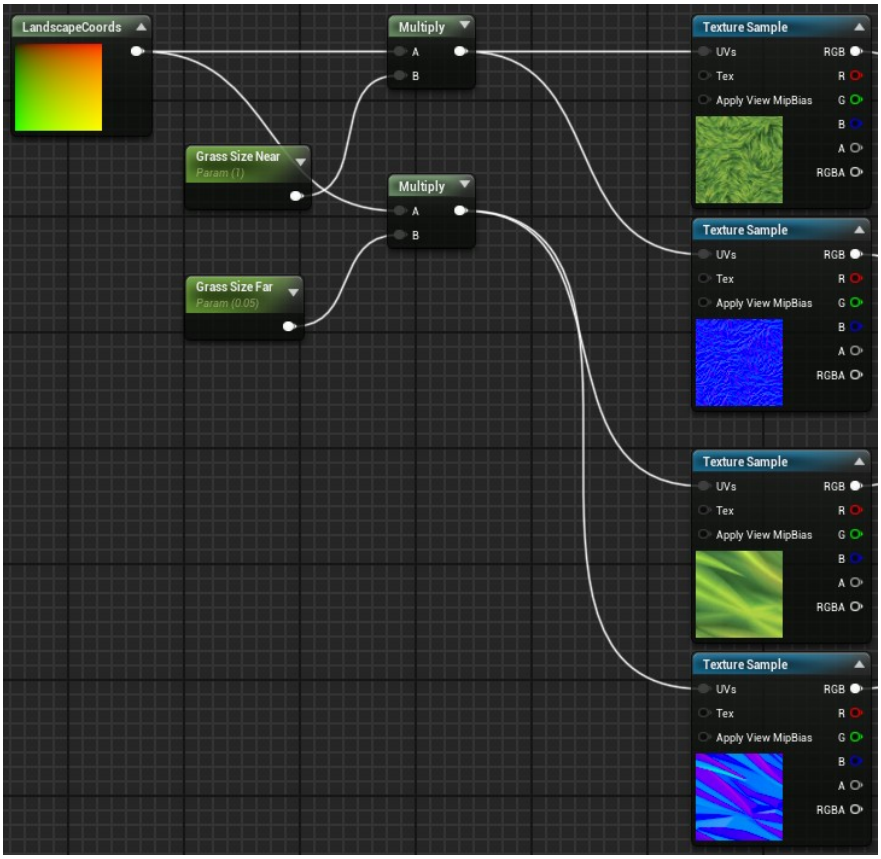


Figura 7.102: Canviar l'UV Tiling a partir de paràmetres

Aquestes textures de diferent mida s'ha de combinar tenint el compte la distància a la càmera. Per fer-ho Unreal Engine 4 implementa un node anomenat `Distance_Blend` que retorna un valor entre 0 i 1 a partir d'un rang i un offset inicial. Aquests valors s'han parametritzat per a poder-los modificar. Les textures s'han fusionat mitjançant una interpolació lineal. El resultat de fusionar les textures i les normals s'ha de retornar al canal corresponent del material. La Figura 7.103 mostra la implementació del material que aplica aquestes variacions a la textura de la gespa.

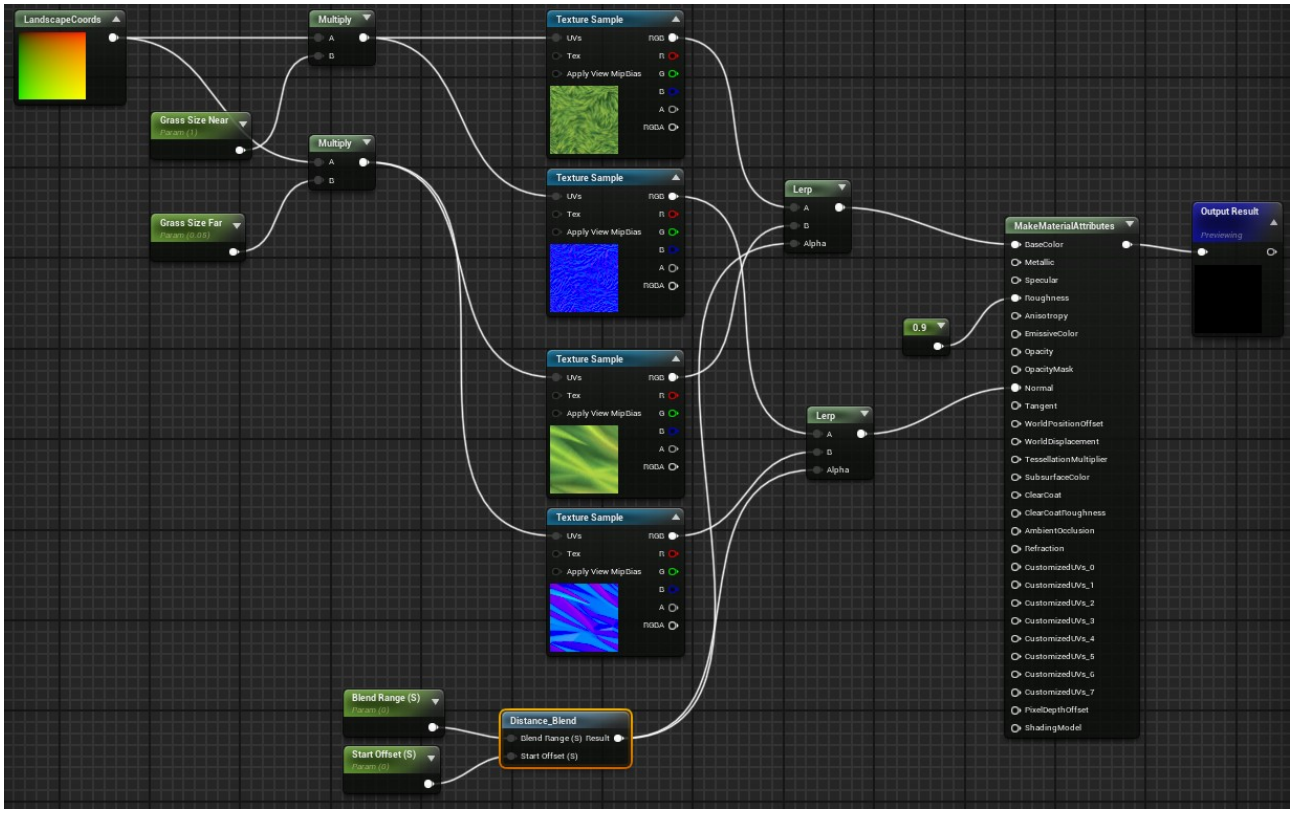


Figura 7.103: Aplicar diferents mides de textures segons la distància

A la Figura 7.104 es pot observar el terreny després d'aplicar aquesta tècnica. Si es compara aquest resultat amb el de la Figura 7.103 es pot comprovar com s'ha reduït el nivell de repetició.

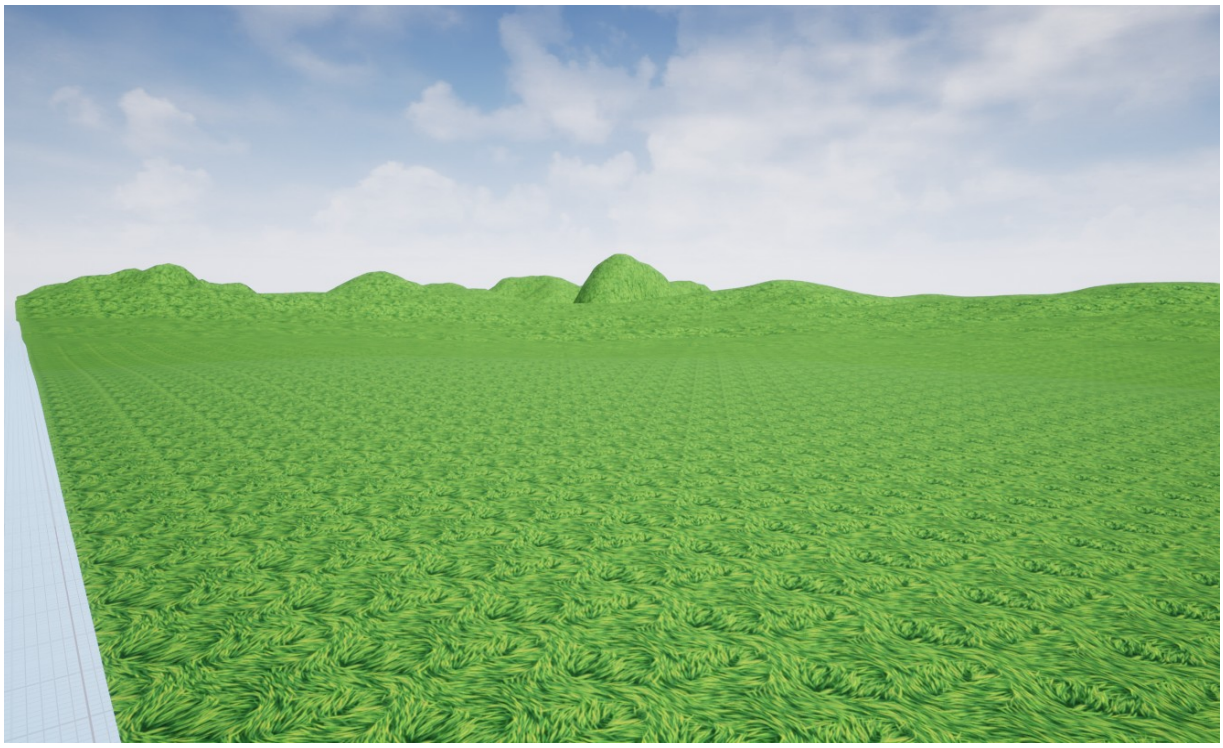


Figura 7.104: Aplicació de la tècnica Distance Tiling

Una altra tècnica molt utilitzada, i que es sol combinar amb la tècnica del *distance tiling,* és la tècnica anomenada *macro texture variation.* Aquesta tècnica consisteix en aplicar una textura que incorpori variacions entre clar i fosc, per després combinar-la amb la textura del terreny per a obtenir diferents intensitats de color repartides de forma aleatòria. Normalment es sol treballar amb una textura que es converteix a tres mides diferents.

Per a generar aquestes textures s'han utilitzat les coordenades de textura multiplicades per un nombre per generar diferents UV Tilings, tal i com s'ha fet a la tècnica anterior. Per evitar que la textura quedi sobreexposada s'ha utilitzat un node *add* amb un valor de 0.5. Finalment, per reduir el contrast de la textura s'ha fet una interpolació del color entre blanc i gris on el paràmetre *alpha* ve determinat per la fusió de les tres textures. La Figura 7.105 mostra la implementació d'aquesta tècnica.

