TecnoCampus

Centres universitaris adscrits a la

Universitat Pompeu Fabra Barcelona

Grau en Disseny i Producció de Videojocs

Implementació d'un Editor de *Behaviour Trees* a *Unity Engine*

MEMÒRIA FINAL

Ricard Perea Ros

Tutor: Dr. Enric Sesa i Nogueras

2018-2019

TecnoCampus Mataró-Maresme

Abstract

This project is focused on the implementation of a behaviour tree graphic editor for Unity Engine. This thesis shows the state of the art of this decision-making system and establishes the desired behavior of the developed tool.

Resum

Aquest treball es centra en la implementació d'un editor gràfic d'arbres de comportament per al programa *Unity Engine.* Al llarg de la tesis es mostra l'estat de l'art d'aquest sistema de presa de decisions i s'estableix quin es el comportament desitjat a l'eina desenvolupada.

Resumen

Este trabajo se centra en la implementación de un editor gráfico de árboles de comportamiento para el programa *Unity Engine.* A lo largo de esta tesis se muestra el estado del arte de este sistema de toma de decisiones y se establece que comportamiento se desea en la herramienta desarrollada.

Índex

Índex de Figures IX

Índex de Taules XI

Glossari de Termes XIII

1. Introducció 1

2. Anàlisis de Referents 3

2.1. Obres que Implementen BTs 3

2.1.1. Halo 2 (Bungie Studios, 2004) 3

2.1.2. Farcry 3 (Ubisoft, 2012) i Farcry 4 (Ubisoft, 2014) 5

2.1.3. Altres 6

2.2. Editors de BTs 6

2.2.1. Unreal Engine 4.21 (*Epic Games,* 2018) 6

2.2.2. Behaviour Bricks (*Padaone Games,* 2018) 7

3. Marc Teòric 11

3.1. Producció de Videojocs 11

3.1.1. Rols Dintre la Producció 11

3.1.2. El Desenvolupament com un Procés Incremental 12

3.2. Intel·ligència Artificial 12

3.2.1. IA als Videojocs 13

3.2.2. Presa de Decisions 14

3.2.2.1. Sistemes Deliberatius i Sistemes Reactius 14

3.2.2.2. Finite State Machines 15

3.2.2.3. Hierarchycal Finite State Machines 16

3.2.2.4. Decision Trees 17

3.2.2.5. Behaviour Trees 17

3.2.3. Blackboards 18

3.3. Behaviour Trees en Profunditat 19

3.3.1. BTs als Videojocs 19

3.3.2. Concepció Clàssica 20

3.3.2.1. Funcionament 20

3.3.2.2. Tipus de nodes 20

3.3.2.3. Nodes amb Memòria 22

3.3.2.4. BTs No Reactius 23

3.3.3. Noves Fronteres 23

3.3.3.1. Utility BTs 23

3.3.3.2. Stochastic BTs 23

3.3.3.3. Hinted Execution BTs 24

3.3.3.4. Dynamic Expansion of BTs 24

3.4. Eines 24

3.4.1. Unity Engine 24

3.4.1.1. Motors de Jocs 24

3.4.1.2. Unity com a Motor de Jocs 26

3.4.1.3. Unity com a Editor 26

3.4.2. C# 27

3.4.2.1. Programació Orientada a Objectes 28

3.4.2.2. Programació Genèrica 28

3.4.2.3. Reflexió 28

4. Objectius 29

4.1. Objectius Principals 29

4.2. Objectius Secundaris 29

5. Disseny Metodològic i Cronograma 31

5.1. Estudi Previ al Desenvolupament 31

5.2. Requisits del Projecte 33

5.3. Metodologia 34

5.4. Planificació 35

6. Desenvolupament del Treball 39

6.1. Primera Iteració 39

6.1.1. Objectius de la Iteració 39

6.1.2. Desenvolupament de la Iteració 40

6.1.2.1. Implementació de la Llibreria 40

6.1.2.2. Implementació d’un BT de Test 43

6.1.3. Valoració de la Iteració 46

6.2. Segona Iteració ... 46

6.2.1. Objectius de la Iteració ... 46

6.2.2. Desenvolupament de la Iteració ... 47

6.2.2.1. Implementació del BTEngine.Core ... 47

6.2.2.2. Implementació del BTEngine.GraphicEditor ... 50

6.2.3. Valoració de la Iteració ... 53

7. Resultats del Treball ... 55

7.1. Resultats Objectius ... 55

7.2. Valoracions Finals ... 56

8. Referències ... 59

Annex ... 63

1.1. Diagrames UML ... 65

1.2. Eina Implementada ... 65

1.3. Codi Font ... 65

1.4. Instruccions d'Ús de l'Eina ... 65

Índex de Figures

Figura 2.1. Halo 2 Exemple de DAG 4

Figura 2.2. Halo 2 Comportaments Permesos Segons el Context 5

Figura 2.3. Unreal Engine BT 7

Figura 2.4. Behaviour Bricks BT 8

Figura 3.1. Representació d’un Desenvolupament Incremental 12

Figura 3.2. Model de la IA d’un videojoc 13

Figura 3.3. Esquema de la presa de decisions 14

Figura 3.4. Exemple d’una FSM 16

Figura 3.5. Exemple d’una HFSM 16

Figura 3.6. Exemple d’un Decision Tree 17

Figura 3.7. Exemple d’un arbre de comportament 18

Figura 3.8. Arquitectura de Pissarra Canònica 19

Figura 3.9. Representació gràfica d’un Selector 21

Figura 3.10. Representació gràfica d’una Seqüencia 21

Figura 3.11. Representació gràfica d’un Paral·lel 21

Figura 3.12. Representació gràfica d’Acció, Condició, i Decorador 22

Figura 3.13. Comparativa de la implementació de un BT amb memòria i sense 23

Figura 3.14. Blocs Funcionals d’un Motor de Jocs 25

Figura 3.15. Exemple d’Inspector Personalitzat 27

Figura 3.16. Exemple de Finestra Personalitzada 27

Figura 5.1. Estructura de Branques ... 35

Figura 5.2. Calendari de Producció ... 35

Figura 5.3. Llegenda Cronograma ... 36

Figura 5.4. Cronograma del Desenvolupament ... 36

Figura 6.1. UML-S Estructura de la Llibreria ... 40

Figura 6.2. UML IBlackboard ... 41

Figura 6.3. UMLS Sistema de Tasques Primera Herència ... 41

Figura 6.4. UMLS Nodes de Control de Flux ... 42

Figura 6.5. UMLS LeafTask ... 42

Figura 6.6. UMLS Llibreria de BTs ... 43

Figura 6.7. UML BehaviouTree ... 43

Figura 6.8. UML Blackboard ... 44

Figura 6.9. BT de Test ... 45

Figura 6.10. BT Buy Groceries ... 45

Figura 6.11. BTEngine.Core UMLS ... 48

Figura 6.12. Task UML ... 49

Figura 6.13. UML SerializedTask ... 50

Figura 6.14. UMLS BTEngine.GraphicEditor ... 51

Figura 6.15. UMLS Sistema de Nodes Gràfics ... 51

Figura 6.16. BTEngine ... 53

Índex de Taules

Taula 3.1. Tipus de Node d'un BT 22

Taula 5.1. Funcionalitats Comunes a Ambdues Eines Referents 31

Taula 5.2. Funcionalitats dels BTs d'Unreal Engine 32

Taula 5.3. Funcionalitats de Behaviour Bricks 32

Taula 5.4. Funcionalitats Desitjades 33

Taula 5.5. Dates destacades 37

Taula 7.1. Resultats Finals 55

Glossari de Termes

Agent	Entitat dotada d'intel·ligència artificial.
API	Interfície de programació d'aplicacions.
Asset	Qualsevol tipus d'arxiu que s'incorpori en un joc.
BT	Arbre de comportament (*behaviour tree* en anglès).
Dithering	Canvi d'estat d'una IA en breus períodes de temps sense executar cap comportament complert de forma apreciable.
FPS	Joc de dispars en primera persona (*first Person Shooter* en anglès).
FSM	Màquina d'estats finits (*finite state machine* en anglès).
HFSM	Màquina d'estats finits jeràrquica (*hierarchycal finite state machine* en anglès).
IA	Intel·ligència artificial.
Serialitzar	Guardar les dades de forma persistent.
UML	Unified Modeling Language.
UMLS	Unified Modeling Language Simplified.

1. Introducció

La intel·ligència artificial aplicada als videojocs és un tema extens que aglutina moltes tècniques i, encara més, formes d'implementar-les (Rabin, 2002). Aquestes tècniques es poden dividir en diferents grups, un d'ells és el de presa de decisions. Dintre d'aquest conjunt el sistema més utilitzat en del món dels videojocs són les màquines d'estats finits (Millington i Funge, 2009). Les FSM però presenten grans inconvenients pel que fa a la seva escalabilitat, manteniment, i per la relació entre la seva modularitat i reactivitat. Els arbres de comportament neixen dintre de la indústria del videojoc com un sistema que evita aquests problemes. Ja què permeten una alta encapsulació del codi i desvinculen la presa de la decisió de la tasca a executar (Colledanchise i Ögren, 2018).

L'objectiu principal d'aquest treball es implementar una eina que permeti la creació i edició d'arbres de comportament a l'editor de *Unity Engine.* Aquesta ha de disposar d'una interfície gràfica per realitzar les accions abans esmentades.

A continuació s'analitzen els referents que s'han utilitzat per a desenvolupar l'objecte d'aquest document. A grans trets existeixen dos grups, obres en les que la implementació de BTs ha estat cabdal, i eines que permeten implementar i editar BTs de forma gràfica. Tot seguit en el marc teòric es contextualitza on s'emmarquen els arbres de comportament dintre de la intel·ligència artificial, i s'explica quin és l'estat de l'art dels mateixos. La secció 4 exposa l'abast dels objectius del treball i la secció 5 explica la metodologia emprada per assolir-los. A continuació, l'apartat 6 explica la solució implementada. Finalment, a la secció 7 es realitza l'anàlisi dels resultats així com una valoració final del treball.

2. Anàlisis de Referents

Aquesta secció mostra algunes de les obres que es consideren referents per la seva relació amb els BTs. L'apartat es divideix en dues subseccions. La primera parla sobre videojocs on la implementació d'aquest tipus de sistema ha estat important en el seu desenvolupament. La segona es centra en analitzar dos editors d'arbres de comportament de diferents característiques.

Cal esmentar, que si bé aquest treball no en parla, els BTs també s'utilitzen en diversos camps de la robòtica. Tot i que aquest document no recull cap referent d'aquest tipus, existeix un ampli camp d'estudi en aquest sector (Colledanchise i Ögren, 2018).

2.1. Obres que Implementen BTs

Els arbres de comportament, tal com Colledanchise i Ögren (2018) expliquen, neixen dintre de la indústria del videojoc. Degut a que aquest tipus de sistema esta força arrelat a la indústria existeixen molts exemples de jocs que els usen, especialment dintre de la producció triple A (Thompson, 2019).

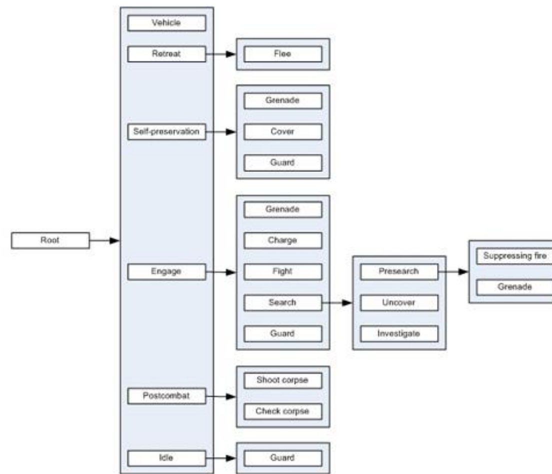
Aquest apartat recull diversos jocs amb característiques que fan que la seva implementació de la IA fos especialment interesant i que es van resoldre amb BTs. El primer cas, *Halo 2,* va asentar les bases dels BTs moderns i va extendre al seu ús a la indústria. El següent cas, si bé no és tan rellevant, parla de la IA sistèmica emprada a la saga *Farcry.* Dintre del mateix marc, s'explica com es modifiquen els sistemes dels agents per adequar-los a les IAs de companyia que apareixen a la cuarta entrega de la mateixa saga.

2.1.1. Halo 2 (Bungie Studios, 2004)

Halo 2 és un FPS ambientat en un món de ciència-ficció en el què el jugador s'enfronta a diversos tipus d'alienígena. Entre d'altres, els reptes per desenvolupar la IA dels *Covenant* (els alienígenes enemics) eren la capacitat de canviar l'arma, els vehicles, i gestionar els comportaments en un món immens (Thompson, 2017). El valor referencial d'aquesta obra radica en que és el primer videojoc que implementa i estableix les bases dels BTs a la indústria.

Isla (2005) recull que *Bungie Studios* estructura la IA de *Halo 2* utilitzant un graf acíclic dirigit per tal de fer usable la implementació de la mateixa als dissenyadors. Si bé no empra explícitament el terme *behaviour tree* per definir el sistema, Thompson (2017) el defineix com a tal en realitzar un estudi de l'obra.

Figura 2.1. Halo 2 Exemple de DAG



Font: (Isla, 2005)

Isla (2005) explica que els comportaments dels agents responen, entre d'altres, als principis següents:

- Coherència: "Si un comportament es defineix com una acció prolongada en el temps, ens hem d'assegurar que els agents comencen, aturen i canvien d'acció al moment adequat." (Isla, 2005).
- Transparència: El comportament ha de permetre que algú no entrenat realitzi suposicions raonables de com reaccionarà l'agent a certs estímuls. En altres paraules, els agents han de comportar-se de forma comprensible per al jugador.

Isla (2005) exposa que per tal d'assolir comportaments que complissin aquests estàndards *Bungie Studios* va adoptar una estratègia de "aproximar-se al sentit comú a base de força bruta". Matisant que s'entén força bruta en el sentit què "com més repertori de comportaments tingui un agent més fàcil és reconèixer-lo com una entitat amb sentit comú" (Isla, 2005).