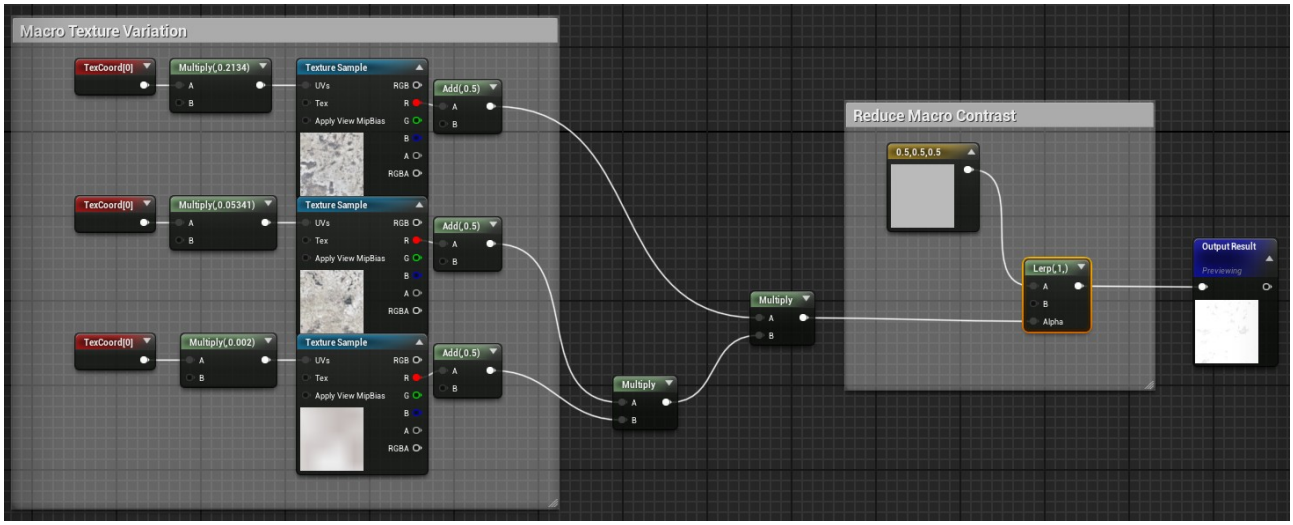


Figura 7.105: Implementació de la tècnica Macro Texture Variation

La Figura 7.106 s'observa com al aplicar aquesta tècnica al terreny la sensació de *tiling* ja és gairebé inexistent.

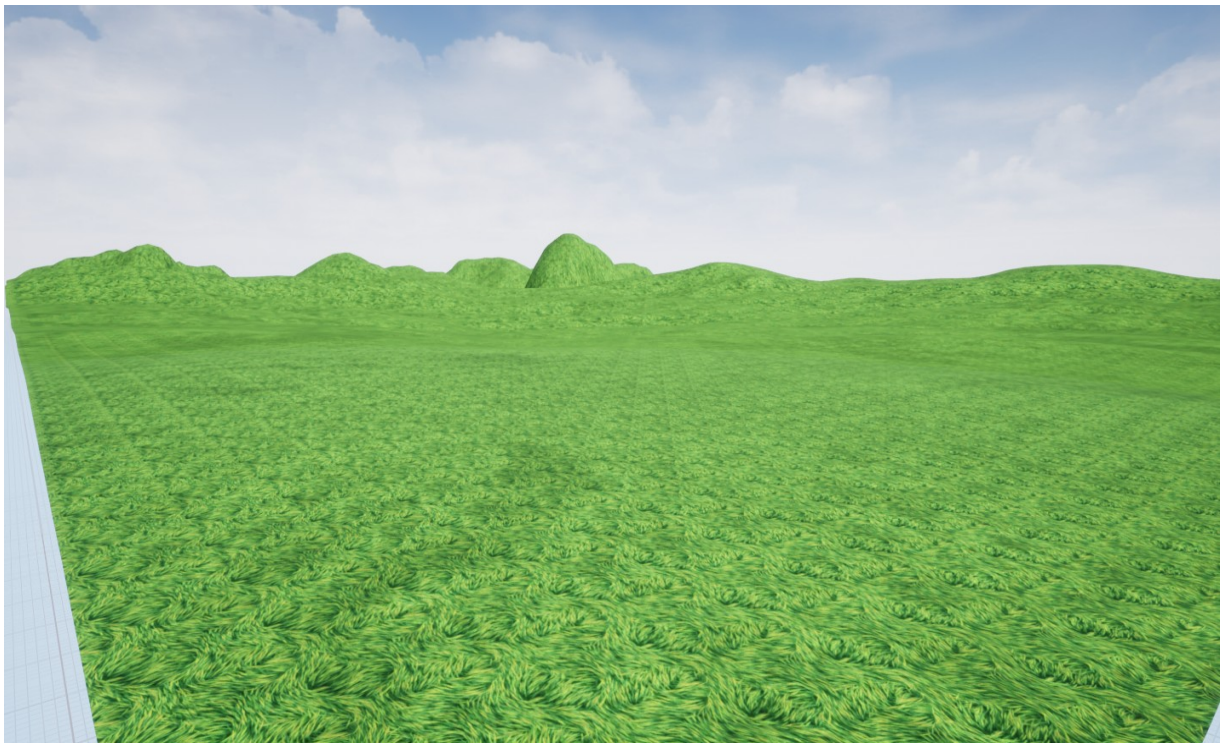


Figura 7.106: Aplicació de la tècnica Macro Texture Variation

7.8.2. Textures automàtiques

En aquest projecte s'ha decidit implementar un material que texturitzi de forma automàtica el terreny a partir de la pendent de cada punt, perquè les zones planes tinguin una textura de gespa i les zones amb desnivell tinguin una textura rocosa. També s'ha aprofitat aquest material per a aplicar models tridimensionals de gespa de forma automàtica només a les zones planes.

L'editor de materials permet crear *Material Funcitons*, que permeten encapsular un conjunt de nodes en un únic node amb paràmetres d'entrada i de sortida. Mitjançant aquestes funcions s'han encapsulat les tècniques anteriors per a cada textura de les utilitzades al terreny.

Encara que el material generi les textures automàticament s'ha de poder pintar a sobre per permetre que els dissenyadors puguin modificar parts concretes del terreny. Per fer-ho s'ha utilitzat el node *`layerBlend`,* que crea capes de textures i les fusiona. Les capes que s'han implementat han sigut:

- Herba: Permet pintar zones d'herba
- Fang: Permet definir camins de terra

- No herba: Permet pintar zones d'herba però sense generar models 3D
- Granja: Permet utilitzar la textura de les zones cultivables però sense crear zones funcionals
- Automàtica: Texturitza el terreny de forma automàtica segons la pendent

Les capes de Herba, Fang i Granja s'han obtingut a partir de les funcions que apliquen la tècnica del *distance tiling* a les seves respectives textures. A les capes Automàtica i No Herba aquesta solució no serveix, ja que el que s'ha de fer es fusionar la textura d'herba i de roca i adaptar-les a la pendent del terreny. Per fer-ho s'ha utilitzat el node *`WorldAlignedBlend.`* Aquest node retorna un valor entre 0 i 1 a partir de dos paràmetres. El primer s'anomena *`Blend Bias`*, i indica la pendent del punt del terreny, i el segon s'anomena *`Blend Sharpness`* i controla com es suavitza la transició entre les dues textures. Per a fusionar les dues capes s'utilitza aquest valor com a *`alpha`* de la interpolació. El blend per defecte d'Unreal Engine no fusiona les normals, donant resultats una mica estranys. Per a solucionar-ho s'ha implementat un *`customBlend`* que fusiona tots els atributs de dos materials. La Figura 7.107 mostra la implementació d'aquest fragment del material.

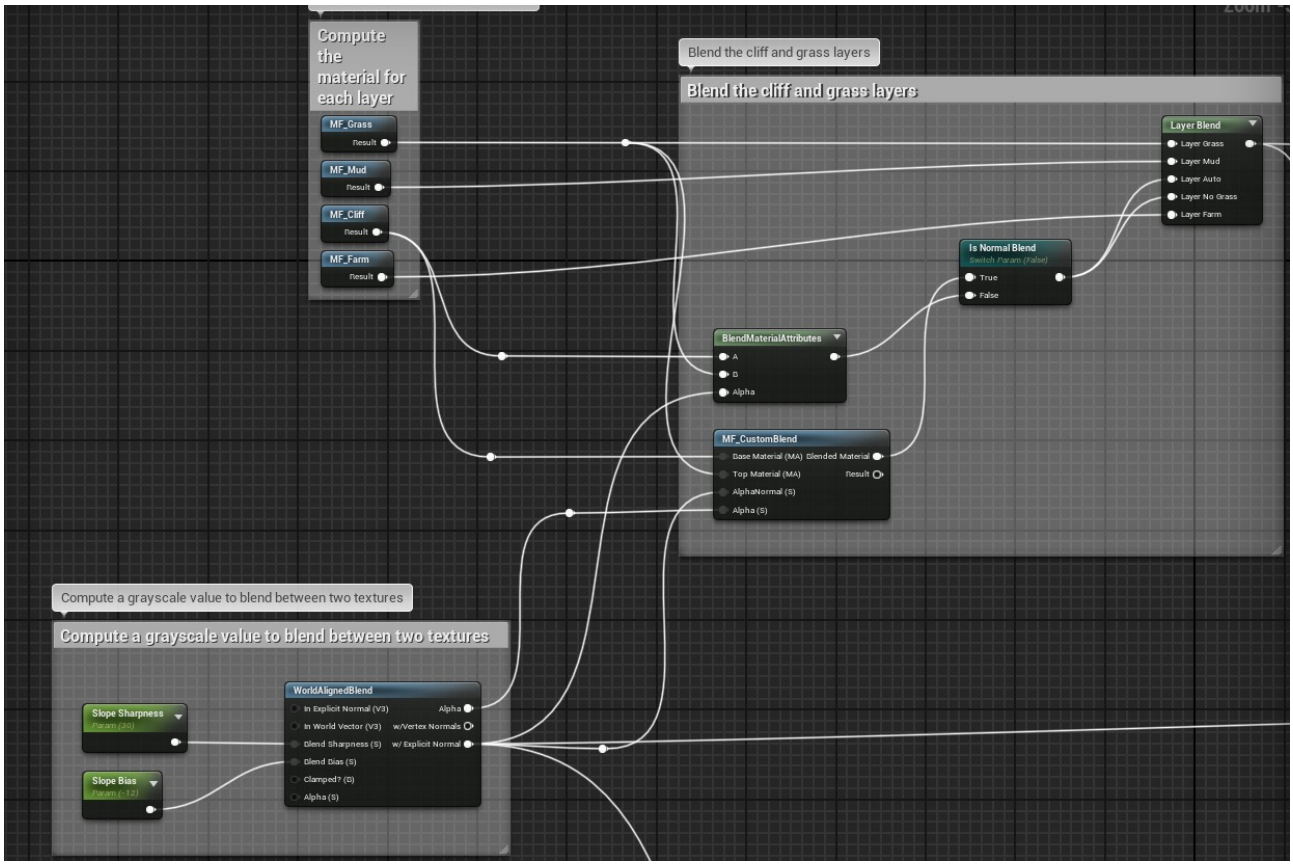


Figura 7.107: Implementació de les diferents capes del material

A la Figura 7.108 es pot observar una comparació entre el blending per defecte d'Unreal Engine i el blending personalitzat fusionant també les normals. Com es pot comprovar, el blending d'Unreal Engine 4 genera unes zones petites de herba dintre de les pendents.