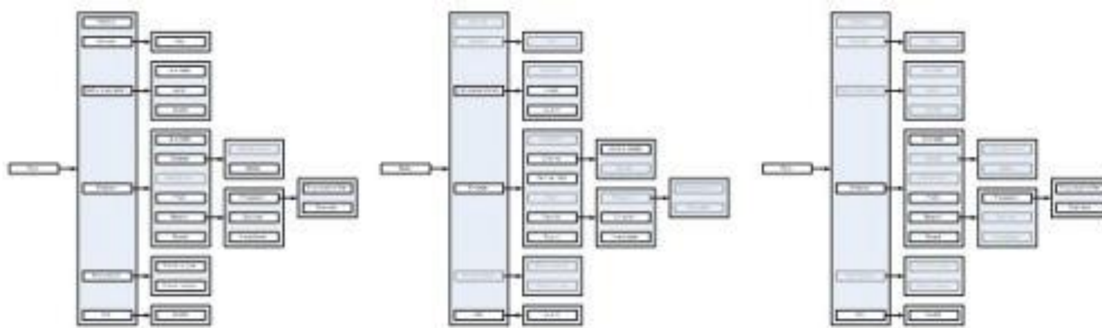
Isla (2005) destaca tres fets que van fer possible la implementació de la IA del *Halo 2*. La primera és el concepte d'utilitat de les tasques, el segon és l'execució contextual, i finalment tenim els comportaments basats en estímuls.

Tot i que la utilitat aplicada als BTs s'explica més detalladament a l'apartat 3.3.3.1, *grosso modo* consisteix en alterar l'ordre d'elecció de les tasques segons la utilitat que tenen en un moment donat per a l'agent. Isla (2005) recull que aquest fet va ser cabdal per simplificar el principi de coherència ja que simplificava l'extinció del *dithering*.

Thompson (2017) destaca en el seu anàlisi la execució contextual com la decisió que va permetre que el sistema de *Bungie Studios* s'executés dintre de uns límits de temps que fessin els agents poguessin prendre decisions en temps real. Els comportaments tan sols son

executables quan certes condicions es donen. Per tant, no cal comprovar la utilitat de totes les tasques de l'arbre, sinó tan sols les d'aquells comportaments que són coherents executar per al context actual de l'agent (Thompson, 2017).

Figura 2.2. Halo 2 Comportaments Permesos Segons el Context



Font: (Isla, 2005)

Finalment, *Bungie Studios* utilitza el que Isla (2005) anomena *stimulus behaviour* (que en aquest treball es recullen com a *hinted BTs* a l'apartat 3.3.3.3) que consisteix a alterar l'ordre de les prioritats de les tasques en base a un estímul extern.

2.1.2. Farcry 3 (Ubisoft, 2012) i Farcry 4 (Ubisoft, 2014)

La saga *Farcry* s'engloba dintre del gènere *FPS* i *sandbox.* Si bé totes les entregues d'aquesta sèrie s'han anunciat com a jocs de món obert, la realitat és que ho són a partir de la tercera entrega (Thompson, 2016). El repte que va encarar aquesta producció des de el punt de vista de la intel·ligència artificial era encaixar agents que interactuessin amb l'entorn, el jugador, i altres NPCs generant situacions interessants. En jocs més lineals l'equip de disseny s'ocupa de posicionar els enemics per crear aquestes situacions, però en un món obert amb centenars de agents autònoms de diversos tipus això no és viable.

La solució que *Ubisoft* va desenvolupar consisteix en el disseny sistèmic del món. Thompson (2016) defineix el concepte com "una varietat de mecàniques i sistemes independents que, un cop combinats o forçats a interactuar, poden desenvolupar en situacions jugables interessants". L'autor divideix els agents en quatre grups, o sistemes, diferents, NPC's aliats, NPC's enemics, NPC's de civils, i animals. Els agents tenen per sobre un controlador que decideix activar-los o desactivar-los segons la seva distancia amb el jugador entre altres factors (Thompson, 2016).

Thompson (2016) defineix el motor creat per *Ubisoft* com *"*una fàbrica d'anècdotes*"* que funciona magistralment. Tots els sistemes esmentats funcionen amb arbres de comportament, aquest fet cobra una importància especial a arrel de la decisió d'incorporar mascotes per a la quarta entrega de la saga. La decisió sorgeix després de l'èxit que va tenir una fase del tercer joc on el jugador controlava un tigre (Thompson, 2018). La fase esmentada anteriorment en el fons funcionava desactivant temporalment els sistemes

implementats i controlant els comportaments via *scripting*. Però a la quarta entrega controlar les mascotes i usar-les com a arma és una mecànica principal que conviu dintre dels sistemes del joc (Thompson, 2018).

Ubisoft es va veure en una situació en la que qualsevol animal del joc era susceptible de convertir-se en una mascota controlada per al jugador i aquests canvis havien d'implementar-se en un lapse temporal de dos anys. Els desenvolupadors van optar per mantenir el codi font que ja tenien de l'anterior joc i afegir micro-comportaments al BT dels animals domables en temps d'execució (Thompson, 2018). Aquesta tècnica, explicada en detall a la secció 3.3.3.4, consisteix en afegir nodes a un BT o treure'n de forma dinàmica en temps d'execució del programa.

2.1.3. Altres

Els BTs, tal com s'ha esmentat anteriorment, estan fortament arrelats dintre de la indústria triple A (Thompson, 2019). Aquest memòria analitza els casos vistos en les seccions anteriors per la seva especial rellevància en l'ús d'aquest recurs. Tanmateix existeixen nombrosos exemples de renom que en fan ús, per exemple Thompson (2019) recull: Alien Isolation (The Creative Assembly, 2014), Saga Bioshock (2k Boston, Digital Extremes, Feral Interactive, Demiurge Studios, 2007-2013), Tom Clancy's The Division (Ubisoft Massive, Ubisoft Reflections, Ubisoft Redstorm, 2016).

2.2. Editors de BTs

Aquesta secció es centra en justificar l'elecció de les següents eines com a referents i presentar els seus trets diferencials. L'anàlisi en profunditat de les mateixes es realitza a l'apartat 5 ja què és on es s'estudien les funcionalitats susceptibles de ser implementades dintre del marc d'aquest treball.

2.2.1. Unreal Engine 4.21 (*Epic Games,* 2018)

Unreal Engine 4 és un motor de videojocs produït per *Epic Games*. Aquest referent s'ha escollit ja què es l'únic motor de videojocs comercial que integra arbres de comportament per defecte. Això dota d'un especial interès la integració de l'eina dintre de l'estructura de classes del mateix. A més a més degut a que *Epic Games* dona suport al desenvolupament d'aquesta eina es pot considerar que és una solució de "gamma alta".

El motor tracta els arbres com a *assets* que els agents poden executar. Com a *asset,* els BTs són els encarregats de controlar els seus paràmetres d'execució, és a dir, la freqüència amb que s'executen, i les tasques màximes que permeten per *tick,* entre d'altres són part de la informació de l'arbre. Per tal d'assegurar la modularitat del sistema, els agents no tenen control sobre el arbre. De la mateixa forma, les diferents tasques de l'arbre no estan comunicades entre si. Tal com s'explica a l'apartat 3.2.3, *Unreal Engine* opta per utilitzar una *blackboard* com a repositori d'informació compartida. Cal destacar que aquesta

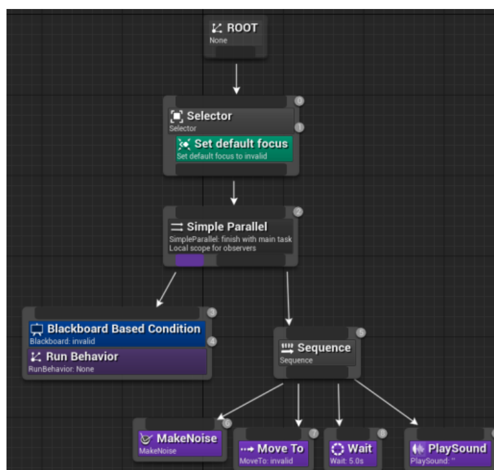
estructura permet fer variables estàtiques, fent que es comparteixi entre totes les instàncies de la mateixa, així com permet l'herència a l'hora de definir noves pissarres.

Si bé els BTs que *Epic Games* incorpora al seu motor són semblants als que vol assolir aquest treball, cal aclarir que no són exactament tal com s'entenen a la concepció clàssica explicada a l'apartat 3.3.2. L'editor d'arbres de comportament d'*Unreal Engine* contempla dos tipus de node a la seva implementació:

- Tasques: Equivalen a les accions dintre de la formulació clàssica dels BTs.
- *Composites* (Nodes Compostos)*:* Són nodes de control de flux. En concret *Unreal Engine* contempla tres tipus: selector, seqüència, i paral·lel simple. Els dos primers funcionen segons la concepció clàssica. L'últim, però, no encaixa amb el que es descriu a la secció 3.3.2.2. El node dissenyat per *Epic Games* permet executar una sola acció principal mentre s'executa un altre comportament en segon pla.

Aquesta eina contempla que ambdós tipus de node puguin tenir modificadors associats. Aquests poden ser o bé decoradors, o bé serveis. Els decoradors funcionen com s'expliquen més endavant a la secció 3.3.2.2. Els serveis, però són un tipus de node paral·lel que s'executa a una freqüència definida per l'usuari.

Figura 2.3. Unreal Engine BT



Font: (Epic Games, 2019)

Al estar integrat dintre del motor de jocs la depuració dels arbres conta amb les mateixes eines que la resta de classes implementades per *Epic Games.* Les tres eines principals de depuració són la pila de crides, la visualització de l'execució en temps real, i els punts d'interrupció (Epic Games, 2019).

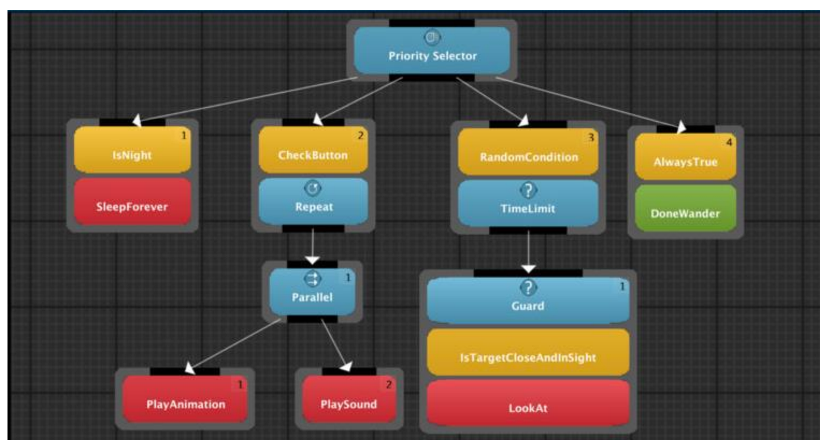
2.2.2. Behaviour Bricks (*Padaone Games,* 2018)

Behaviour Bricks és una extensió de *Unity Engine* publicada per *Padaone Games* l'any 2016. L'eina es considera un referent interessant ja que plasma una versió dels BTs que va

més enllà de la concepció clàssica, tal com la mateixa empresa defineix: "Behaviour Bricks és un motor que implementa l'estat de l'art BTs amb un editor visual" (Padaone Games, 2018). A més a més, en contraposició d'*Unreal Engine* es pot considerar aquesta eina de "gamma baixa" ja què no té un equip dedicat al seu suport.

Com en el cas anterior, la construcció dels BTs es fa al voltant d'una pissarra estàtica que actua com a suport per a realitzar l'intercanvi de dades entre les diferents tasques. En aquest cas la pissarra no existeix com una entitat pròpia sinó que el component que executa l'arbre mostra com a atributs totes les variables que haurien d'estar a la pissarra.

Figura 2.4. Behaviour Bricks BT



Font: (Padaone Games, 2018)

Hi ha dos punts a destacar d'aquest referent, el primer és que integra l'eina dintre del editor de *Unity Engine*¸ el segon punt són els nodes que van més enllà dels que es mostren a l'apartat 3.3.2.2. Aquests són:

- Selector Aleatori: Actua com un selector normal però enlloc d'executar els nodes fills d'esquerra a dreta ho fa aleatòriament.
- Selector Prioritari: La particularitat d'aquest selector és que tots els nodes fills tenen un decorador de condició que s'ha d'avaluar abans d'executar-se. Quan el selector executa els nodes fills comprova les condicions dels nodes que han fallat abans. En cas que una de les comprovacions s'avaluï com a certa executa el fill amb més prioritat.
- Seqüència Aleatòria: Funciona com una seqüència normal amb la particularitat que l'ordre d'execució és aleatori.

Des de el punt de vista d'integració amb el motor de videojocs, *Behaviour Bricks* opta per una solució semblant a la d'*Epic Games.* Els BTs són un *asset* del joc que pot ser executat des de un component que es lliga al agent que ha d'exhibir el comportament. Aquest component es qui, com s'ha esmentat anteriorment, implementa la pissarra i, a diferència del cas anterior, controla els paràmetres d'execució de l'arbre de comportament. *Padaone*

Games incorpora també tres eines de depuració, la pila de crides, els punts d'interrupció, i la capacitat de veure el flux d'execució del arbre (Padaone Games, 2018).

3. Marc Teòric

Aquesta secció del treball recull tots els conceptes teòrics que fonamenten la implementació descrita a l'apartat 4. La secció es divideix en quatre blocs. El primer consisteix en una breu introducció sobre la producció de videojocs. A continuació es dedica un apartat a contextualitzar els arbres de comportament dintre del que s'entén com a IA. El tercer bloc exposa l'estat del art dels BTs. Finalment, la última secció presenta les eines i conceptes que conformen el marc de desenvolupament de l'eina.

3.1. Producció de Videojocs

L'objectiu d'aquesta primera secció és oferir al lector un context que li faciliti la comprensió del procés de producció d'un videojoc atès que l'eina que es vol desenvolupar pren sentit en algunes de les etapes d'aquest procés. En l'apartat següent es mostra qui són els beneficiaris d'una eina com la que es desenvolupa en aquest treball. Mentre que al 3.1.2 es visualitza quan i perquè és útil la implementació de BTs al desenvolupament d'un videojoc. Cal aclarir que aquest treball no s'emmarca dintre de la producció de videojocs i per tant és una breu introducció abans d'entrar a l'objecte del treball.