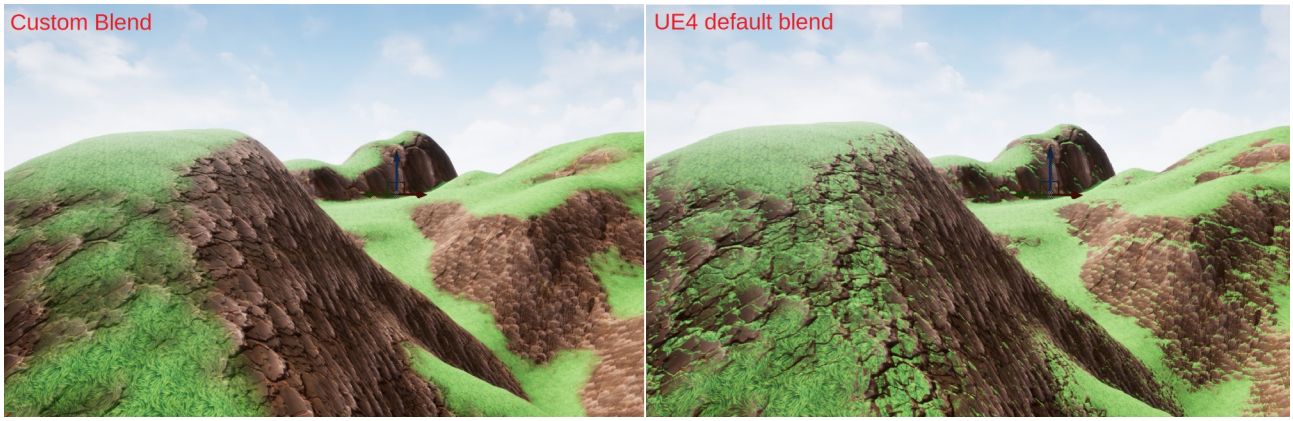


Figura 7.108: Comparativa entre els blending default d'UE4 i el blending custom

El node `LayerBlend` retorna una estructura del tipus `MaterialAttributes` que conté tots els atributs d'un material. A l'atribut `baseColor` resultant s'ha aplicat la tècnica de la *Macro Texture Variation* per tal que afecti a sobre de totes les capes del terreny. La Figura 7.109 mostra el codi que realitza aquesta tasca.

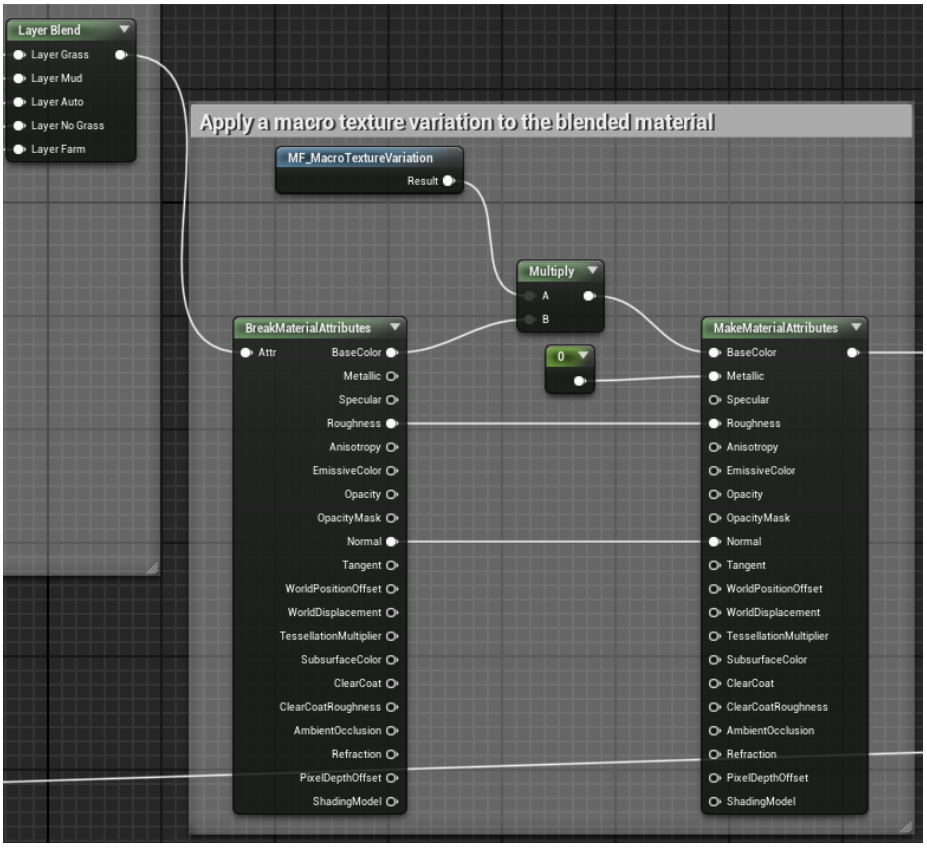


Figura 7.109: Aplicar la tècnica de la Macro Texture Variation al material resultant de fusionar les capes

Com s'ha explicat en punts previs, quan el jugador utilitza l'aixada per a crear zones de cultiu dibuixa a sobre d'un RenderTarget, que representa el terreny, les àrees on es pot cultivar. Un cop s'ha obtingut el material definitiu del terreny fa falta aplicar a sobre de les zones cultivables la textura corresponent, perquè el jugador pugui veure en quines parts del mapa pot plantar. Per fer-ho s'ha de mirar per cada pixel del terreny si està en una zona blanca del RenderTarget. Mitjançant el node *AbsoluteWorldPositon* s'obté la posició al món del pixel actual, d'aquesta posició es pot descartar la component Z, ja que s'està treballant en textures bidimensionals, i dividint la X i la Y per la mida del mapa s'obtenen les coordenades de textura del punt. El resultat s'ha de saturar per a deixar-lo entre 0 i 1, i poder-lo utilitzar com a *alpha* al fer un blend entre el material resultant del terreny i el material de les zones de cultiu. Aquest blend té un problema, ja que permet crear zones de cultiu a tot el mapa, i a la realitat no es pot plantar en zones amb molta pendent. La Figura 7.110 mostra un exemple d'una zona de cultiu en una paret vertical.

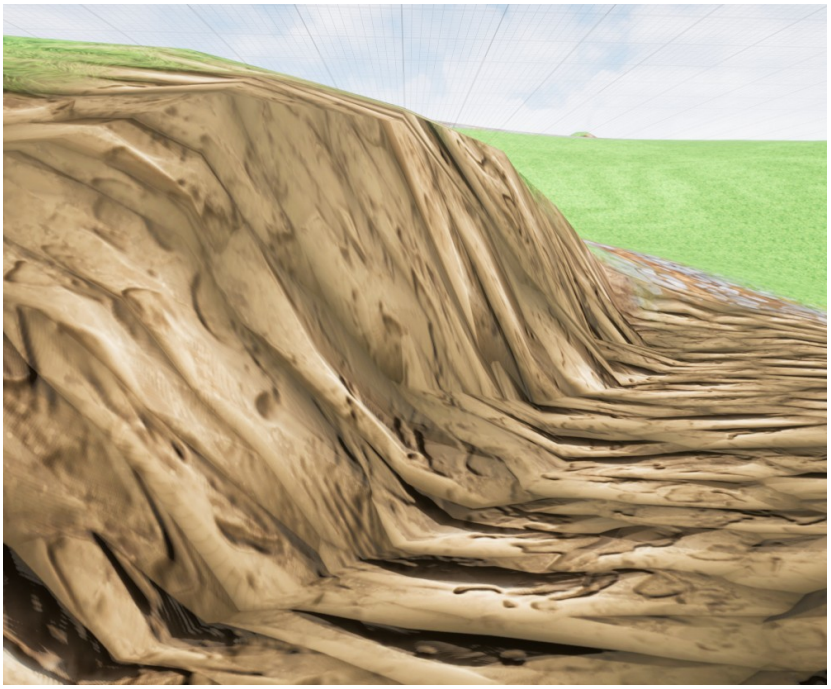


Figura 7.110: Problema amb la pendent al crear zones de cutliu

Per a solucionar aquest problema s'ha utilitzat el resultat del node *WorldAlignedBlend* per a fer que les zones de cultiu només es puguin pintar en zones on hi ha gespa. La Figura 7.111 mostra com s'utilitza el RenderTarget per a aplicar la textura de les zones cultivables a sobre de les textures del terreny.