3.1.1. Rols Dintre la Producció

Rollings i Morris (2004) estableixen un model de rols que es desglosa en els camps de gestió i disseny, programació, art, música, miscelània, suport i QA. La obra citada esmenta que la seva divisió de rols es basa en un cas ideal i que normalment els equips no cobreixen totes les places. Aquest apartat es centra en els dos rols següents:

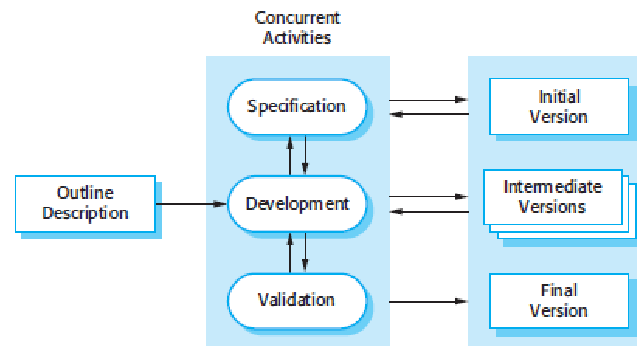
- Programació: És la part del equip responsable de la implementació del joc. Rolling i Morris (2004) recullen que "En un equip de programadors, cadascú s'especialitza en una àrea diferent del programa[...]."(pàg. 250). Els autors a més esmenten que els diferents membres del equip no tenen perquè saber com s'han implementat les parts que estan fora del seu àmbit.
- Disseny: Aquesta part de l'equip, a banda de fer el disseny inicial del producte, són els responsables (conjuntament amb QA) de que estigui ben implementat a nivell d'experiència de joc (Rollings i Morris, 2004).

Millington i Funge (2009) expliquen que "Els arbres de comportament realment brillen quan s'integren amb una interfície gràfica per editar-los. D'aquesta manera els dissenyadors, artistes tècnics, i els dissenyadors de nivells poden ser autors potencials de IAs complexes"(pàg. 334). A més a més, tal com és veu a la secció 3.3 els BTs es basen en la implementació de tasques senzilles que formen accions més complexes mitjançant la seva combinació en l'arbre (Colledanchise i Ögren, 2018).

Unint les idees esmentades es pot observar que els arbres de comportament permeten la flexibilitat que necessita l'equip de programació per dividir-se la implementació de les tasques. A l'hora que un editor gràfic dels mateixos brinda la oportunitat a l'equip de disseny de poder polir els comportaments dels agents del joc sense dependre de ningú.

3.1.2. El Desenvolupament com un Procés Incremental

Figura 3.1. Representació d'un Desenvolupament Incremental



Font: (Sommerville, 2011, pàg. 33)

La figura 3.1 mostra un desenvolupament incremental. Sommerville (2011) descriu aquest procés com la implementació d'una versió preliminar que s'adapta en base al *feedback* dels usuaris. L'autor explica també que tot i tenir clars avantatges generalment aquest tipus de desenvolupament sol comportar problemes de manteniment.

Tal i com es veu en el apartat 3.3 els BTs són un tipus d'estructura dissenyada per tal de desvincular les tasques de la estructura que les escull. D'aquí deriva la gran modularitat dels arbres de comportament. Aquesta característica els fa ideals per desenvolupaments on el manteniment hagi de ser ràpid i senzill (Colledanchise i Ögren, 2018).

3.2. Intel·ligència Artificial

Tot i què la majoria de definicions parteixen de considerar que el terme fa referència a construir sistemes que emulin la ment humana, Wang (2008) exposa que no existeix un consens sobre com definir la IA de forma unívoca. Aquesta falta de consens deriva de les diferents interpretacions del que es comprés com a "intel·ligència humana".

Si bé és cert que no existeix una definició que unifiqui totes les interpretacions que impliquen el terme IA, gran part de les definicions que existeixen són funcionals i vàlides (Wang, 2008). Al llarg d'aquest document es considera com a IA la branca que es centra en els videojocs, que Rabin (2014) defineix com aquella que "Cerca crear l'aparença d'intel·ligència i generar una experiència concreta per l'espectador" (pàg. 4).

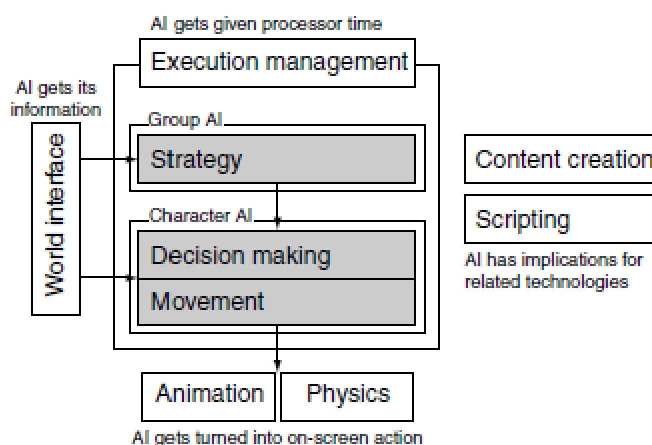
3.2.1. IA als Videojocs

Buckland (2005) estableix dues grans diferències entre la IA que s'investiga a l'Acadèmia i la que s'aplica al desenvolupament de videojocs. El primer tret diferencial parteix de la gestió dels recursos disponibles. L'Acadèmia, segons Buckland (2005), esmenta: "S'han de resoldre els problemes de la forma òptima, fent poc èmfasi en les limitacions de hardware o temps"(pàg. XIX) tot i que aquest punt és cabdal en la implementació d'un videojoc. La segona gran diferència parteix de la naturalesa dels videojocs com a entreteniment. Els agents que es construeixen dintre d'aquest marc es dissenyen per a ser percebuts com a intel·ligents alhora que no sempre encertin. Millington i Funge (2009) comparteixen aquesta visió sobre les diferències entre el món acadèmic i la indústria alhora que matitzen i afegeixen que "els desenvolupadors de jocs sempre extreuen alló que els és útil de les investigacions acadèmiques"(pàg. 484).

La intel·ligència artificial aplicada als videojocs cobra importància gracies a *Pac-Man* (1980) al 1979, tot i que no evoluciona gaire fins al mitjans dels anys 90. Durant aquesta dècada diversos jocs com ara *Goldeneye 007*(1997), *Metal Gear Solid* (1987), o *Warcraft* (1994), experimenten amb el concepte i estableixen la IA com a punts d'interès a l'hora de comprar un joc. D'ençà fins avui dia ha sorgit una gran diversitat de tipus d'agents i encara més maneres d'implementar-los (Millington i Funge, 2009).

Millington i Funge (2009) estableixen tres grans necessitats a l'hora d'implementar un agent en els jocs moderns. La primera consisteix en moure l'agent, la segona és decidir a on moure'l, i la tercera és la necessitat de pensar tàcticament o plantejar estratègies. En base a aquests pilars és construeix el model següent:

Figura 3.2. Model de la IA d'un videojoc



Font: (Millington i Funge, 2009, pàg. 9)

A l'obra citada a la figura 3.2 s'explica que el bloc de moviment comprèn tots els algorismes que converteixen les decisions en moviments. Buckland (2005) divideix aquesta capa en

dues grans seccions, els comportaments motors i la locomoció. Els primers s'encarreguen de calcular la trajectòria i velocitats desitjades, mentre que la segona representa el com l'agent aplica aquest comportament. L'autor sosté que aquesta separació permet reutilitzar els mateixos comportaments aconseguint resultats diferents.

Millington i Funge (2009) defineixen la secció de presa de decisions partint de la premisa que els agents tenen diversos comportaments que poden executar i un conjunt de regles per escollir quin es més escaient en cada moment. L'objecte d'aquest treball forma part d'aquest bloc que s'explica en més detall a la secció següent.

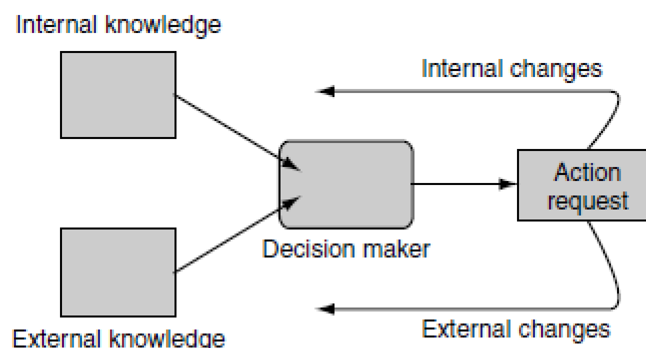
Tot i què el model parla de tres grans seccions, el autors defineixen que existeix un conjunt de tècniques compreses entre la presa de decisions i els algoritmes de moviments. L'objectiu d'aquestes és el de cercar la millor opció per desplaçar l'agent. Aquest conjunt de tècniques s'engloben sota el nom de *pathfinding*.

La secció d'estratègia cobra rellevància quan cal coordinar diversos agents. Aquests tenen les seves estratègies de decisió, moviment, i *pathfinding* a nivell individual que influeixen i alhora són influïdes per altres agents (Millington i Funge, 2009).

3.2.2. Presa de Decisions

Millington i Funge (2009) recullen que tot i existir diversos sistemes de presa de decisions, la majoria es basen en la mateixa idea: "L'agent processa un set d'informació i genera una acció per executar. Aquest sistema rep com a entrada el coneixement que te l'agent i genera com a sortida una petició d'acció." (pàg. 293). Cal destacar que aquets dos autors divideixen el coneixement del de l'agent entre el referent al estat intern del mateix, i els estímuls que rep de l'exterior, tal com es representa a la figura 3.3.

Figura 3.3. Esquema de la presa de decisions



Font: (Millington i Funge, 2009, pàg. 294)

3.2.2.1. Sistemes Deliberatius i Sistemes Reactius

Ghallab, Nau, i Traverso (2016) expliquen que un sistema deliberatiu és aquell que decideix les accions i com executarles per tal d'assolir un objectiu en base a un raonament. Els autors

expliquen que “Aquest raonament consisteix en emprar models predictius de les capacitats de l’agent i el seu entorn per simular el resultat d’executar una acció” (pàg. 2). El resultat d’un sistema del·liberatiu és un pla d’acció (Ghallab, Nau, i Traverso, 2016).

Colledanchise i Ögren (2018) defineixen la reactivitat com la capacitat de reaccionar de forma ràpida i eficient als canvis. Dintre d’aquest marc engloben totes les estratègies de canvis de tasques o comportaments. Entre aquestes destaquen les màquines d’estats finits, els arbres de decisió, i els arbres de comportament, que s’expliquen en els apartats subsegüents. El resultat d’un sistema reactiu és una acció a executar.

Tal com es pot observar a la obra de Millington i Funge (2009), gran part de les estratègies de presa de decisions en videojocs es basen en sistemes reactius. Això es deu a que en moltes ocasions els agents disposen d’una serie finita de comportaments, que al intercanviarse per adapta-lo a les necessitats del moment creen la il·lusió d’inteligència que es busca.

3.2.2.2. Finite State Machines

Rabin (2014) exposa què les màquines d’estats finits (o *FSM*) són el mecanisme més utilitzat pels programadors de videojocs a l’hora d’implementar els agents del joc. Això es deu a que el concepte darrere d’aquest model és simple i ràpid d’implementar. El seu treball descriu que:

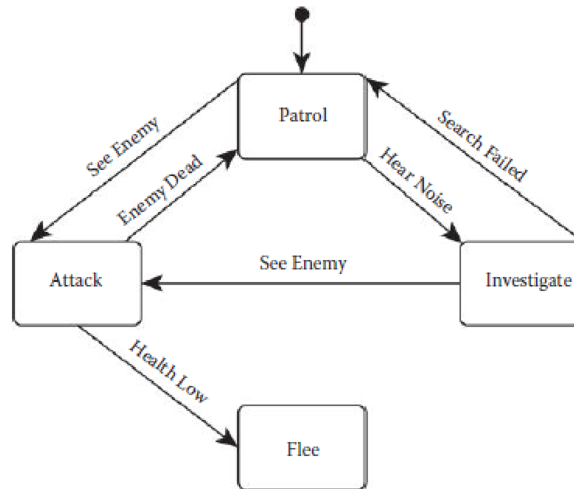
> Una FSM desglossa la IA d’un NPC en peces anomenades estats. Cada estat representa un comportament específic, o una configuració interna, i només un sol estat és considera actiu en cada moment. Els estats connecten entre si mitjançant transicions. Aquestes són connexions direccionals que canvien l’estat actiu en cas de que es compleixin certes condicions (pàg. 48).

Els estats de les FSM són els que implementen les transicions als altres estats. Aquestes transicions són unidireccionals i requereixen de una condició explícita per funcionar. En certa manera funcionen com una funció *goto* (Colledanchise i Ögren, 2018).

Colledanchise i Ögren (2018) alerten que els desaventatges de les FSM apareixen quan es generen sistemes complexos amb molts estats. En concret expliquen que el problema principal són les transicions. Aquestes transicions generen problemes de:

- Manteniment: Al afegir o eliminar estats s’han de revisar totes les transicions i estats interns de la FSM.
- Escalabilitat: En sistemes complexos la lectura i interpretació de les transicions és confusa i difícil de modificar.
- Modularitat: Els estats són qui controlen cap a on transiten. Aquest fet fa que sigui complicat reutilitzar un estat en màquines diferents.

Figura 3.4. Exemple d'una FSM



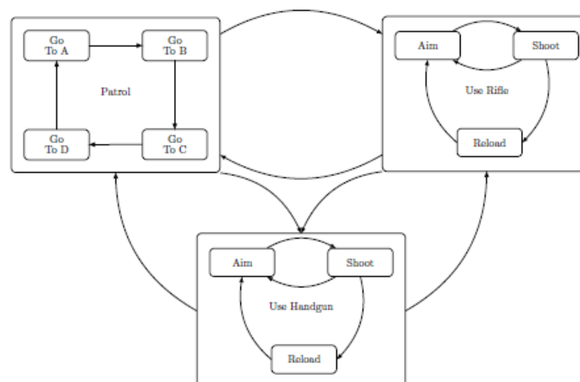
Font: (Rabin, 2014, pàg. 48)

3.2.2.3. Hierarchycal Finite State Machines

Colledanchise i Ögren (2018) expliquen que les màquines d'estats jeràrquiques alleugen els problemes de les FSM ja que "En una HFSM un estat pot ser contenidor de un o més sub-estats. Un estat d'aquest tipus s'anomena superestat." (pàg.24).

Les HFSM són un conjunt de FSM unides a traves de transicions genèriques. Aquest tipus de transició representa una transició de qualsevol dels sub-estats d'un superestat a qualsevol dels sub-estats d'un altre superestat. Aquesta transició, com en el cas de les FSM, és direccional i canviarà de comportament en cas que es compleixi una condició (Colledanchise i Ögren, 2018).