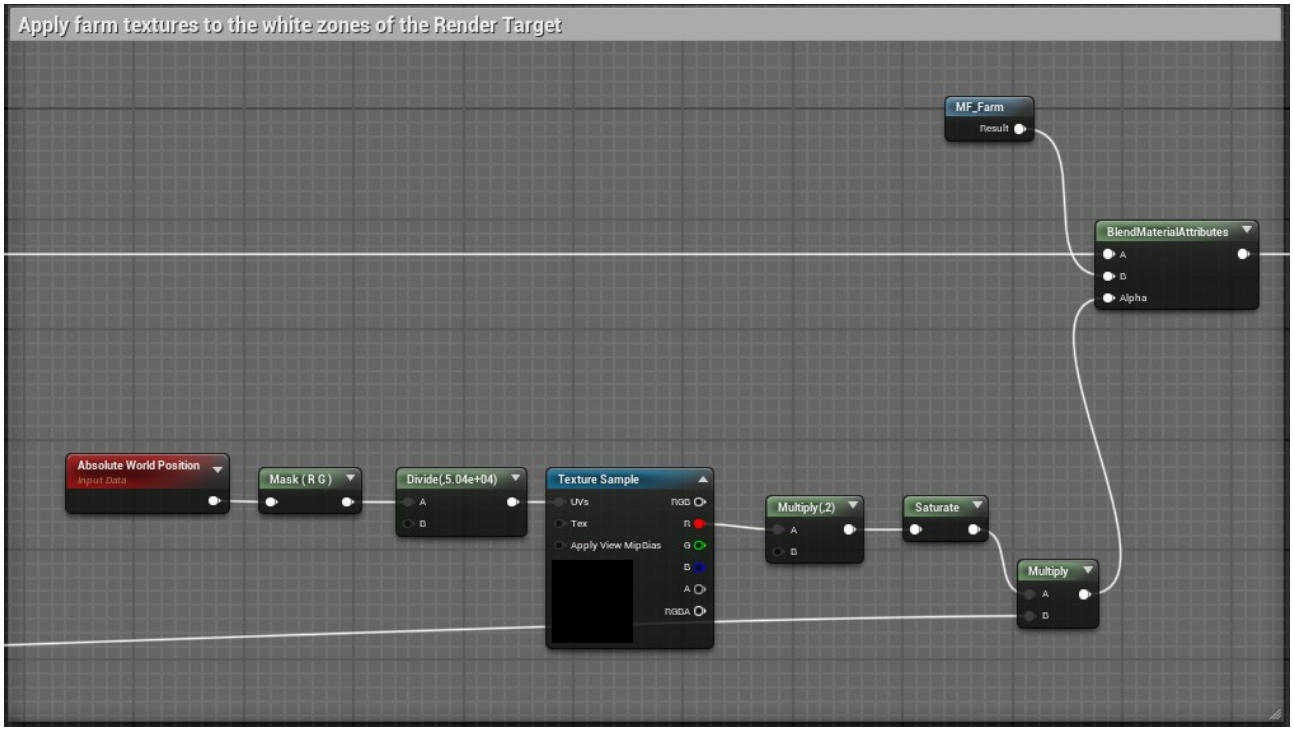


Figura 7.111: Aplicar les textures de les zones cultivables a sobre del terreny

La Figura 7.112 mostra com queda la mateixa zona de cultiu tenint en compte la pendent.

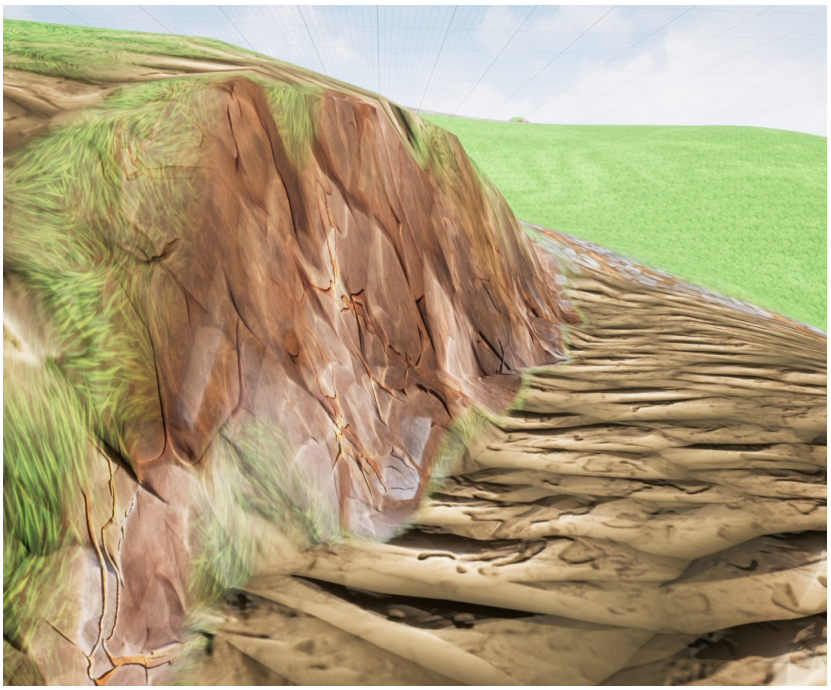


Figura 7.112: Problema amb la pendent al crear zones de cultiu

Aprofitant que s'ha implementat aquest material que texturitza automàticament el terreny s'ha decidit afegir-li una funcionalitat que li permeti instanciar models d'herba de forma automàtica en determinades capes. Per fer-ho s'ha utilitzat el node *`GrassOutput`*, al qual

se li passa una textura i un objecte del tipus *LandscapeGrassType,* que emmagatzema en un vector models tridimensionals. A partir d'aquests elements instancia els models a les zones definides de la textura. La Figura 7.113 mostra el *LandscapeGrassType* que s'ha creat per aquest projecte, i es pot observar com és un vector d'una posició amb un model d'herba i diferents paràmetres per configurar com apareix, com poden ser la densitat o l'escala.

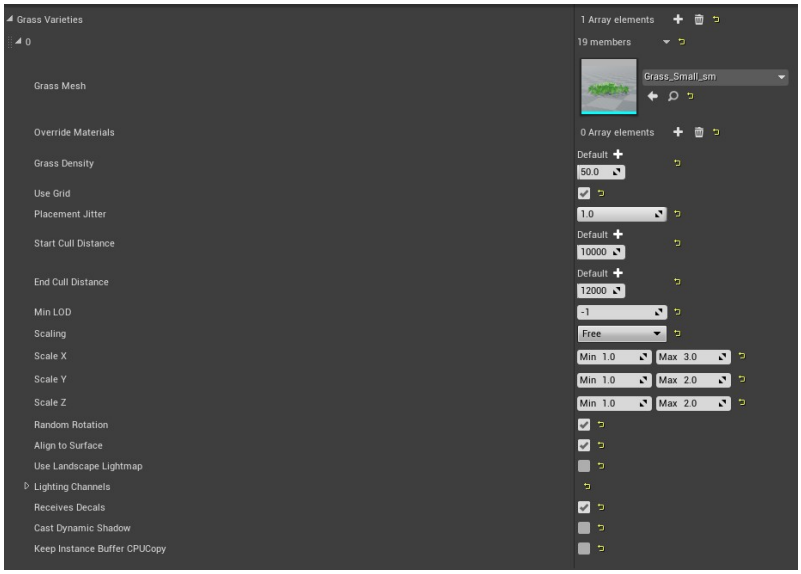


Figura 7.113: Landscape Grass Type

Com que només s'ha de posar herba a les zones de poca pendent també s'ha utilitzat la sortida del node *WorldAlignedBlend,* però a més a més s'ha restat el valor de les capes que no han de tenir herba, que són les capes No Herba, Fang i Granja. La Figura 7.114 mostra els nodes que implementen aquesta funcionalitat.

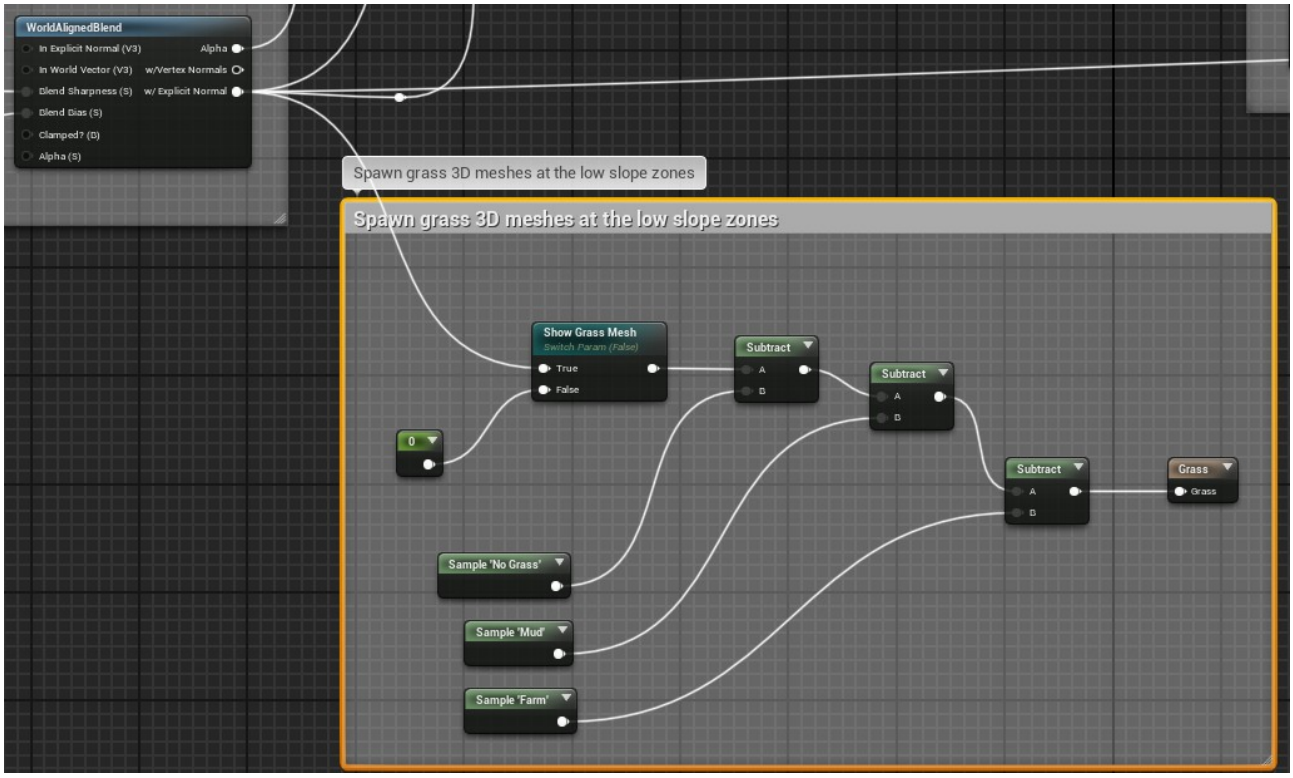


Figura 7.114: Posar models d'herba de forma automàtica a les zones planes

La Figura 7.115 mostra l'aplicació de herba només a les zones planes.

Figura 7.115: Herba col·locada a les zones planes

Finalment, la Figura 7.116 mostra el material automàtic resultat.

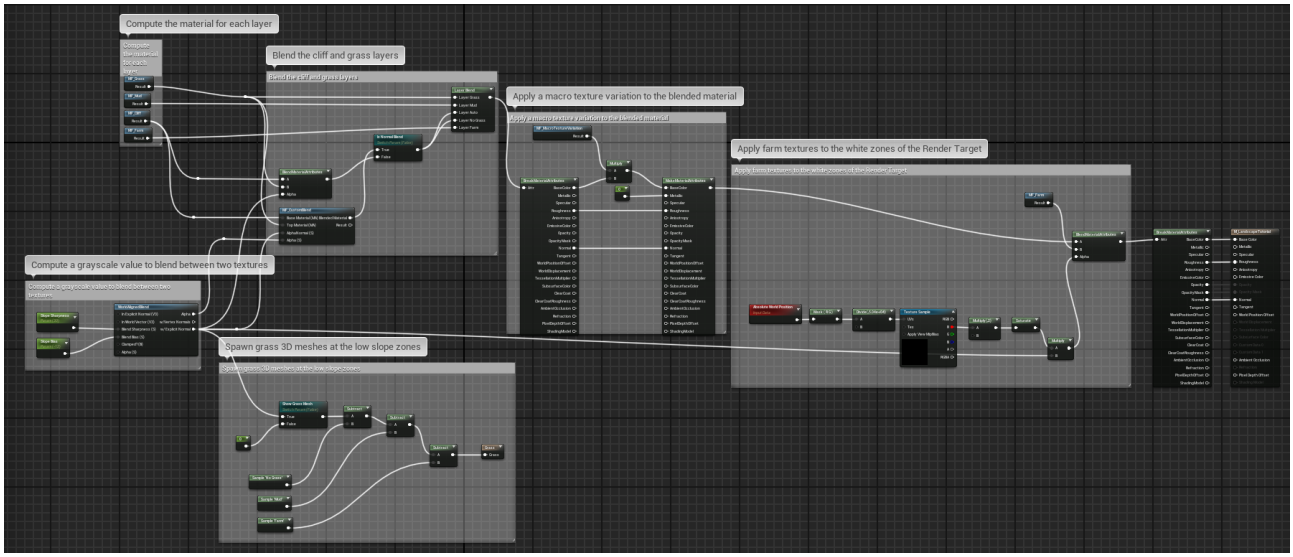


Figura 7.116: Material del terreny amb texturització automàtica

7.8.3. Col·locar models a partir d'Actor Foliage

Al ser un escenari gran no era viable col·locar tots els objectes com arbres, pedres, etc de forma manual. Per fer-ho hi havien dues opcions: utilitzar els models dels arbres i roques amb el node `GrassOutput` o utilitzar l'eina de Foliage d'Unreal Engine 4, que permet col·locar amb un pinzell models a sobre d'un terreny. S'ha decidit utilitzar la segona solució, ja que el node `GrassOutput` només pot col·locar static meshes, i en aquest projecte s'havien de col·locar objectes del tipus Actor, ja que tots els elements que conformen l'escenari són ítems i permeten que el jugador hi interactuï. Per a poder utilitzar els ítems del joc des de l'editor de Foliage s'han hagut de crear un tips especial d'Actors anomenats `ActorFoliage`, que són uns objectes que són reconeguts des de l'editor de Foliage i permeten vincular qualsevol classe que hereti d'Actor. La Figura 7.117 mostra l'ActorFoliage creat per un tipus d'arbre.

Figura 7.117: ActorFoilage creat per un tipus d'arbre

S'han creat aquests tipus d'Actors per tots els objectes que s'havien de col·locar al mapa, i mitjançant el pinzell s'han anant aplicat a diferents zones els objectes necessaris. Per exemple, a la part de muntanyes s'ha incrementat la intensitat de pedres i s'ha reduït la d'arbres. La Figura 7.118 mostra l'editor de Foliage amb tots els objectes creats.

Figura 7.118: Tipus d'ActorFoliage implementats al joc

7.8.4. Resultat final

A la Figura 7.119 s'observa l'escenari resultant obtingut després d'aplicar totes les tècniques mencionades en aquest apartat:

Figura 7.119: Escenari final

7.9. Tècniques d’optimització

Al tenir una estètica LowPoly el joc no té una càrrega computacional elevada, però tot i això s’ha decidit utilitzar algunes tècniques d’optimització, més per veure com funcionen, que per necessitat del propi joc.

7.9.1. Culling

El *culling* consisteixen en no renderitzar tots els objectes d’una escena per a treure càrrega a la targeta gràfica. Hi ha diferents mètodes de *culling,* però la base de tots es renderitzar només els objectes que veu el jugador. Unreal Engine 4 implementa quatre mètodes de *culling,* però en aquest projecte només s’han utilitzat els mètodes de distància i de visió de la càmera.

El *culling* a partir de distància permet determinar a partir de quina distància es veuen els diferents models. Aquesta tècnica es volia implementar en els models de l’herba i dels arbres, però finalment només s’ha pogut aplicar a l’herba, ja que els arbres al ser Blueprints no suporten aquesta tècnica. A la Figura 7.120 es pot observar una comparativa entre els models renderitzats utilitzant diferents distàncies de *culling*.

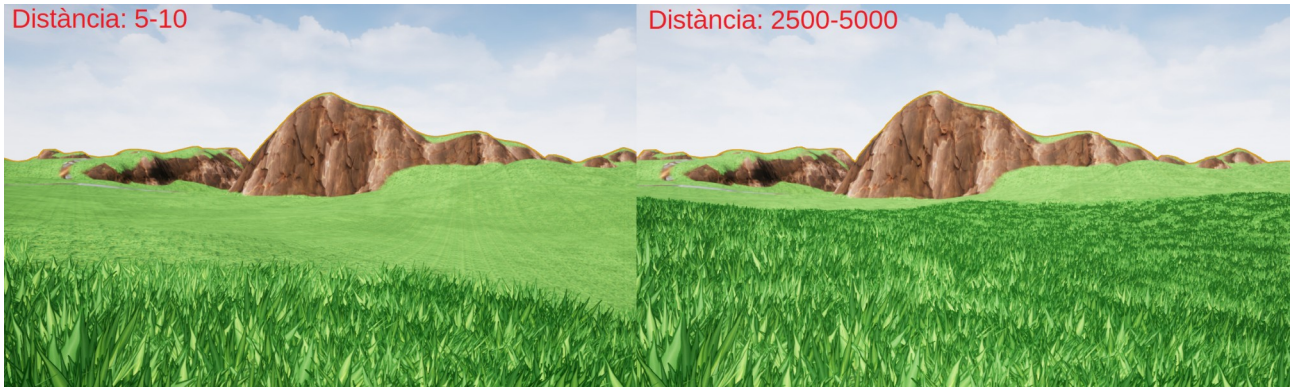


Figura 7.120: Culling a partir de distància

El mètode del *View Frustum* només renderitza els objectes que estan dintre del camp de visió (Field of View) de la càmera. La Figura 7.121 mostra el que es veu mitjançant la càmera del joc i el que s’està renderitzant realment.

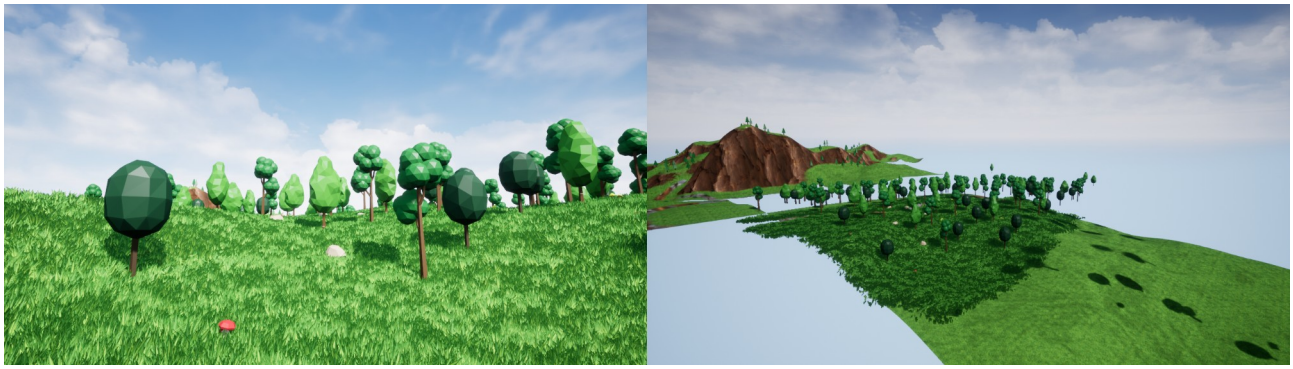


Figura 7.121: View Frustum culling

7.9.2. World Composition

World Composition és una eina pensada per a treballar amb escenaris molt grans. Per defecte Unreal Engine 4 treballa amb *persistent levels,* i carrega tota la informació d'un escenari a la memòria RAM del sistema. En escenaris petits això no suposa cap problema, però si es vol treballar amb un món molt gran aquest mètode no es útil, ja que hi haurien molts elements en memòria que estarien lluny del jugador. *World Composition* permet dividir un escenari en diferents subnivells independents, aportant avantatges tant des del punt de vista de la implementació com de la optimització.

A Unreal Engine 4 cada nivell s'emmagatzema en un fitxer, fent que sigui impossible que dues persones estiguin treballant sobre el mateix nivell simultàniament, però si enlloc de tenir un nivell es creen subdivisions, diferents desenvolupadors poden treballar en un mateix escenari de forma simultània.

Pel que fa a la optimització, mitjançant *World Composition* es pot fer *Level Streaming*, que consisteix en constantment carregar i descarregar de memòria els subnivells que compleixen unes determinades condicions respecte al jugador, com per exemple la distància a la que es troben.

L'escenari d'aquest projecte és de la mida per defecte dels terrenys d'Unreal Engine 4, i no suposa cap problema d'optimització, ja que l'escenari cap en poca memòria, però tot i això s'ha decidit implementar les diferents zones del mapa com el poble o la cabanya del bosc com a subnivells. A la Figura 7.122 es pot observar la jerarquia de nivells, on el terreny està al P*ersistent Level, és* a dir, que existeix a tots els altres nivells, però el poble, la zona inicial i la caseta del bosc estan en nivells independents.

Figura 7.122: Jerarquia de nivells del joc

7.9.3. Timers vs EventTick

La majoria de motors de videojocs implementen funcions que s'executen cada frame del joc. En algunes situacions és necessari utilitzar-les, com per exemple per actualitzar les físiques d'un objecte, però moltes vegades s'acaben utilitzant aquestes funcions per a fer comprovacions que no caldria fer a cada frame.

Si considerem que el joc s'executa aproximadament a 60 Frames Per Segon aquestes funcions es cridaran 60 vegades cada segon. Si molts objectes executen codi cada frame el joc anirà molt més lent.

Tot i que utilitzar aquestes funcions en la majoria de casos té un cost molt petit en el rendiment del joc, una bona pràctica és intentar utilitzar-les el menys possible. En aquest projecte no s'ha utilitzat l'esdeveniment *EventTick,* excepte per els objectes del propi motor que l'implementen, i en canvi s'han utilitzat timers per a executar el codi menys vegades.