Figura 3.5. Exemple d'una HFSM



Font: (Colledanchise i Ögren, 2018, pàg. 25)

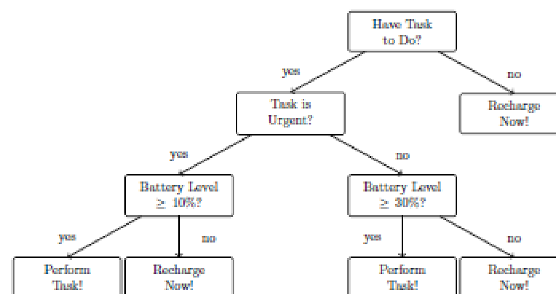
Els citats autors conclueixen que les HFSM presenten una millor modularitat i escalabilitat que les FSM. Tot i seguir presentant problemes de manteniment en sistemes complexos.

3.2.2.4. Decision Trees

Colledanchise i Ögren (2018) defineixen els arbres de decisió com una estructura que es composa de "Una llista de if-else encaixats uns dins del altres que s'empren per prendre decisions. Les fulles del arbre descriuen les accions que s'ha decidit executar, mentre que la resta de nodes són predicats a avaluar."(pàg.35).

Les condicions de cada decisió són comprovacions simples. Si una consulta es composa de varies condicions, aquestes es desglossen en preguntes ordenades per prioritat. A més a més, generalment, les decisions tenen una resposta binaria, és a dir, una pregunta sol desglossar-se en dues opcions (Millington i Funge, 2009).

Figura 3.6. Exemple d'un Decision Tree



Font: (Colledanchise i Ögren, 2018, pàg. 36)

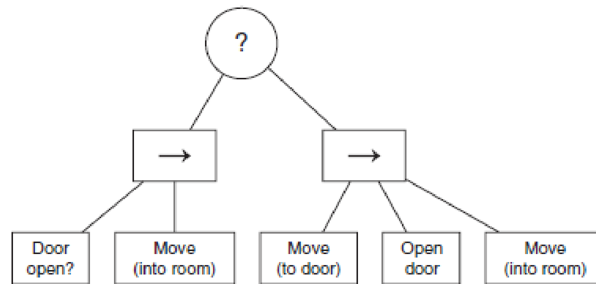
Tot i ser altament modulars, intuïtius, i fàcils d'implementar els arbres de decisió presenten un gran problema. Degut a la poca informació que es pot extreure dels seus nodes la gestió d'errors es complica (Colledanchise i Ögren, 2018).

3.2.2.5. Behaviour Trees

Millington i Funge (2009) descriuen els arbres de comportament com una síntesi de diverses tècniques tals com les HFSM, planificació, execució d'accions, i *scheduling,* organitzades de forma que inclús els desenvolupadors que no tenen un perfil tècnic poden crear-los. El nucli dels BT s'estableix en base al concepte de *tasca*. Millington i Funge (2009) expliquen:

> Les tasques es composen de sub-arbres per representar accions complexes. De la mateixa forma, aquestes tasques poden formar part d'una tasca superior. Aquesta modularitat és la que fa els arbres de comportament interessants. Les tasques implementen una interfície comuna i són autocontingudes, això permet generar estructures jeràrquiques sense preocupar-se de les implementacions de les sub-tasques (pàg. 334).

Figura 3.7. Exemple d'un arbre de comportament



Font: (Millington i Funge, 2009, pàg. 337)

Tot i que es realitza una explicació més detallada dels arbres de comportament a la secció 3.3, cal destacar que els BTs són una tècnica més costosa en temps de computació que les vistes anteriorment (Rabin, 2014).

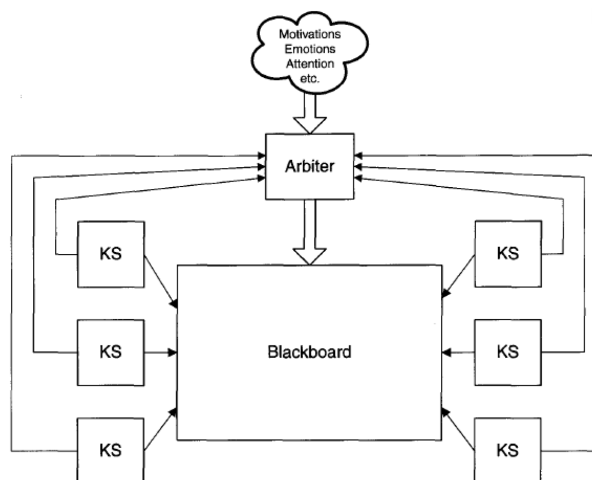
3.2.3. Blackboards

A Rabin (2002) s'afirma que: "Tots els tipus de sistemes socials requereixen coordinar accions. […] El mateix tipus de problemes apareixen al construir agents. […] Ja què dos comportaments poden voler coses contradictories" (pàg.333).

Les *blackboards* apareixen com a metàfora d'una pisarra física. En ambdós casos la pisarra proveeix un espai compartit en el que l'estat d'un problema i la seva solució són visibles. (Rabin, 2002). Champanard (2007) matitza a la seva web que: "L'aventatge de les pissarres es que proveeixen modularitat. Els diferents sistemes deixen de dependre entre ells i comparteixen la informació a través de la pissarra".

Champanard (2007) parla de les pissarres com a contenidors d'informació. En concret l'obra parla de pissarres estàtiques ja que la informació que poden contenir es fixa en la fase de disseny i la informació que s'hi enmagatzema no sol ser ampliable. L'autor proposa un sistema basat en l'arquitectura de pissarra canònica mitjançant el qual els diferents comportaments d'un agent poden comunicar-se entre si.

Figura 3.8. Arquitectura de Pissarra Canònica



Font: (Rabin, 2002, pàg. 334)

En essència aquest model es divideix en tres grans blocs: *blackboard,* fonts de coneixement (*knowledge source* en anglès), i l'àrbitre. Rabin (2002) els defineix com:

- *Blackboard:* És un panell d'informació públic i compartit entre els individus o comportaments.
- Fonts de coneixement (*KS):* És el conjunt d'entitats que operen amb la pissarra. Quan aquests actuen modifiquen l'estat de la *blackboard.*
- Àrbitre: És l'entitat que gestiona les peticions d'acció dels *KS.* En funció de l'estat del agent o de l'assoliment de la meta, l'àrbitre selecciona un únic *KS* a executar. Cal destacar que aquesta figura desapareix en el sistema proposat per Champanard (2007).

3.3. Behaviour Trees en Profunditat

Aquesta secció es centra en presentar l'estat de l'art dels BTs. En concret es para especial atenció a les característiques importants de cara a la seva implementació en el món dels videojocs.

3.3.1. BTs als Videojocs

Colledanchise i Ögren (2018) documenten que el naixement dels BTs es produeix dintre de la indústria dels videojocs. Els autors exposen que aquests van néixer "per incrementar la modularitat de les estructures de control dels NPCs. Ja què en aquesta indústria billonaria la modularitat és una propietat clau que permet la reutilització de codi, el disseny incremental de funcionalitats, i un testeig eficient." (pàg. 4).

3.3.2. Concepció Clàssica

La font principal d'aquest apartat són Colledanchise i Ögren (2018). En cas que no s'explicíti una altra font es fa constar que la informació s'extreu d'aquesta obra.

3.3.2.1. Funcionament

Un BT és una estructura en forma d'arbre arrelat i direccional. El concepte d'arrelat neix del fet que existeix un primer node que no depèn de cap altre per executar-se. El de direccional fa referència a que l'execució del arbre segueix un ordre concret (de dalt a baix i d'esquerra a dreta).

Dintre de l'arbre de comportament existeixen dos tipus de node. Els nodes interns s'anomenen nodes de control de flux. Els externs solen anomenar-se nodes d'execució o accions. La diferència entre ambdós és que els nodes interns tenen almenys un node fill i els externs (també anomenats fulles) no en tenen cap. Cal destacar que tots els nodes d'un BT tenen un node pare a excepció del node arrel.

Colledanchise i Ögren (2018) descriuen el funcionament dels arbres de comportament de següent forma:

> L'execució parteix del node arrel que genera una senyal anomenada tick que permet l'execució d'un altre node. Aquesta senyal navega a traves dels nodes fills que s'executen si, i tan sols si, reben aquest tick. El node fill retorna sempre un estat que pot ser running si esta en execució, succes si ha acabat la tasca de manera satisfactoria, i *failure* en cas que no hagi acabat bé (pàg.6).

3.3.2.2. Tipus de nodes

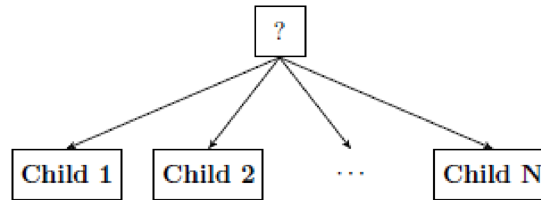
Colledanchise i Ögren (2018) exposen que en la concepció clàssica dels BTs existeixen cuatre tipus de nodes de control de flux (selector, seqüencia, paral·lel, i decorador) i dos tipus de nodes d'execució (condició i acció).

- Selector (*Selector/Fallback*): Millington i Funge (2009) expliquen el seu funcionament com:

  > Retorna immediatament amb *success* en cas que un dels seus fills s'executi o finalitzi amb èxit. Mentre no succeeixi això executa el fill següent. En cas de que cap fill finalitzi amb èxit o estigui executant la seva tasca retorna *failure* (pàg. 335).

  Sol representar-se amb l'etiqueta [?] i es important remarcar que en cas que un dels fill retorni *success* no executa cap altre node (Colledanchise i Ögren, 2018).

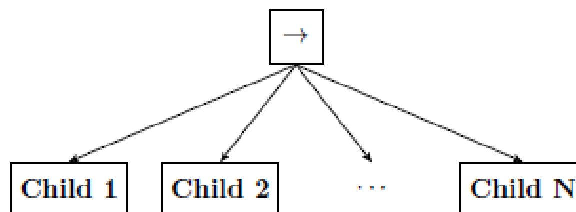
Figura 3.9. Representació gràfica d'un Selector



Font: (Colledanchise i Ögren, 2018, pàg. 7)

- Seqüencia (*Sequence):* Aquest node retorna *success* tan sols en el cas que tots els seus fills s'hagin executat amb èxit. En la resta de casos executa els seus fills fins que trobi un que retorni *running* o *failure.* En aquest cas torna el mateix valor que el seu fill i atura la seva execució. Sol representar-se amb l'etiqueta [→].

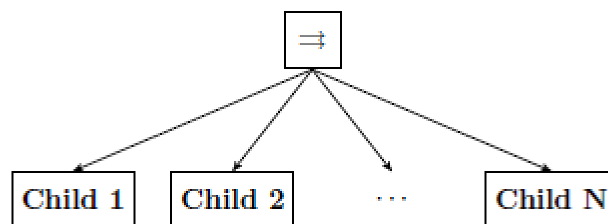
Figura 3.10. Representació gràfica d'una Seqüencia



Font: (Colledanchise i Ögren, 2018, pàg. 7)

- Paral·lel (*Parallel*): L'execució d'un node paral·lel es considera que ha finalitzat amb èxit si un nombre *M* de fills retorna *success,* on *M* és un nombre definit per l'usuari. El node falla si el total de fills – *M* + 1 retornen *failure.* En qualsevol altre cas torna *running*.

Figura 3.11. Representació gràfica d'un Paral·lel



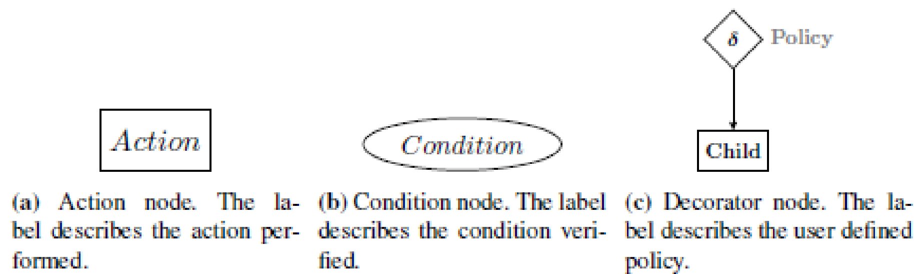
Font: (Colledanchise i Ögren, 2018, pàg. 8)

- Decorador (*Decorator):* És un tipus de node de control de flux que tan sols pot tenir un node fill. Modifica el valor de retorn del fill i afegeix requisits especials d'execució pel mateix. Exemples clàssics són invertir el valor del fill, permetre

l'execució durant x segons i després fallar sinó s'ha completat la tasca, o permetre que el node fill falli com a màxim *N* vegades.

- Condició: Els autors defineixen que aquest tipus de node mai pot tornar el valor *running*. La seva execució consisteix en comprovar un predicat i retornar *success* o *failure* en funció del mateix.
- Acció: Colledanchise i Ögren (2018) recullen que "Quan rep un tick una Acció executa una comanda. Retorna *success* si l'acció finalitza amb èxit, o *failure* sinó ha pogut completar-se. En qualsevol altre cas retorna *running*"(pàg.7).

Figura 3.12. Representació gràfica d'Acció, Condició, i Decorador



(a) Action node. The label describes the action performed.

(b) Condition node. The label describes the condition verified.

(c) Decorator node. The label describes the user defined policy.

Font: (Colledanchise i Ögren, 2018, pàg. 8)

Taula 3.1. Tipus de Node d'un BT

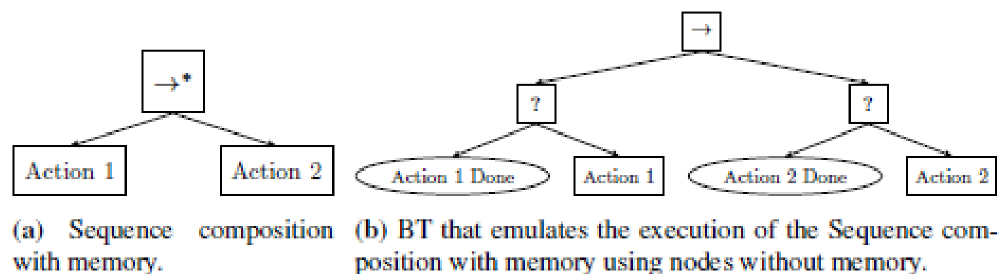
Node type	Symbol	Succeeds	Fails	Running
Fallback	?	If one child succeeds	If all children fail	If one child returns Running
Sequence	→	If all children succeed	If one child fails	If one child returns Running
Parallel	⇉	If $\geq M$ children succeed	If $> N - M$ children fail	else
Action	text	Upon completion	If impossible to complete	During completion
Condition	text	If true	If false	Never
Decorator	◊	Custom	Custom	Custom

Font: (Colledanchise i Ögren, 2018, pàg. 9)

3.3.2.3. Nodes amb Memòria

Els BTs són essencialment reactius, per aquest motiu la seva execució sempre parteix de l'arrel i comprova tots els nodes que calguin abans d'executar una acció. A vegades, però a l'hora d'implementar un BT el dissenyador ja sap que certs nodes no han de repetir la seva execució. Per facilitar aquest tipus d'implementació existeixen els nodes amb memòria. Aquests recorden l'estat que han retornat els fills en execucions anteriors i no els executen. Un cop el node pare ha finalitzat la seva tasca (tan amb èxit com fracassant) tornarà a avaluar tots els fills, guardant el nou valor de devolució. En altres paraules, la memòria s'allibera quan el node retorna quelcom diferent de *running* (Colledanchise i Ögren, 2018)

Figura 3.13. Comparativa de la implementació de un BT amb memòria i sense



(a) Sequence composition with memory.