Per demostrar-ho s'han implementat dos esdeveniments que imprimeixen el text «Function Called» per pantalla, i s'ha cridat un des de `l'EventTick` i l'altre mitjançant un timer cada 0.2 segons. La Figura 7.123 mostra en blau les crides des del timer i en roig les crides des de `l'EventTick`, i es pot observar com entre cada execució del timer s'ha executat 12 vegades el codi de l'*Event Tick.*

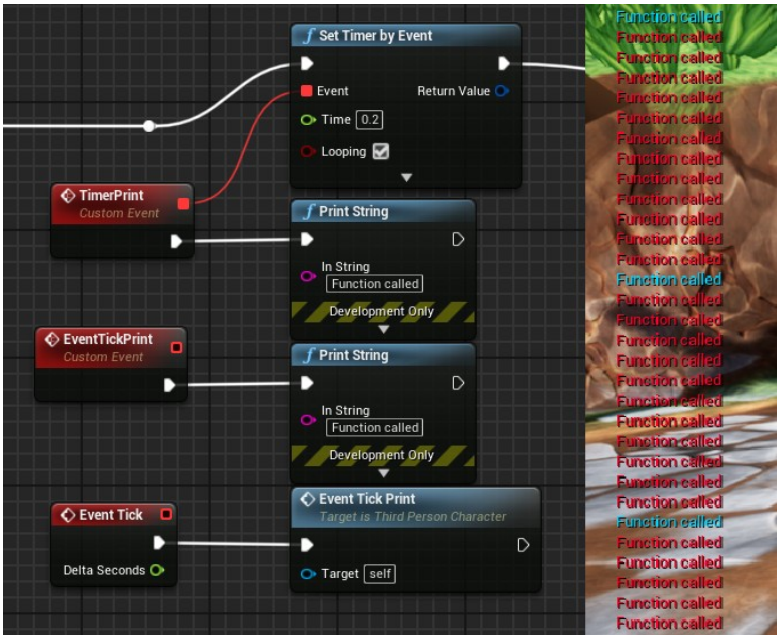


Figura 7.123: Timer vs EventTick

7.10. Proves

Durant el desenvolupament s'han estat fent proves de funcionament de les mecàniques de forma continua. Les proves s'han realitzat en diferents escenaris, començant per una petita sala buida on testejar les primeres versions de les mecàniques, passant per terrenys petits on hi havia més llibertat de moviments per a veure com funcionava la integració de diferents mecàniques i finalitzant en el terreny que s'ha utilitzat com a escenari final.

Les proves sobre la base en C++ han suposat una dificultat afegida a les proves de la part de Blueprints, ja que qualsevol accés incorrecte a memòria feia que l'editor del motor deixés de funcionar, i moltes vegades no quedava clar quin era l'error que havia desencadenat el problema. Si a això s'afegeix que Unreal Engine 4 tendeix a penjar-se sol de tant en tant, cada cop que es feien canvis al codi i el motor es congelava s'havia de tornar a provar d'executar el joc per saber si l'error era del codi o havia estat un error del motor.

Unreal Engine 4 implementa un conjunt de funcions de Debug que permeten des de mostrar missatges per pantalla fins a dibuixar línies i punts a l'escenari. S'han utilitzat funcions de Blueprints com *`PrintString()`*, *`DrawDebugLine()`* o *`DrawDebugPoint()`* per a mostrar informació dintre del joc. En C++ s'ha utilitzat la funció *`UE_LOG()`* que envia a la consola i als fitxers de log els missatges definits.

8. Feedback extern

Quan es desenvolupa un videojoc és important conèixer l'opinió de jugadors que no hagin format part del procés de creació, ja que els membres de l'equip de treball tenen molt clar el funcionament del joc i poden passar per alt detalls que afectin negativament en la jugabilitat del títol.

Aquest projecte ha estat molt centrat en la implementació d'una base de mecàniques sòlida a sobre de la qual fos fàcil afegir contingut nou, i per aquest motiu s'ha decidit llençar una petita demo per rebre feedback sobre el funcionament d'aquestes.

La demo s'ha distribuït de forma pública a través de la plataforma especialitzada en títols independents itch.io [7], i a més s'ha enviat a persones de confiança dels desenvolupadors. S'han escollit persones de diferents perfils per poder tenir opinions tant de jugadors habituals, que ja estan acostumats als controls bàsics dels jocs de supervivència, com de persones que no han jugat mai a cap joc. Amb aquests perfils s'ha buscat poder rebre feedback de la part de funcionament de les mecàniques per part dels jugadors experimentats, i de la dificultat per jugar per part de persones que no tenen coneixements previs sobre el funcionament d'altres jocs.

Al completar la demo els jugadors podien omplir un formulari [8] per valorar els següents aspectes del joc:

- Progrés del joc: Els jugadors valoren la progressió al llarg de la demo, concretament si han tingut algun problema que no els hagi deixat complir algun dels objectius.
- Controls: Els jugadors valoren com d'intuïtius són els controls del joc.
- Moviment: Els jugadors valoren com es sent el moviment del personatge a l'hora de caminar, saltar o córrer.
- Report de bugs: Si s'ha trobat algun bug en aquest punt es pot aportar informació sobre aquest.
- Valoració personal: Els jugadors poden donar una opinió personal de la demo.
- Compartició del log de la partida: Els jugadors poden compartir un arxiu que es genera al completar la demo amb informació de la partida, útil per als desenvolupadors.

La demo es va llençar el dia 25/04/2021, i fins al moment de l'elaboració d'aquest document ha estat descarregada 14 cops. A la Figura 8.1 es mostra l'evolució de les visites i les descarregues des del dia de llançament fins al 03/06/2021.

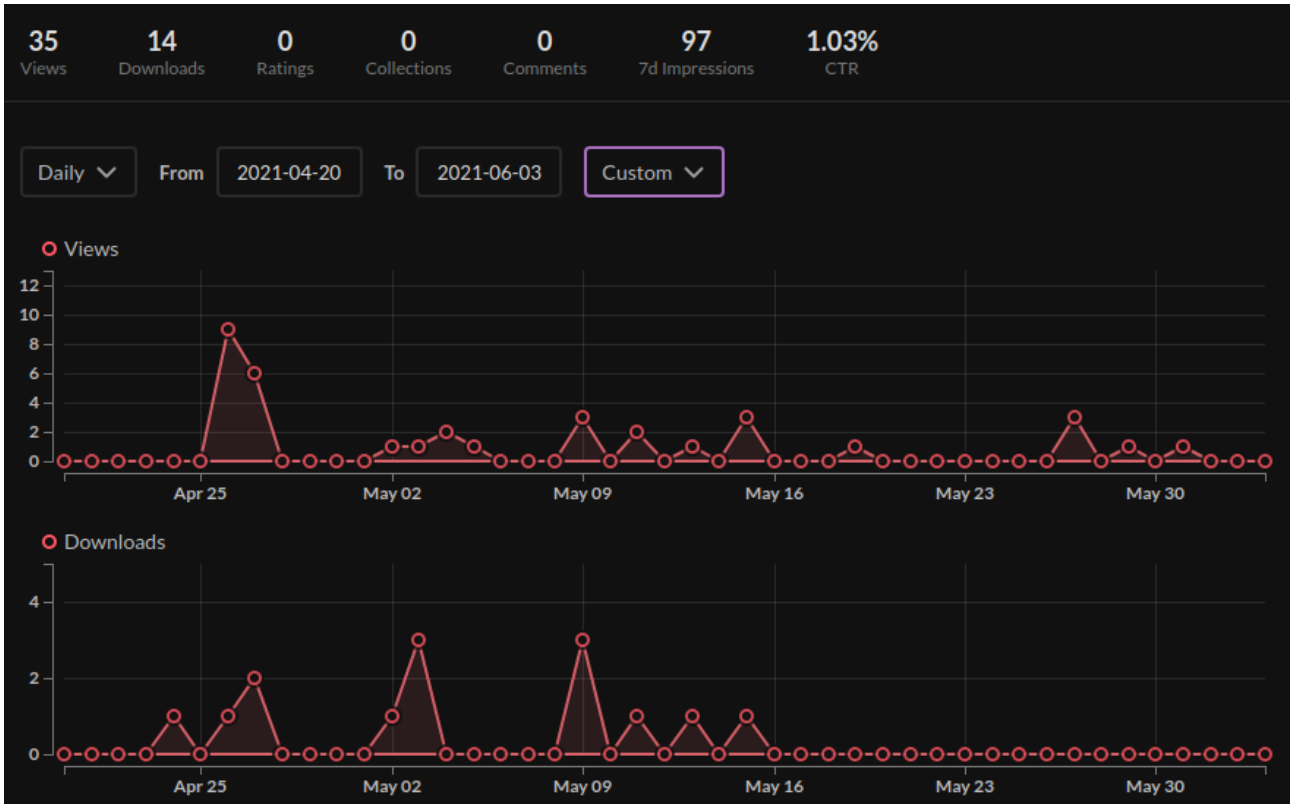


Figura 8.1: Evolució de les visites i descarregues de la demo

Aquestes 14 descarregues han generat 8 respostes al formulari, donant un percentatge de respostes del 57%.

8.1. Valoració de les respostes

Bloc 1: Progrés

A la Figura 8.2 es pot veure com la majoria dels jugadors que han provat el joc han tingut algun problema per completar la demo. Els problemes més repetits han estat:

- Al prémer ESC amb una interfície oberta aquesta es quedava a la pantalla la resta de la partida
- Problemes per aconseguir la Fibra Vegetal
- Dificultat per agafar ítems en 3ª persona
- Problemes per entendre el menú de construcció i la col·locació dels objectes

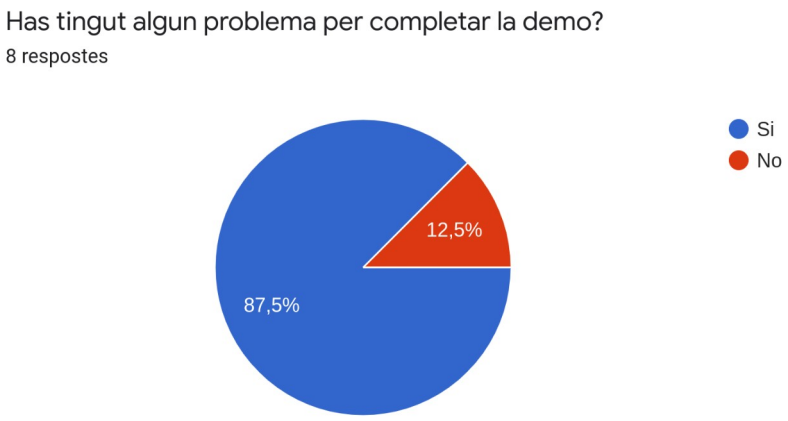


Figura 8.2: Respostes sobre el progrés del joc

Bloc 2: Controls

A la Figura 8.3 es pot veure com 2/3 dels jugadors han fet una valoració positiva dels controls, trobant-los intuïtius o molt intuïtius, i el 1/3 restant els ha trobat poc intuïtius. Les valoracions negatives sobre els controls han estat:

- Costa recordar-se de totes les tecles
- Costa entendre les opcions del martell

També cal destacar que hi ha hagut diferents comentaris positius, tant d'usuaris experimentats com d'usuaris no habituats a jugar.

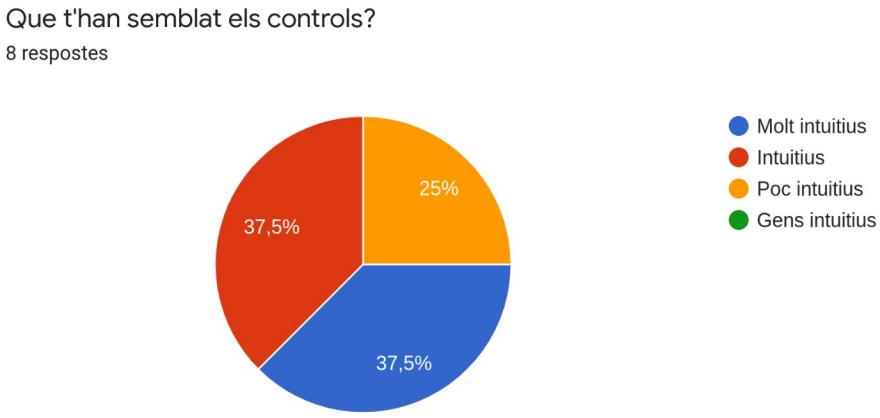
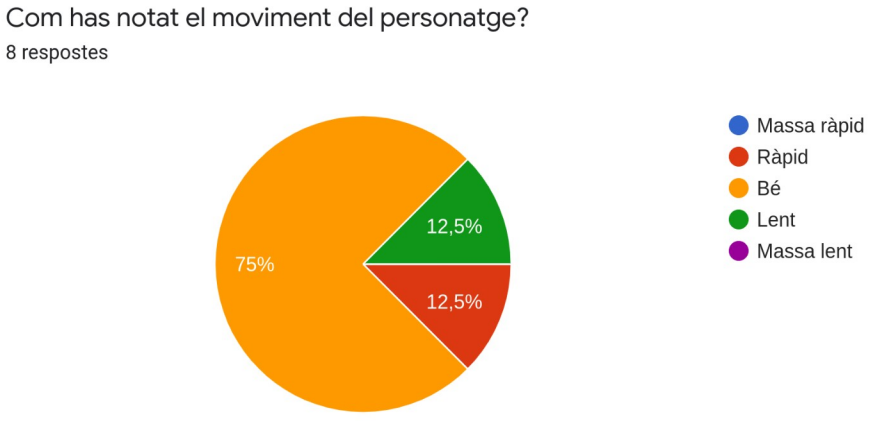


Figura 8.3: Respostes sobre els controls del joc

Bloc 3: Moviment

A la Figura 8.4 es pot veure com 3/4 dels jugadors consideren que el moviment del jugador està bé, i el 1/4 restant es divideix entre ràpid i lent. Els jugadors que han indicat que els controls estan bé remarquen que hi ha un bon balanç entre el moviment i el temps que es necessita per a realitzar les accions.



Bloc 4: Report de bugs

El bug que més s'ha repetit ha estat que al prémer ESC amb alguna interfície oberta, aquesta es quedava a la pantalla fins al final de la partida. Diferents persones han detectat aquest bug en vàries interfícies.

Un altre bug que ha aparegut un parell de vegades ha estat que, tot i haver realitzat totes les tasques d'un objectiu, aquest no es marcava com a completat.

Bloc 5: Valoració personal

Totes les valoracions rebudes han estat positives i s'han pogut extreure conclusions interessants. Un parell de persones han considerat que el joc és poc rejugable, i han demanat una versió multijugador o més objectius, també s'han fet consideracions referents al nombre d'objectes de l'escenari que no s'utilitzen. Finalment, han hagut alguns usuaris que han considerat que les fibres i les pedres eren difícils de veure entre la herba de l'escenari.

8.2. Millores a partir del feedback

A partir del feedback rebut s'han realitzat algunes modificacions per tal de millorar els punts més dèbils del projecte.

Millores visuals i de gameplay:

- S'ha incrementat la mida de les pedres perquè ressaltin més entre l'herba de l'escenari.
- S'ha modificat la hitbox de les fibres vegetals perquè siguin més fàcils d'agafar.

Correcció de bugs:

- S'ha corregit el bug que bloquejava les interfícies a la pantalla si es premia ESC.

9. Resultats

9.1. Legislació i normativa vigent

El joc desenvolupat no implementa cap sistema que es vegi afectat per la Llei Orgànica de Protecció de Dades, LOPD, o per la Llei de Serveis a la Societat de la Informació i Comerç Electrònic, LSSICE, ja que ni emmagatzema dades de caràcter personal ni inclou cap sistema que comporti una activitat econòmica, com pagaments o micropagaments.

Tots els assets de tercers incorporats al projecte han estat adquirits prèviament i utilitzats sota la seva llicència d'ús, de forma que el projecte no pot tenir problemes amb els drets d'autor.

PEGI [9], o Pan European Game Information, és un sistema europeu que classifica els videojocs entre diferents categories, i assigna una edat mínima recomanada. Aquest sistema està pensat per a oferir una orientació als compradors dels videojocs, per ajudar-los a saber de forma ràpida i visual les característiques d'aquest. La Figura 9.1 mostra totes les icones del sistema PEGI i el seu significat.

Figura 9.1: Icones PEGI

El projecte implementat s'hauria d'etiquetat com a PEGI 3, que vol dir que és apte per a tots els públics, ja que no conté sons ni imatges que pugin espantar als nens ni conté cap tipus de violència.

9.2. Joc final

A continuació es mostrara el resultat final del projecte a partir d'una serie de captures de pantalla de les parts més importants. A més a més, en el següent enllaç es pot veure el funcionament de les característiques principals del joc en un breu vídeo resum: https://www.youtube.com/watch?v=speFL97wMws

Figura 9.2: Menú inicial

Figura 9.3: Escenari i jugador

Figura 9.4: Interfícies integrades a la pantalla

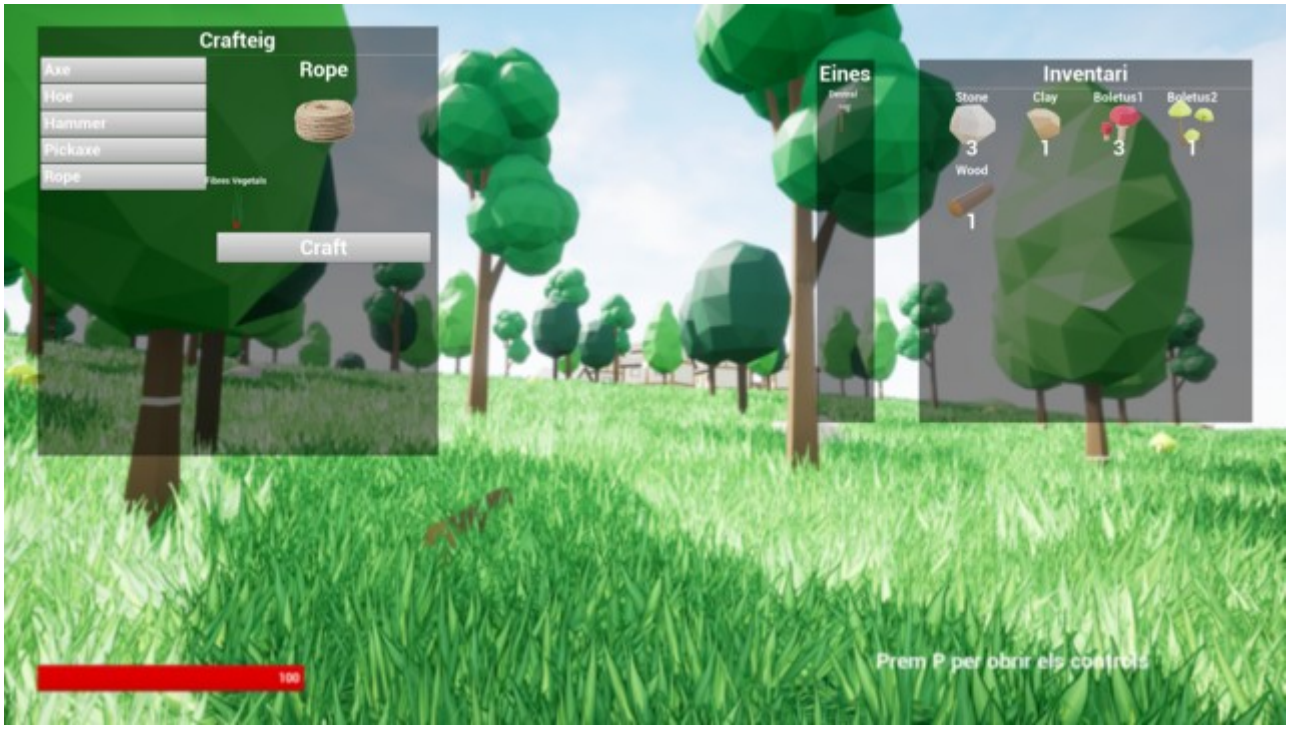


Figura 9.5: Inventari i sistema de crafteig

Figura 9.6: Sistema d'objectius

Figura 9.7: Zones cultivables i mecànica de plantar

Figura 9.8: Previsualització de construccions

Figura 9.9: Col·locació de construccions

Figura 9.10: Interfícies de construcció

Figura 9.11: Diferents edificis

Figura 9.12: Pantalla final

10. Conclusions

Abans de començar a pensar idees per a realitzar el TFG, sempre s'havia volgut fer un projecte que tingues una motivació extra a sobre de la implementació d'un videojoc. Al ser un projecte tant important es volia donar més profunditat no tan sols dissenyar un joc perquè agradin les mecàniques o l'estètica. En aquest punt va sorgir la idea de vincular el projecte a la terra de l'equip de desenvolupament, i a més a més ambientant-lo en una època propera però molt poc coneguda, com és el final del segle XIX i el començament del segle XX.

A partir d'aquesta idea s'havia de plantejar com enfocar el treball, si centrar-se en la part d'història i com transmetre aquest estil de vida als jugadors, o donar èmfasi a la representació de l'escenari i dels objectes típics de la regió, o si enfocar-se més en la part de desenvolupament d'un videojoc. Com que l'equip de desenvolupament és d'un perfil molt més tècnic que artístic es va decidir treballar en la implementació d'una aplicació sòlida però molt modular per establir una base a partir de la qual poder transmetre tant la història com la representació de la vida a l'època.