(b) BT that emulates the execution of the Sequence composition with memory using nodes without memory.

Font: (Colledanchise i Ögren, 2018, pàg. 11)

3.3.2.4. BTs No Reactius

Colledanchise i Ögren (2018) anomenen com a BT no reactiu a aquells arbres de comportament que no contemplen el retorn del valor *running* en els seus nodes. Aquest model no permet que l'arbre reaccioni als canvis més enllà de la reacció que pugui fer l'acció que s'està executant. Tot i estar més limitats que els BTs reactius el seu ús és força extés tal com recullen Millington i Funge (2009).

3.3.3. Noves Fronteres

Aquest apartat presenta els límits actuals de l'estat de l'art. Els següents títols mostren tècniques que encara no s'han consolidat com a pràctiques comuns, o bé són experimentals.

3.3.3.1. Utility BTs

Rabin (2014) descriu la teoria de la utilitat com "El procés de mesurar la utilitat relativa d'una acció particular" (pàg. 113). Exposa seguidament que per prendre bones decisions s'ha de quantificar el valor real de l'acció enlloc de comprovar que senzillament és una acció valida.

En base a aquesta teoria és pot implementar un selector que enlloc d'executar els fills en un ordre prefixat ho faci en base a la possible utilitat de les accions (Rabin, 2014). Colledanchise i Ögren (2018) exposen però que la propagació de la utilitat al llarg de l'arbre és confusa i presenta certes dificultats que encara no tenen una solució consensuada.

3.3.3.2. Stochastic BTs

Colledanchise i Ögren (2018) expliquen que és un plantejament semblant al del apartat anterior però enlloc de computar la utilitat de les accions es valora les probabilitats d'èxit de les mateixes. Si bé és més senzill d'implementar presenta un problema. En cas d'existir dues accions amb altes probabilitats d'èxit el sistema sempre executarà la mateixa.

3.3.3.3. Hinted Execution BTs

Ocio (2012) exposa un sistema que permet modificar l'execució dels BTs en temps real sense modificarlos. La idea parteix d'alterar momentaniament la prioritat dels nodes del BT mitjançant pistes que els desenvolupadors fan arribar al node de control. El BT intenta executar primer les suggerencies però en cas que no sigui posible executarà el seu comportament per defecte.

Aquestes suggerencies poden ser tant positives, accions que es volen prioritzar, com negatives, accions que es volen evitar. Aquest tipus de pistes solen comportar alterar l'ordre dels fills en els selectors. Tot i que també poden permetre l'accés a una branca específica del BT (Colledanchise i Ögren, 2018).

3.3.3.4. Dynamic Expansion of BTs

Flórex-Puga, Gómez-Martín, Díaz-Agudo, i González-Calero, (2008) expliquen que l'objectiu d'aquesta tècnica és permetre que els dissenyadors puguin definir les propietats d'un comportament sense concretar-lo. Els autors expliquen que "Els arbres de comportament dinàmics són una extensió dels BTs on alguns nodes del arbre emmagatzemen consultes en lloc de comportaments. L'arbre s'expandeix en temps d'execució substituint les consultes per comportaments mitjançant un procés basat en la similitud" (pàg.1).

3.4. Eines

L'objectiu d'aquesta secció és detallar la base teòrica de l'entorn en el que s'implementa l'eina objectiu. És menester aclarir que no es realitza una descripció exhaustiva de les eines ja que no forma part de l'objecte d'estudi del treball. Tanmateix, es busca dotar de context al lector i oferir una visió global de les principals característiques que s'usen de les mateixes.

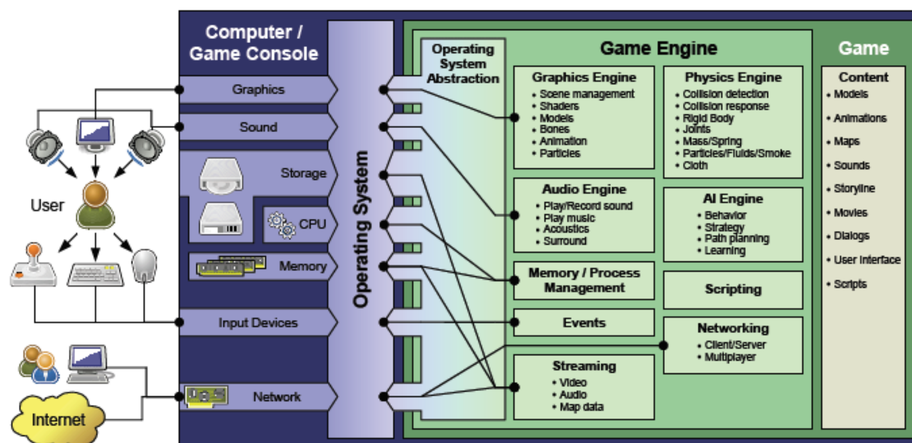
3.4.1. Unity Engine

Tal com *Unity* (2019) exposa, el seu *software* homònim és àmpliament emprat a la indústria "*Unity* s'utilitza per crear la meitat dels jocs del món. La nostra plataforma en temps real, impulsada per eines y serveis, ofereix increïbles possibilitats per desenvolupadors de jocs i creadors de diferents indústries i aplicacions." A continuació es defineix què s'entén com a motor de jocs, com s'estructura l'*engine* de *Unity*, i finalment quines característiques té com a editor.

3.4.1.1. Motors de Jocs

Marks, Windsor i Wünsche (2007) estipulen que un motor de jocs és un *software* necessari per desenvolupar videojocs. Aquest, permet desvincular la creació del joc del *hardware* on s'executa, reduint així la dificultat tècnica de la implementació (Marks, Windsor, & Burkhard, 2007).

Figura 3.14. Blocs Funcionals d'un Motor de Jocs



Font: (Marks, Windsor, & Burkhard, 2007)

Els autors, en consonància amb el que recull Unity (2019), exposen que un motor de jocs es composa de diversos blocs. Generalment aquests són els representats a la figura 3.14. Marks, Windsor i Wünsche (2007) els defineixen com:

- Motor Gràfic: És el bloc encarregat de carregar, mostrar, i en definitiva, gestionar, la part visual del joc. Aquesta inclou des de models 3D fins a textures o partícules.
- Motor Físic: Serveix per simular comportaments realistes dels diferents elements físics del joc. Aquest bloc implementa models matemàtics per a la simulació física de cossos rígids de formes arbitràries, cossos articulats, etc.
- Motor d'Àudio: És l'encarregat de gestionar tots els elements sonors del joc així com les seves característiques.
- Gestió de Memòria: En els jocs moderns no es possible tenir tots els elements carregats simultàniament per a l'execució del mateix. Aquest bloc s'encarrega de la gestió de càrrega i purga de la memòria.
- Esdeveniments: Filtra i processa tots els esdeveniments del sistema. *Timers, inputs,* connectivitat, tots s'unifiquen i es distribueixen un únic bucle de joc.
- *Streaming:* Cert tipus de dades tals com la música o els vídeos no és necessari que estiguin carregats en memòria per poder utilitzar-los. Aquest bloc es l'encarregat de produir aquest tipus de càrrega.
- *Networking*: És el bloc encarregat de gestionar tot el referent a connectivitat entre el joc i els servidors.
- Motor d'IA: Els autors recullen que es sol incorporar solucions genèriques tals com algoritmes de cerca de camins, màquines d'estat, etc.
- *Scripting:* És en essència el bloc que permet editar el comportament del joc. Marks, Windsor, & Burkhard (2007) exposen que "La flexibilitat d'un motor de jocs és la característica més important a l'hora de crear contingut variat. Això s'aconsegueix

gràcies al sistema *d'scripting* que permet un accés directe a les funcions del motor" (pàg. 275).

Marks, Windsor i Wünsche (2007) expliquen que els motors de joc són especialment interessants ja què es mantenen constantment actualitzats. A més a més, tal com *Unity* (2019) també recull, gestionen el mateix contingut per a fer funcionar el joc en plataformes de diferents característiques.

3.4.1.2. Unity com a Motor de Jocs

Unity Engine en essència s'estructura segons els blocs vistos a l'apartat anterior (3.4.1.1). En aquest apartat s'explica l'aproximació de *Unity* a la usabilitat dels mateixos. En concret es parla de com s'estructura els blocs d'*scripting*, i de quines eines dota a l'usuari el bloc d'IA.

El primer que cal destacar és que, a diferència d'altres programes similars com podria ser *Unreal Engine* (vist a l'apartat 2.2.1), *Unity Engine* és un *software* tancat. És a dir, el codi font no és accessible ni modificable. *Unity* centra els seus esforços en generar una documentació amplia per a entendre el seu sistema d'*scripting*. Dotant del mateix d'una importància cabdal, ja què és la via d'extensió del motor a banda del sistema de programació del joc (Unity, 2019).

El sistema d'*scripting* de *Unity* actua com un eix que permet editar tots els altres blocs. El llenguatge escollit per a tal fi és *C#* del que es parla a l'apartat 3.4.2. Aquest sistema és força flexible pel que fa a dotar de llibertat a l'usuari a l'hora d'encarar el seu projecte dintre del paradigma de la programació orientada a objectes (Unity, 2019).

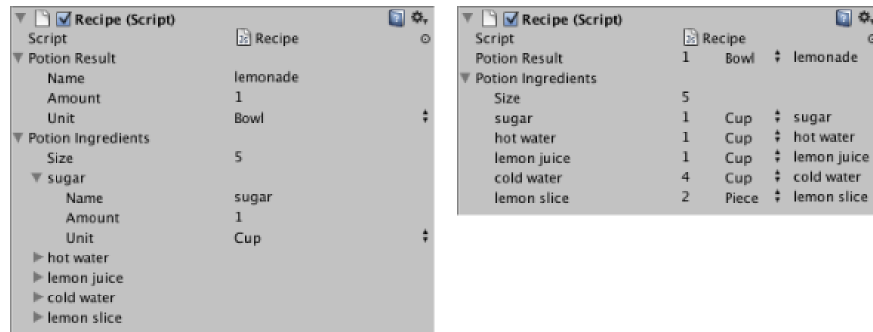
L'altre gran bloc que afecta directament a l'objecte del treball és el d'intel·ligència artificial. Tal com s'ha esmentat anteriorment (3.4.1.1) els motors solen dotar de solucions genèriques d'IA. En el cas de *Unity Engine,* els recursos que ofereix són eines de *pathfinding* (Unity, 2019).

3.4.1.3. Unity com a Editor

Unity permet generar extensions del motor de tres formes (Unity, 2019):

- Inspectors Personalitzats: Un inspector de *Unity Engine* és una finestra que permet veure i modificar certes propietats d'un objecte. *Unity* dota a l'equip de desenvolupament de recursos per poder modificar l'aparença i funcionalitats dels elements que es mostren a l'inspector per tal de fer-lo més usable. La figura 3.15. mostra a l'esquerra l'inspector per defecte d'una classe *Recipe.* A la dreta un inspector personalitzat per a la mateixa classe (Unity, 2019).

Figura 3.15. Exemple d'Inspector Personalitzat



Font: (Unity, 2019)

- Finestres d'Editor: L'API de *Unity* (2019) recull el procés que permet crear una finestra nova a l'editor de *Unity Engine*. Aquest finestra es pot dotar de la funcionalitat desitjada, fet que converteix aquesta via d'extensió de l'editor en la més recomanada per dotar de d'interfície gràfica un subsistema del joc (Unity, 2019). La figura 3.16. mostra un exemple de finestra personalitzada.

Figura 3.16. Exemple de Finestra Personalitzada

Font: (Unity, 2019)

- *Property Drawers*: Permeten alterar l'aparença de els membres d'un *script* que implementin un atribut personalitzat (Unity, 2019).

3.4.2. C#

La sintaxis del llenguatge s'inspira i alhora simplifica l'ús de la sintaxis de C++. En essència, *C#* proveeix de solucions sintàctiques a les accions més comuns de *C++, C,* i *Java* (Microsoft, 2015). Microsoft (2015) defineix C# com "Un elegant llenguatge orientat a objectes fortament tipificat que permet als desenvolupadors construir una gran varietat de aplicacions segures y robustes que s'executen dintre del entorn de treball .NET."

3.4.2.1. Programació Orientada a Objectes

La documentació de Microsoft (2015) explica que C# és un llenguatge orientat a objectes. Com a tal, l'autor de la mateixa recull que "C# suporta els conceptes d'encapsulació, herència, i polimorfisme. Totes les variables i mètodes s'encapsulen dintre de definicions de classe. Una classe pot heretar directament d'una única altra classe, però pot implementar qualsevol nombre d'interfícies. *"*

Microsoft (2015) també recull que les *structs* són una versió més lleugera de les classes. Això es concreta en el fet que no suporten herència, però si que poden implementar interfícies.

Microsoft (2015) especifica què a banda d'aquestes característiques, C# també inclou:

- Delegats: Microsoft (2015) els defineix com "Signatures de mètodes encapsulades que permeten notificacions de events fortament tipificats."
- Propietats: També anomenats *accessors* són un tipus especial de mètodes que permeten definir l'accessibilitat de membres privats d'una classe.
- Atributs: Declaren metadades sobre els tipus de dades quan el programa s'executa.
- Comentaris en XML per a fins de documentació.
- LINQ (*Language-Integrated Query*): *C#* suporta consultes LINQ.

3.4.2.2. Programació Genèrica

Microsoft (2015) recull que a partir de la versió 2.0 de C# s'introdueixen els tipus genèrics, a més, exposa:

> Aquests introdueixen el concepte de paràmetres de tipus a l'entorn de treball .NET. Amb aquest concepte es possible dissenyar classes i mètodes que deleguen la especificació de un o més tipus fins que la classe es declarada i instanciada per el codi d'un tercer.

3.4.2.3. Reflexió

La reflexió proveeix l'usuari d'eines que permeten explorar i accedir als membres, mètodes, i atributs d'un tipus de dada. Això és especialment útil en cas de voler accedir a les metadades d'un membre, per examinar i instanciar tipus en temps d'execució, etc. (Microsoft, 2015).

4. Objectius

L'objectiu principal d'aquest treball és el desenvolupament d'una eina que integri la capacitat d'implementar i editar arbres de comportament al motor de jocs *Unity Engine*. A fi de concretar quin és l'abast real d'aquest objectiu es desglossa de la forma següent.

4.1. Objectius Principals

- La creació d'una llibreria que permeti implementar i utilitzar BTs en la seva formulació clàssica així com la documentació pertinent per poder utilitzar-la.
- Dotar la llibreria abans esmentada d'una interfície d'edició gràfica dintre de *Unity Engine.*

4.2. Objectius Secundaris

- Dotar la llibreria creada d'un sistema de depuració.
- Donar suport a la creació de *Hinted BTs.*

5. Disseny Metodològic i Cronograma

La secció que aquí comença te com a objecte establir les línies mestres del desenvolupament de la part pràctica del treball. En primer lloc es realitza l'anàlisi en profunditat dels referents presentats a l'apartat 2.2. i s'estableix la taula de funcionalitats que s'inclouen a la eina final. A continuació, s'estableixen els requisits del desenvolupament. Tot seguit es presenta la metodologia emprada i les decisions que afectaran a tot el projecte. Per finalitzar, es presenta un cronograma on s'estima les diferents dates clau.

5.1. Estudi Previ al Desenvolupament

L'objectiu d'aquesta secció és definir quines funcionalitats són les desitjades a l'eina desenvolupada. L'estudi comença analitzant quines de les *features* destacades de les eines referents s'inclouen dintre del llistat de les desitjades explicant el perquè es descarta la resta. Finalment s'elabora una taula amb les *features* diferencials de la eina pròpia.

L'apartat 2.2. recull els trets diferencials de les eines referents. Ambdues eines, però, tenen forces funcionalitats comuns a banda de les ja esmentades. Les més destacades es recullen a la taula 5.1:

Taula 5.1. Funcionalitats Comunes a Ambdues Eines Referents

Funcionalitats Comunes a Ambdues Eines		
ID	**Funcionalitats**	**Desitjada**
1	Guardar els BTs com un asset dintre del projecte	Si
2	Parametritzar les tasques màximes per *Tick*	Si
3	Ús d'una pissarra per desvincular les tasques de la resta	Si
4	Element extern que executa l'arbre	Si
5	Pila de crides	Si
6	Visualització de la execució en temps real	Si
7	Customització del node	Si
8	Nodes clàssics	Si
9	Creació de *Decorators* propis	Si
10	Creació de *Actions* propies	Si
11	Creació de *Conditions* propies	Si
12	Diferents colors segons el tipus de node en l'editor visual	Si
13	Index del node visible en l'editor visual	Si
14	*Breakpoints*	No

Font: Elaboració Pròpia

Com es pot observar a la taula 5.1 la única *feature* que ambdues eines implementen i es descarta són els punts de ruptura. Aquesta decisió es pren en base al temps disponible de desenvolupament. Tal com es mostra a l'apartat 5.2, el temps de producció és breu, i en ares

de dotar l'eina d'un sistema de depuració es consideren més interesants les funcionalitats 5 i 6. Cal matisar que si bé es descarta com una funcionalitat primordial de l'eina, es susceptible d'afegir-se més endavant en cas de produir-se una producció més veloç del que s'estima.

Taula 5.2. Funcionalitats dels BTs d'Unreal Engine

	Unreal Engine BT	
ID	**Funcionalitats**	**Desitjada**
1	Pissarra amb variables estàtiques	Si
2	Comentaris visibles en l'editor visual	Si
3	Parametritzar la frequencia d'execució del arbre	Si
4	Creació de *Services*	No
5	*Simple Parallel* permet l'execució paral·lela de un node i un arbre	No
6	Els parametres d'execució del BT formen part del mateix	No
7	*Service* com a codi d'execució paral·lela	No

Font: Elaboració Pròpia

La taula 5.2. mostra com es descarta la aproximació d'*Unreal Engine* pel que fa al tracte dels nodes paral·lels (*features 4,5,7).* El criteri aplicat parteix de la base que el node *Simple Parallel* es pot suplir modificant lleugerament el disseny de l'arbre, o bé creant un *decorator* personalitzat que implementi aquesta funció. Aquest criteri s'aplica també al node *Service*. D'altra banda es descarta la funcionalitat 6 ja què l'eina hauria de permetre que dos agents amb un mateix BT l'executessin de forma diferent en funció del criteri del equip desenvolupador.

Taula 5.3. Funcionalitats de Behaviour Bricks

	Behaviour Bricks	
ID	**Funcionalitats**	**Desitjada**
1	Els parametres d'execució del BT formen part del ent executor	Si
2	*Random Selector*	No
3	*Priority Selector*	No
4	*Random Sequence*	No

Font: Elaboració Pròpia

Tot i que cap de les anteriors eines permeten la implementació de nodes de flux personalitzats, aquest treball pretén permetre aquestes llibertats a nivell de llibreria. Aplicant el mateix tipus de raonament les funcionalitats 2, 3, i 4 de la taula 5.3. es descarten ja què els nodes bàsics seran ampliables.

Taula 5.4. Funcionalitats Desitjades

Funcionalitats desitjades	
ID	**Funcionalitats**
1	Guardar els BTs com un asset dintre del projecte
2	Ús d'una pissarra per desvincular les tasques de la resta
3	Pissarra amb variables estàtiques
4	Nodes clàssics
5	Creació de *Selectors* propis
6	Creació de *Sequences* propies
7	Creació de *Decorators* propis
8	Creació de *Actions* propies
9	Creació de *Conditions* propies
10	Customització del node
11	Element extern que executa l'arbre
12	Els parametres d'execució del BT formen part del ent executor
13	Parametritzar la frequencia d'execució del arbre
14	Parametritzar les tasques màximes per *Tick*
15	Diferents colors segons el tipus de node en l'editor visual
16	Index del node visible en l'editor visual
17	Pila de crides
18	Visualització de la execució en temps real
19	Comentaris visibles en l'editor visual
20	*Hinted BTs*

Font: Elaboració Pròpia

La taula 5.4. recull la llista de *features* que es desitgen implementar. Com es pot observar la major part son comunes a les que implementen les solucions referenciades. La llista, però, intenta agafar les característiques més interesants d'ambdues i fusionar-les com en el cas de la 3 o la 12. A més a més, aporta valor obrint la modificació dels nodes de flux.

5.2. Requisits del Projecte

Abans de entrar en detall i justificar les decisions preses per tal d'assolir l'objectiu del treball és menester parlar dels requisits del mateix. Degut a la natura del projecte, aquests s'han dividit en requisits tècnics, i acadèmics. Els requisits tècnics del projecte són els següents:

- L'entorn de desenvolupament és *Unity Engine.*
- El llenguatge de programació és C#.
- El temps per implementar l'eina és de dotze setmanes aproximadament.

Els requisits acadèmics del projecte són els següents:

- Totes les decisions preses han d'estar documentades.

5.3. Metodologia

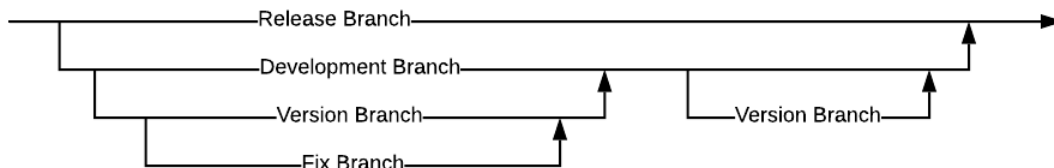
L'objectiu d'aquest apartat és definir el com es gestiona el desenvolupament a nivell metodològic. Es recullen aquí les decisions enteses com a línies mestres del projecte, més endavant, a la secció 6 es parla de les solucions implementades. Aquesta distinció es realitza en ares de poder documentar millor l'evolució de la eina, ja què s'entén que al ser un projecte de caire acadèmic es tan important el registre del procés com el resultat.

Com s'ha vist en l'apartat 5.2, el temps de desenvolupament és curt. En base a minimitzar el risc de no assolir tots els objectius finals s'utilitza un sistema incremental de producció basat en estratègies àgils similar al visualitzat en la Figura 3.1. Representació d'un Desenvolupament Incremental Tot i que les fites del projecte s'estableixen a la secció següent (5.4), és menester especificar que s'entén que la eina ha avançat una iteració complerta quan s'assoleix una versió estable de la mateixa amb les noves característiques incorporades. Cal destacar que si bé la implementació és iterativa es realitza un estudi previ del codi existent per tal que les noves *features* quedin ben integrades al sistema.

La següent decisió també està supeditada a la necessitat de minimitzar l'impacte temporal que pugui generar un error durant la producció. Durant la implementació de la eina s'utilitza *bitbucket,* un sistema de control de versions basat en *Git.* L'elecció d'un sistema que permeti diverses branques es pren en ares de poder deixar patent fins on arriba cada iteració. L'estructura que es segueix es la mostrada a la figura 5.1. on les branques s'usen per:

- *Release Branch*: Tan sols s'actualitza amb versions complertes i estables de l'eina.
- *Development Branch*: S'actualitza amb versions estables de l'eina.
- *Version Branch*: S'actualitza durant el desenvolupament de la versió, per tant, és la branca d'ús més comú. No es contempla dividir-la en branques de característiques a excepció de aquelles que siguin potencialment problemàtiques o que estiguin fora del pla de producció previst.
- *Fix Branch*: En cas de detectar un error greu, es genera una branca nova per arreglar-lo i així poder comprovar que no es perden característiques al solucionar-lo.

Figura 5.1. Estructura de Branques



Font: Elaboració Pròpia

En base al que mostra l'apartat 3.4, el llenguatge de programació emprat per desenvolupar amb *Unity Engine* és *C#.* S'escull aquest llenguatge degut a que es pretén una integració completa de l'eina dintre del editor. La solució s'implementa dintre del paradigma de la programació orientada a objectes per ta d'aprofitar al màxim les capacitats del llenguatge. Tot i que, com s'ha esmentat amb anterioritat, l'arquitectura de la solució final es veu a la secció 6.

5.4. Planificació

Figura 5.2. Calendari de Producció

Planificació	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Febrer																															
Març																															
Abril																															
Maig																															
Juny																															
Juliol																															

Font: Elaboració Pròpia

El cronograma de la figura 5.2 mostra la planificació establerta de la producció d'aquest treball. Seguint la llegenda de la figura 5.3, la concreció de les tasques a realitzar així com de la seva durada estimada es realitza del onze al quinze de Febrer.

Figura 5.3. Llegenda Cronograma

Festiu	
Entrega Oficial	
Entrega Esborrany	
Planificació	
Redacció Memòria	
Producció	
Elaboració Defensa	

Font: Elaboració Pròpia

Tal com la figura 5.2. contempla, es reajusta el cronograma en el període establert fins arribar a la figura 5.4. Cal esmentar que s'estableix un canvi de criteri a l'hora de considerar el que s'entén com una iteració complerta, i alhora es pondera el temps dedicat a les iteracions segons l'esforç que suposa la seva implementació.

Figura 5.4. Cronograma del Desenvolupament

Planificació	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Març																															
Abril																															
Maig																															
Juny																															
Juliol																															

Font: Elaboració Pròpia

- Primera iteració (4 Març – 12 d'Abril): Llibreria funcional que implementi BTs al motor.

Un cop finalitzat aquest període el projecte haurà arribat a la meitat del seu temps de producció. Del dos d'Abril al cinc del mateix mes es fa una revisió del èxit assolit i tasques pendents. Durant aquestes dates es reajusta la planificació per tal d'arribar amb el major nombre d'objectius complerts. Les següents iteracions es desglossen de la següent forma:

- Segona iteració (25 d'Abril– 22 de Maig): Eina funcional que implementa els objectius principals del treball polida i estable.
- Tercera iteració (28 Maig – 4 Juny): Eina final estable que implementa tots els objectius del treball.

Deixant els períodes següents per la redacció, correcció, i edició de la memòria del treball així com la seva defensa:

- 15 Abril – 23 Abril: Correcció de la memòria intermèdia.
- 23 Maig – 26 Juny: Correcció de la memòria final.
- 5 Juny – 10 Juny: Correcció de la memòria final.
- 14 Juny – 24 Juny: Elaboració de la defensa oral.

Si be el cronograma estipulat a la figura 5.4. es respecta quasi del tot, degut a un imprevist que s'explica a l'apartat 6.2.2.1 al final s'opta per eliminar la tercera iteració. Augmentant així el marge per a finalitzar la segona fins el 4 de Juny tal com es veu a la taula 5.5.

Taula 5.5. Dates destacades

Planificació	11 Febrer - 15 Febrer
Primera Iteració	4 Març-12 Abril
Revisió Planificació	2 Abril - 5 Abril
Correció Memòria	15 Abril - 23 Abril
Entrega Memòria Int	24-abr
Segona Iteració	25 Abril- 4 Juny
Correció Memòria	23 Maig - 26 Maig
Lliurament Esborrany	27 Maig
Correció Memòria	4 Juny -10 Juny
Entrega Memòria Final	11 Juny - 13 Juny
Elaboració Defensa Oral	14 Juny - 24 Juny
Defensa Oral	25 Juny - 11 Juliol

Font: Elaboració Pròpia

6. Desenvolupament del Treball

Un cop explicada la metodologia general emprada per la elaboració de la part pràctica d'aquest treball així com les principals fites de la mateixa a l'apartat anterior, aquesta secció es centra en el registre de la evolució del treball. El registre s'ha estructurat de forma que cada iteració té un anàlisi individual i finalment es realitza una valoració global dels resultats obtinguts durant el desenvolupament.

Els anàlisis comencen amb una petita introducció de quin es l'estat de la implementació abans d'iniciar la iteració. A continuació s'exposen els objectius del cicle distingint-los entre principal i secundari, i dividint-los segons si han estat assolits o no. Tot seguit es dedica una secció a detallar com s'han implementat les noves funcionalitats. Finalment es realitza una valoració del procés.