Inicialment es va decidir treballar amb Unreal Engine 4 utilitzant el sistema de Blueprints, ja que era el què s'havia vist durant el grau, però a mesura que es descobria el funcionament intern del motor, i la metodologia seguida per equips grans a l'hora de fer jocs es va decidir virar cap a una implementació hibrida entre C++ i Blueprints. Aquesta decisió ha permès aprendre moltíssimes coses noves tant sobre el funcionament concret de l'eina com sobre programació en general i molt especialment programació orientada a objectes, ja que des de les primeres fases del projecte s'ha intentat cuidar el disseny des del punt de vista de l'enginyeria del software, planificant amb cura les classes i les relacions d'herència entre elles, buscant la màxima eficiència en les estructures de dades utilitzades, i establint una base de Blueprints per a modificar els components visuals.

En els projectes realitzats durant el grau no s'havia tingut temps de treballar de forma avançada amb aspectes com les animacions, els sons o els shaders complexos, així que es va considerar que el projecte de final de grau era un bon moment per a anar més enllà en el funcionament d'aquests aspectes molt importants en la implementació de videojocs. La feina feta en aquesta part ha estat molt satisfactòria, ja que ha permès veure i aprendre com funcionen tècniques com *animation blending, màquines d'estats, distance tiling, macro texture variation, culling,* entre moltes altres.

Tots aquests punts mencionats havien de culminar en una demo tècnica escalable, a partir de la qual poder seguir el desenvolupament del projecte ja més enfocat en la història i l'estètica. La demo implementada compleix tots els objectius inicials, tot i que en diferent mesura. Si dividim els objectius en Tècnics, Estètics i Narratius s'extreuen les següents conclusions:

- Objectius tècnics: Des del punt de vista de la implementació, la demo compleix tots els objectius plantejats, ja que funciona a sobre d'una base de codi molt sòlida en C++ i que permet afegir contingut a partir de dos clics. L'estructura en C++ està coberta a més alt nivell per una sèrie de Blueprints a partir de les quals es pot implementar automàticament qualsevol tipus d'ítem, i l'únic que cal fer és modificar els paràmetres de disseny exposats, com el model, el nom o la descripció i altres paràmetre concrets dels diferents tipus d'ítems.
- Objectius estètics: Aquesta part era on menys experiència es tenia a l'hora de fer el disseny. Adquirir els assets ha permès crear ràpidament escenaris de forma coherent sense haver de treure temps a la part de disseny i implementació del software. En tot moment s'ha intentat cuidar l'estètica dels elements implementats, per exemple a l'hora de representar el material del terreny, i el resultat final es considera que ha complert tots els objectius.
- Objectius narratius: La part d'integrar la recerca sobre l'estil de vida i el context a dintre del joc, tot i que compleix els objectius plantejats d'explicar el funcionament d'algunes tècniques i eines, es considera que hagués pogut tenir un nivell més elevat de profunditat, però per motius de temps s'ha preferit polir la part d'implementació abans que afegir moltes tècniques i recursos. Tot i que la demo no incorpori moltes d'aquestes tècniques, la base construïda permetria afegir-les de forma molt ràpida.

La valoració global del projecte és molt positiva, ja que s'ha aprés molt tant de temes que ja es coneixien com de temes totalment nous. Al llarg del projecte s'ha passat per moltes de les fases de disseny d'un videojoc, i la decisió de llençar una demo pública per a rebre feedback ha permès experimentar, a petita escala, la sensació de treure una aplicació al mercat. Veure la forma de fer les coses d'altres persones diferents a l'equip de desenvolupament ha permès trobar molts errors que no s'havien testejat durant el desenvolupament, i alguns inclús van requerir del llançament d'una segona versió de la demo per a solucionar-los.

11. Treball futur

El joc implementat estableix una base de mecàniques i funcionalitats sòlida, permetent afegir contingut de forma ràpida, fent que sigui molt fàcil afegir nou contingut com ítems, eines o missions. Tot i això es podrien afegir algunes funcionalitats o mecàniques extra com:

- **Més ítems:** Incrementar el nombre d'ítems de tots els tipus. Afegir nous recursos, més menjar, edificis, receptes...
- **Més informació sobre l'estil de vida de l'època:** A sobre de la demo es poden afegir més elements per transmetre més informació sobre l'època.
- **Implementar una versió seriosa del joc:** Crear un nou mode que estigui més orientat en la part de l'estil de vida i de les tècniques que en la part de supervivència, per a poder-lo utilitzar com a eina pedagògica a les escoles.
- **Representació de més tècniques tradicionals:** A la demo implementada s'han explicat les tècniques més bàsiques de recol·lecció de recursos, però s'ha definit el sistema per a crear noves tècniques a partir d'ítems i zones de producció.
- **Intel·ligència Artificial:** Afegir NPCs (Non Playable Characters) per diferents zones del mapa que puguin interactuar amb el jugador. Aquests personatges podrien donar noves missions o permetre comerciar amb el jugador.
- **Ramaderia:** Permetre que el jugador pugui criar animals per a obtenir recursos derivats com llet o carn.
- **Més paràmetres de supervivència:** Implementar components per a representar la gana o el fred del jugador per a donar més profunditat a la part de supervivència.
- **Cicle dia/nit**: Canviar el sistema d'il·luminació estàtic per un de dinàmic que representi el dia i la nit. Aquest canvi hauria d'anar acompanyat d'una sèrie de modificacions en el gameplay per a diferenciar els dos cicles, com per exemple, incrementar el fred que té el personatge si és de nit.
- **Estacions:** Implementar un sistema que representi les estacions de l'any. Les estacions durarien un nombre determinat de dies del joc, i afectarien en aspectes com el fred, els recursos que es poden obtenir o els cultius que es poden plantar.

- **Models personalitzats:** Deixar de dependre d'una llibreria de models genèrics per a crear els objectes tradicionals, ja que al ser tant concrets s'ha hagut de reinterpretar-los a partir d'objectes genèrics.
- **Multijugador cooperatiu:** Permetre que diferents jugadors pugin sobreviure en una mateixa partida, compartint recursos.
- **Migrar el projecte a Unreal Engine 5:** L'estiu passat es va anunciar la nova versió d'Unreal Engine, i recentment s'ha llençat una primera versió en accés anticipat perquè els desenvolupadors puguin començar a treballar amb les noves característiques. Treballar amb UE5 permetria utilitzar les noves tecnologies *Nanite* i *Lumen*. *Nanite* permet tenir a l'escenari milions de triangles sense haver de gestionar els nivells de detall dels ítems, ja que es fa des del propi motor. Això permet als artistes treballar directament amb les versions *HighPoly* dels models, ja que al importar-los el propi motor serà l'encarregat d'escollir com gestionar la geometria. *Lumen* és un sistema que permet utilitzar il·luminació global sense necessitat de precalcular els rebots de la llum, ja que està optimitzat per a treballar en temps real. Aquest sistema ofereix il·luminacions molt realistes de forma totalment dinàmica.

12. Referències

[1] EpicGames. (2021). *Hardware and Software Specifications.* [Consulta: 11/06/2021] https://docs.unrealengine.com/4.26/en-US/Basics/RecommendedSpecifications/

[2] Unity Technologies. (2021). *Low Poly Ultimate Pack.* [Consulta: 11/06/2021] https://assetstore.unity.com/packages/3d/props/low-poly-ultimate-pack-54733

[3] EpicGames. (2021). *Another Stylized Material Collection 8.* [Consulta: 11/06/2021] https://www.unrealengine.com/marketplace/en-US/product/another-stylized-material-collection-07?lang=en-US&sessionInvalidated=true

[4] GameDevTv. (2021). *Gamemaster Audio ProSound Mini Pack.* [Consulta: 11/06/2021] https://www.gamedev.tv/p/gamemaster-prosound-mini

[5] EpicGames. (2021). *Unreal Architecture.* [Consulta: 11/06/2021] https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/

[6] Adobe. (2021). *Mixamo.* [Consulta: 11/06/2021] https://www.mixamo.com/#/

[7] ItchCorp. (2021). *Itch.io.* [Consulta: 11/06/2021] https://itch.io/

[8] Google. (2021). *Forms.* [Consulta: 11/06/2021] https://forms.gle/Tbi6BceRCrzzf6169

[9] PEGI. (2021). *Pan European Game Information.* [Consulta: 11/06/2021] https://pegi.info/es

13. Bibliografia

1. Doran, J. Sherif, W. Whittle, S.(2019). *Unreal Engine 4.X Scripting With C++ Cookbook: develop quality game components and solve scripting problems with the power of C++ and UE4*. PACKT Publishing Limited.
2. Epic Games. (2021). *Unreal Engine Answerhub.* [Consulta: 11/06/2021]. https://answers.unrealengine.com/index.html
3. Epic Games. (2021). *Unreal Engine Documentation*. [Consulta: 11/06/2021]. https://docs.unrealengine.com/
4. Epic Games. *Unreal Engine Forums.* [Consulta: 11/06/2021]. https://forums.unrealengine.com/
5. Epic Games. *Youtube UE4.* [Consulta: 11/06/2021]. https://www.youtube.com/user/UnrealDevelopmentKit

14. Annexos

14.1. Codi font del joc

El codi font del joc pesa gairebé 30GB, per això s'ha pujat comprimit al Google Drive. Per accedir-hi s'ha de descarregar el ZIP des de l'enllaç que es troba al fitxer «codiFont.txt», a l'arrel de la carpeta del projecte.

14.2. Respostes completes del formulari

A la carpeta del projecte s'ha inclòs l'arxiu «DemoTFG.csv» que conté en format CSV totes les preguntes del formulari i les vuit respostes completes.

15. Manual d’usuari i d’instal·lació

A la carpeta del projecte s’ha incorporat un enllaç per a descarregar l’executable per a Windows del joc. Per a jugar s’ha de descarregar des del Google Drive l’arxiu «Executable.zip» i descomprimir la carpeta «WindowsNoEditor». Dintre d’aquesta carpeta es troba l’arxiu «Projecte_TFG.exe», que inicia el joc. Si l’ordinador on s’està executant té instal·lats els Visual C++ Redistributable Packages for Visual Studio el joc s’iniciarà automàticament. Si aquests paquets no estan instal·lats el propi executable oferirà la opció d’instal·lar-los automàticament.

Un cop iniciat el joc els controls són els següents:

- Personatge:
  - Moviment: W A S D
  - Moviment de la càmera: Ratolí
  - Saltar: Barra espaiadora
  - Esprintar: SHIFT
  - Interacutar: E
  - Obrir Inventari: TAB
  - Obrir missions: Y
- Eines:
  - Utilitzar eina: Click esquerra
  - Seleccionar destral: 1
  - Seleccionar aixada: 2
  - Seleccionar pic: 3
  - Seleccionar martell: 4
  - Desseleccionar eina: 1, 2, 3 o 4
- Construir/plantar:
  - Seleccionar: Click dret

- Construir/plantar: Click esquerra

La Figura 14.1 mostra la distribució dels controls en un teclat.

Figura 15.1: Controls sobre un teclat