6.1. Primera Iteració

Stricto sensu l'estat del desenvolupament abans d'aquesta iteració és inexistent. Si bé és cert que no existeix res tangible abans de finalitzar aquest cicle, si que existeixen les línies mestres del desenvolupament. En base a tot el que s'ha establert fins al moment aquestes directrius són:

- Les iteracions conclouen en versions usables del producte.
- La implementació es realitza dintre de l'entorn de *Unity Engine* amb *C#.*
- Degut a la natura del entorn la solució general es realitza sota el paradigma de la programació orientada a objectes.

6.1.1. Objectius de la Iteració

Tal com es recull a l'apartat 5.4, la primera iteració es considera que ha conclòs un cop existeix una llibreria de codi usable que permet l'ús d'arbres de comportament al motor. La creació d'aquesta llibreria es desglossa en els següents objectius assolits:

Principals:

- Implementació dels nodes de control flux: *Selector, Sequence, Parallel, Decorator.*
- Implementació dels nodes "fulla": *Action, Condition.*
- Implementació de la classe *BehaviourTree.*
- Implementació d'un component *Monobehaviour* que executi l'arbre creat.

Secundaris:

- Implementació d'un arbre per testejar el correcte funcionament de la llibreria.

- Generar la documentació pertinent per l'ús de la llibreria.
- Emprar els sistemes d'atributs de C# per incrementar la usabilitat la llibreria.

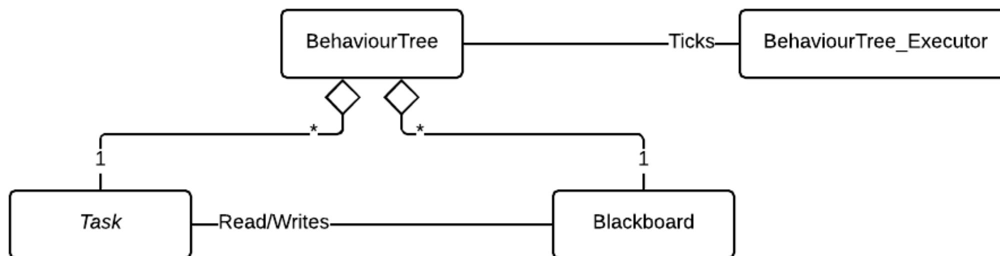
6.1.2. Desenvolupament de la Iteració

6.1.2.1. Implementació de la Llibreria

Tot i que se'n parla més endavant a la secció 6.1.2.2 la majoria del temps emprat en aquesta iteració ha estat invertit en dissenyar l'arquitectura de les classes. La estructura definida es planteja com el nucli de la solució final. Cal esmentar que tot i ser considerada final es susceptible de canvis. En cas de que les classes que es defineixen a continuació es modifiquin s'esmentarà a l'anàlisi de la iteració on s'hagin fet.

Emprant una aproximació del estil divideix i venceràs podem observar dos blocs d'acció: implementar els arbres de comportament, i implementar un script que els executi. Els arbres alhora estan compostos d'un conjunt de tasques encadenades, que culminen en un node arrel, i una pissarra. En base a aquesta divisió sorgeix la estructura representada a la figura 6.1.

Figura 6.1. UML-S Estructura de la Llibreria

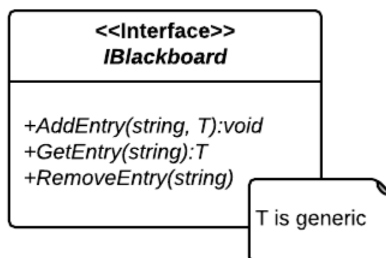


Font: Elaboració Pròpia

Tal com mostra la figura 6.1 la classe *BehaviourTree_Executor* és qui genera la senyal (*tick* d'ara endavant). Per això, te com a atributs les opcions d'execució del arbre (freqüència, repetició, pissarra amb la que executar l'arbre, etc.). La classe *BehaviourTree* en el fons no és res més que un embolcall per una tasca arrel i la pissarra que emprarà l'arbre.

La pissarra s'implementa com una interfície *IBlackboard* que declara les funcionalitats bàsiques per consultar, escriure, i eliminar informació. S'opta per construir la llibreria a arrel d'una interfície enlloc d'una classe concreta per tal de dotar de tota la llibertat possible a un futur usuari de la mateixa. Tanmateix s'entén que la pissarra no es concreta al llarg del treball (a nivell de llibreria) i per tant no pren sentit fer-la una classe abstracta ja què no tindrà camps.

Figura 6.2. UML IBlackboard

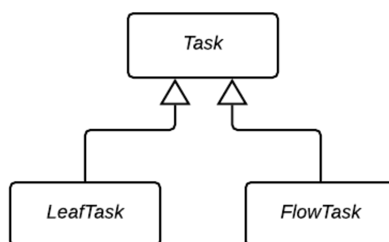


Font: Elaboració pròpia

El sistema de tasques, igual que la pissarra, es recolza en les facilitats que ofereix C# i la seva orientació a objectes. En aquest cas s'aprofita el polimorfisme explicat a l'apartat 3.4.2.1. de forma que es treballa a nivell conceptual amb *Task* però la implementació permet concretar el tipus i funcionament de cada tasca.

La implementació d'aquest sistema parteix d'una classe abstracta *Task,* que declara dos mètodes genèrics *Tick,* i *OnAbort*. Si bé es podria haver partit d'una interfície (ja que la classe, de moment, no té membres propis) s'ha optat per fer una classe en vistes d'implementar un sistema de depuració més endavant. D'aquest deriven la classe abstracta *LeafTask,* i la classe abstracta *FlowTask.*

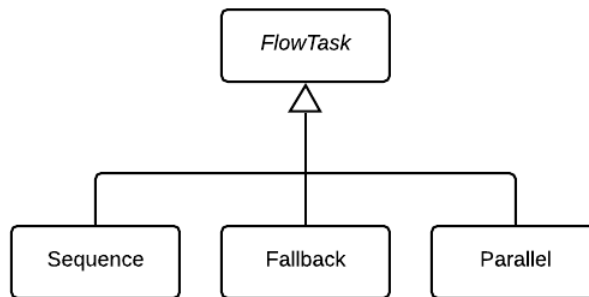
Figura 6.3. UMLS Sistema de Tasques Primera Herència



Font : Elaboració Pròpia

La diferència principal de la divisió que mostra la figura 6.3 radica en el nombre de sub-tasques o fills. Mentre que les classes que hereten de *LeafTask* no en tenen, les que hereten de *FlowTask* en tenen varis. Ambdues classes són abstractes i en cap cas implementen els mètodes heretats forçant així la implementació d'un tercer nivell de tasques que són les vistes en l'apartat 3.3.2.2. La figura 6.4 mostra com els nodes *Sequence, Fallback,* i *Parallel* hereten la classe abstracta *FlowTask* i implementen tots els mètodes necessaris per poder ser instanciades i emprades.

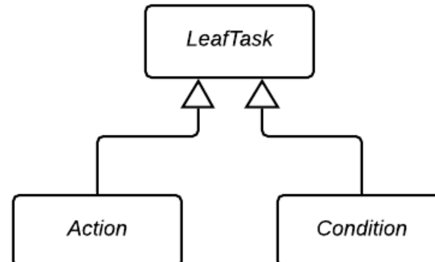
Figura 6.4. UMLS Nodes de Control de Flux



Font : Elaboració Pròpia

La figura 6.5 mostra com els nodes *Action* i *Condition* hereten la classe abstracta *LeafTask* però segueixen sent abstractes. A diferència del cas anterior aquestes classes implementen els mètodes heretats però declaren uns altres que els usuaris de la llibreria hauran d'implementar. Així s'eviten valors de retorn que no siguin vàlids en el cas de la condició, i alhora doten de més flexibilitat al programador en el cas de les accions.

Figura 6.5. UMLS LeafTask

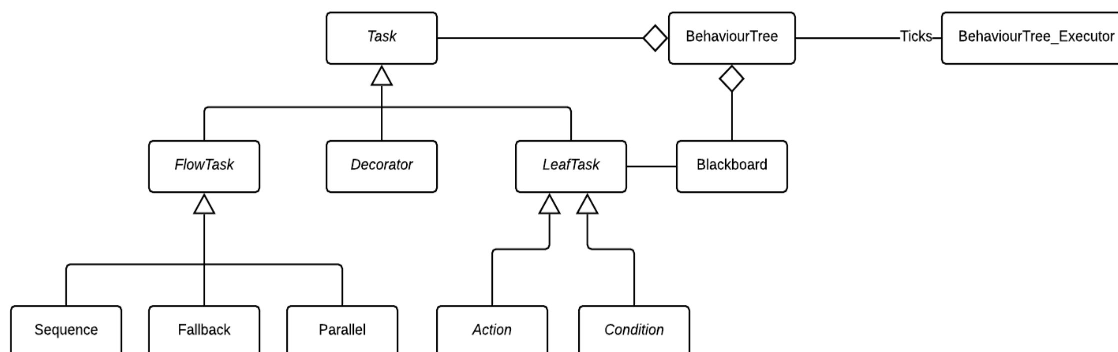


Font : Elaboració Pròpia

En el cas de *Condition* l'usuari final implementarà un mètode *Evaluate* que podrà retornar *true* o *false.* Aquesta funció es crida dintre del mètode *Tick* (ocult a l'herència) i s'interpreta el resultat a l'estat de la tasca que correspongui per retornar *Succes* o *Failure*. En el cas de *Action,* l'usuari ha d'implementar les funcions *InitTask, UpdateTask,* i *OnTaskEnd.* Com en el cas anterior, el mètode *Tick* crida aquestes funcions.

L'últim node per implementar és el *Decorator*. El cas de la classe *Decorator* és un híbrid entre els dos grups presentats fins al moment. Te un sol node fill sobre el que opera, i alhora és totalment modificable. Degut a això s'ha optat per fer una classe abstracta que hereta de *Task* i no implementa cap mètode més enllà del constructor. La figura 6.6 mostra el UML simplificat de la estructura vista fins al moment.

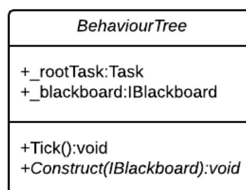
Figura 6.6. UMLS Llibreria de BTs



Font: Elaboració Pròpia

Finalment, la classe abstracta *BehaviourTree* (figura 6.7) és una agregació d'una *Task* i un objecte *IBlackboard.* Aquesta classe implementa un mètode virtual Tick que permet realitzar l'execució de l'arbre, i un defineix un mètode abstracte *Construct* que delega la definició dels nodes de l'arbre en consonància amb la pissarra (rebuda per paràmetre) que empra en temps d'execució.

Figura 6.7. UML BehaviouTree



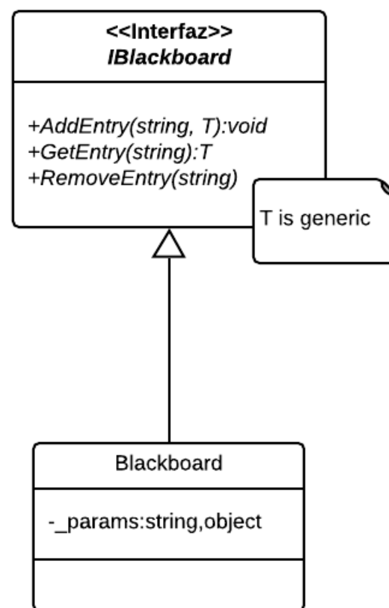
Font: Elaboració Pròpia

6.1.2.2. Implementació d'un BT de Test

L'objectiu principal d'aquest test és el de realitzar un cicle complert de l'ús de la llibreria. En altres paraules, consisteix en l'extensió de la mateixa per adaptar-la a un cas concret i la correcta execució de l'arbre generat.

El primer pas per a la creació d'un BT de test és concretar la *IBlackboard* en un tipus de dada concret. Tal com mostra la figura 6.8 la classe *Blackboard* és un contenidor d'informació que serà accessible des de les tasques que necessiten manipular-ne. Aquesta classe dota d'un embolcall a un diccionari *<string, object>*. Així qualsevol tipus de dada pot ser guardada, ja què totes les classes a C# deriven de la classe *System.object.* L'embolcall aporta els mètodes genèrics per consultar i manipular les dades al diccionari que declara la interfície *IBlackboard*.

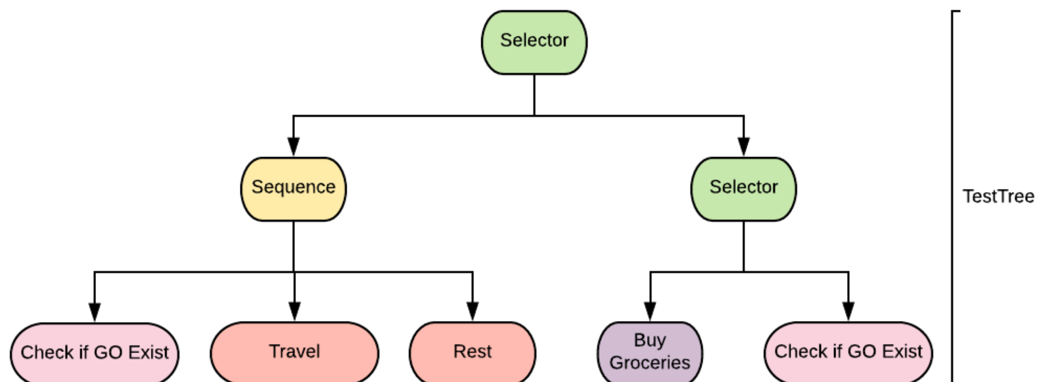
Figura 6.8. UML Blackboard



Font: Elaboració Pròpia

A continuació es defineix l’arbre de comportament que s’implementa durant el test. La figura 6.9 mostra de forma gràfica el comportament desitjat. La tasca *Check if GO Exist* és una concreció de la tasca *Condition,* i les tasques *Travel* i *Rest* són concrecions de la tasca *Action* vistes ambdues en l’apartat anterior. Aquest nou conjunt de tasques es parametritzen utilitzant una instància de la classe *Blackboard* i el seu funcionament consisteix en mostrar missatges per la consola de *Unity Engine.* D’aquesta forma es pot realitzar un seguiment de l’execució de l’arbre en temps real.

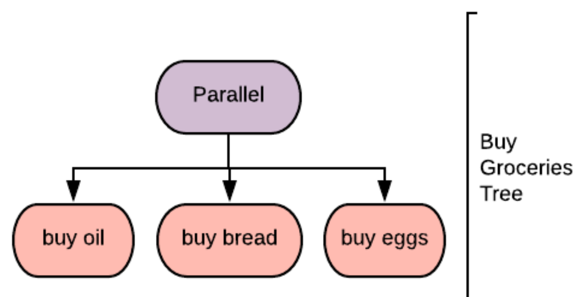
Figura 6.9. BT de Test



Font: Elaboració Pròpia

Cal destacar que la tasca *BuyGroceries* és en realitat un arbre de comportament que es concatena utilitzant la seva tasca arrel com a una de les tasques filles del selector. Aquest arbre (figura 6.10) consisteix en un node *Parallel* que executa diverses instàncies de la tasca *BuyItem* parametritzades de forma diferent. *BuyItem* és, de nou, una concreció de la tasca *Action*.

Figura 6.10. BT Buy Groceries



Font: Elaboració Pròpia

El test es realitza de forma satisfactòria, tot i que es realitzen certes modificacions temporals a l'script *BehaviourTree_Executor* per tal de poder realitzar el test dintre d'un espai temporal el més reduït possible. Aquestes modificacions consisteixen en la creació de una instància de la pissarra i la seva correcta parametrització dintre del mètode *Start* del script. Aquesta decisió és considera innòcua ja què en la propera iteració les pissarres es doten d'un editor gràfic, i per tant, aquestes modificacions són eliminades.

6.1.3. Valoració de la Iteració

La valoració d'aquest primer cicle consta de varis elements a destacar, però el més important és, sens dubte, la gran diferència entre el temps estimat d'implementació i el temps real que ha estat necessari. La durada estimada per a aquesta iteració era de dues setmanes (del 18 de Febrer al 1 de Març), la final però ha estat de aproximadament un mes i mig (del 4 de Març al 12 d'Abril).

Aquesta demora ha estat fruit de diversos factors, la majoria externs al treball. El principal, però, és la manca d'experiència a l'hora d'estimar les hores de producció del treball. L'altre gran factor és la falta d'estudi previ del disseny del software fet que va ocupar un terç del temps emprat. Si bé aquesta endarreriment és preocupant, un cop realitzat aquest cicle, el treball consta d'un nucli sòlid per a les properes iteracions.

El segon gran punt a destacar és l'assoliment del primer dels objectius del treball. Al finalitzar aquest cicle el treball consta d'una llibreria funcional que permet implementar BTs en la seva concepció clàssica. La llibreria és fàcilment ampliable per tal de generar nous tipus de nodes que implementin funcionalitats més properes a les vistes a la secció 3.3.3.

6.2. Segona Iteració

Tal com es detalla en l'apartat anterior, la implementació a l'inici d'aquesta iteració consisteix en una llibreria usable que permet la creació i execució d'arbres de comportament parametritzats de forma externa mitjançant pissarres. Tal i com es concreta a continuació, l'objectiu principal d'aquest cicle és dotar de un suport d'edició gràfica la llibreria desenvolupada.

6.2.1. Objectius de la Iteració

Els objectius de la segona iteració segons el que es recull a l'apartat 5.4, consisteixen en la creació d'una eina funcional, polida i estable que permeti la creació i edició de BTs dintre del motor. Aquest objectiu és desglossa en:

Principals:

- Assolits:
    - Guardar els arbres de comportament com un *asset* dintre del projecte.
    - Representar de forma gràfica els arbres generats emprant la llibreria.
    - Dotar la representació gràfica dels BTs de la capacitat d'edició dels mateixos.
    - Crear arbres de comportament de zero amb la eina generada.
    - Executar els arbres generats amb l'eina correctament.
    - Guardar les pissarres com un *asset* dintre del projecte.
- Pendents:

- Dotar la pissarra de representació gràfica.
- Dotar la representació gràfica de la pissarra de la capacitat d'edició.
- La correcta execució de tots els tipus d'arbre (gràfics o no) amb qualsevol dels tipus de pissarra (gràfica o no).

Secundaris:

- Fer l'eina el més usable possible.
- Fer l'eina el més estètica possible.

6.2.2. Desenvolupament de la Iteració

De nou, la forma d'aproximar aquesta iteració és la de dividir el codi tan com sigui possible (mentre tingui sentit fer-ho). Per començar es distingeixen dos grans blocs:

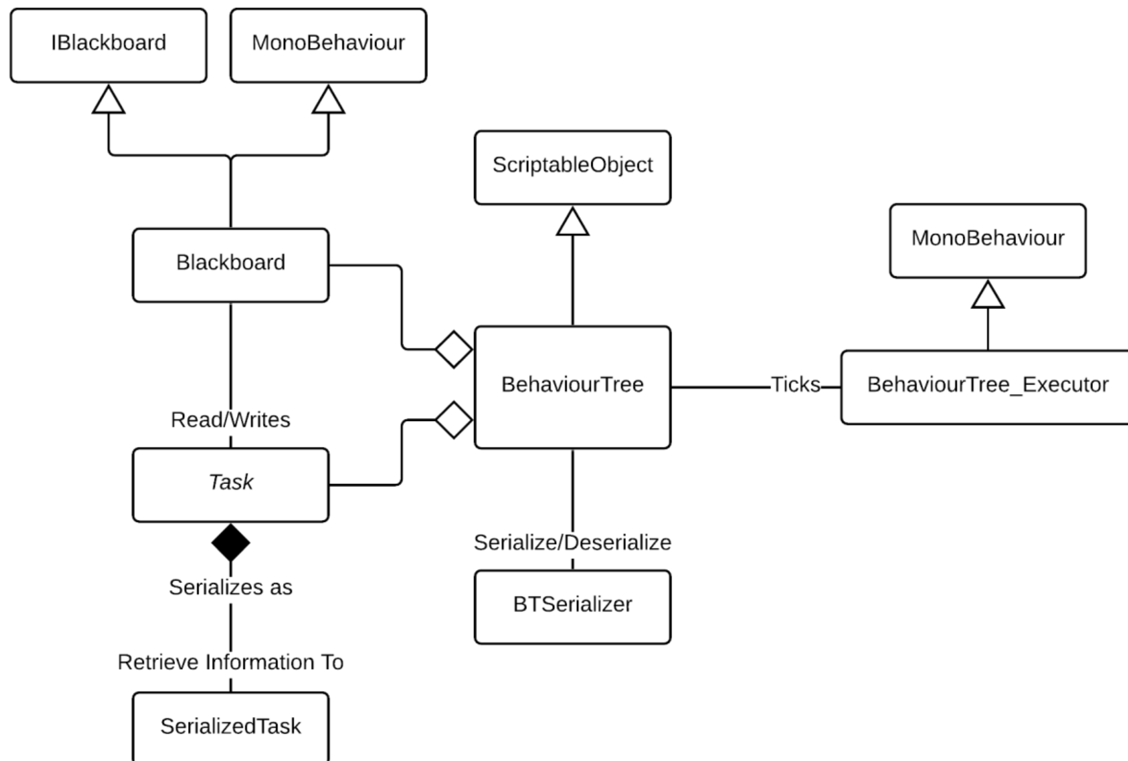
- *BTEngine.Core*: comprèn les classes mínimes per crear, modificar, parametritzar, executar, i guardar un arbre de comportament emprant la llibreria.
- *BTEngine.GraphicEditor*: comprèn totes les classes pròpies de l'editor gràfic.

En ares de millorar l'ús de l'eina aquests blocs defineixen dos espai de noms. Cal destacar que si bé el *BTEngine.Core* (*Core* d'ara endavant) és totalment independent del *BTEngine.GraphicEditor* (*GraphicEditor* d'ara endavant), aquest segon si que fa ús del primer.

6.2.2.1. Implementació del BTEngine.Core

La figura 6.11. mostra el UML simplificat del disseny del *Core*. Com es pot observar es construeix sobre la implementació realitzada de la figura 6.1, i s'afegeixen dues classes més, *SerializedTask,* i *BTSerializer.* Abans d'entrar en detall de com funcionen les classes afegides, cal destacar les modificacions fetes en les classes *Task, BehaviourTree* i *Blackboard.*

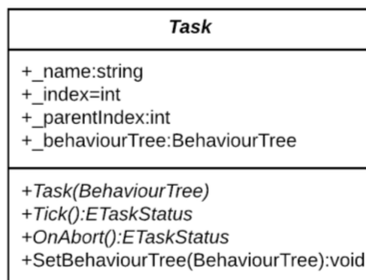
Figura 6.11. BTEngine.Core UMLS



Font: Elaboració Pròpia

La classe abstracta *Task* en l'anterior iteració no requeria de cap paràmetre per a poder instanciar classes derivades. En aquesta iteració però, s'implementa una referencia a un BT dintre de la classe, i es recomana instanciar les tasques passant-li per paràmetre l'arbre al que es vinculen. Aquesta decisió permet desvincular del tot les tasques de les pissarres, permeten que qualsevol tasca pugui accedir a la *Blackboard* del seu BT. Fins ara tan sols tenien accés a aquestes les tasques derivades de *FlowTask.* Tanmateix, com es pot observar en la figura 6.12, s'afegeixen diversos camps útils per diferenciar les diferents tasques a l'hora de representar-les i serialitzar-les.

Figura 6.12. Task UML



Font: Elaboració Pròpia

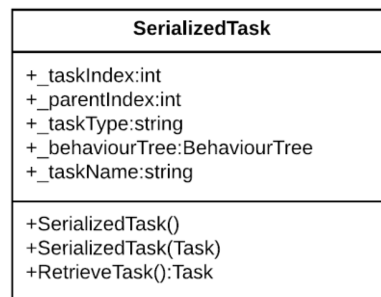
Tot i que l'impacte que ha causat el sistema de serialització del motor en el desenvolupament d'aquesta iteració es valora en l'apartat 6.2.3, és pertinent esmentar que ha estat el causant de la gran majoria de modificacions realitzades sobre el codi previ. Aquest sistema propi de *Unity Engine* tan sols reconeix com a camps serialitzables els que estiguin en classes derivades de *UnityEngine.Object.* És a dir, per tal de poder guardar les modificacions fetes sobre un objecte aquest ha d'estar integrats dintre de l'entorn de treball. A més a més, no permet serialitzar classes abstractes, referencies a interfícies, ni objectes niats. Tampoc entén el polimorfisme, és a dir, en cas de tenir una *Action* referenciada com una *Task, Unity Engine* la serialitza com a *Task.*

Cal destacar que, si bé el motor està dotat de mecanismes per a adaptar algunes de les problemàtiques que es deriven de les limitacions d'aquest sistema, es descarten totes en aquest treball. Es pren aquesta decisió ja què l'eina ha de ser flexible i ampliable pels usuaris, i aquests mecanismes podrien interferir directament amb els projectes on l'eina es susceptible d'integrar-se.

Un cop exposades les limitacions anteriors, les classes *BehaviourTree* i *Blackboard* es modifiquen per tal de fer-les heretar de *ScriptableObject* i *MonoBehaviour* respectivament. Aquesta decisió integra aquests dos objectes dintre de l'entorn de treball permeten així la seva serialització. La classe *BehaviourTree,* a més a més, afegeix un camp que consisteix en una llista de *SerializedTask.* De nou, degut a que el sistema de serialització no reconeix els objectes niats ni les classes abstractes s'opta per crear un camp nou que és pot serialitzar, enlloc de modificar el codi existent.

La decisió anterior porta a la necessitat de crear les dues classes que s'han afegit. La classe *SerializedTask* és l'equivalent de la classe *Task* que conté tota la informació de la tasca que s'ha de poder guardar així com tota la necessària per poder recuperar el tipus de la tasca i la seva posició dins l'arbre tal com mostra la figura 6.13.

Figura 6.13. UML SerializedTask



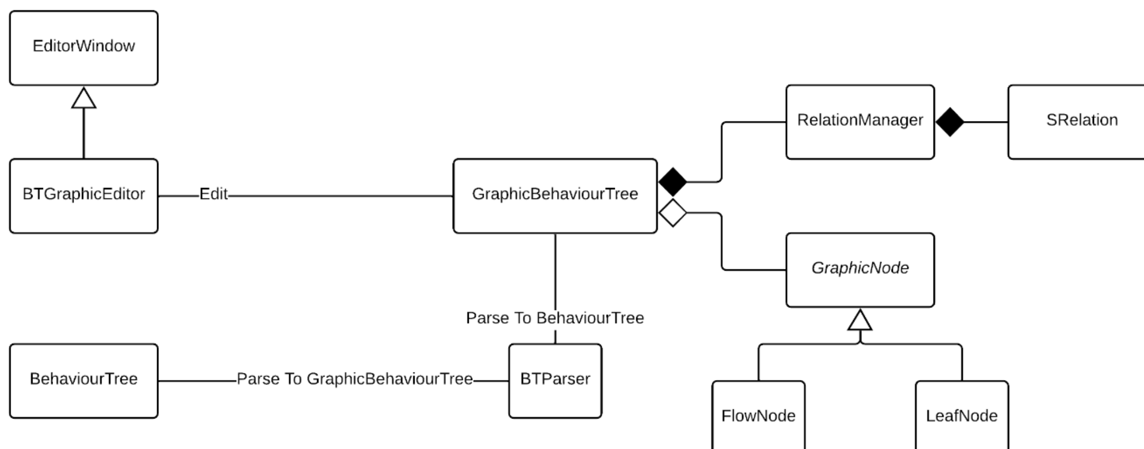
Font: Elaboració Pròpia

Per últim s'implementa la classe estàtica *BTSerializer* que s'encarrega d'iterar les tasques de l'arbre i deixar-lo preparat per a la seva serialització, així com de deserialitzar-lo. Com s'ha esmentat anteriorment *Unity Engine* proveeix una interfície per implementar aquestes funcions. No obstant, no es té el control de quan es criden aquestes funcions, per tant s'opta per desvincular aquest codi del propi arbre per tal de tenir el control de quan es porta a terme la serialització.

6.2.2.2. Implementació del BTEngine.GraphicEditor

La figura 6.14. mostra el UML simplificat del *GraphicEditor*. L'aproximació per a desenvolupar aquest codi és la de crear estructures de classes que siguin equivalents a les classes vistes al *Core*. Ara bé, aquestes noves classes implementaran la gestió de l'editor gràfic enlloc de les funcionalitats pròpies dels BTs. D'aquesta forma l'editor gràfic no opera mai sobre els objectes del *Core,* sinó què treballa en classes pròpies que es converteixen en objectes propis del *Core* un cop finalitzada la edició.

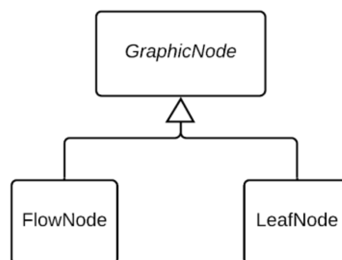
Figura 6.14. UMLS BTEngine.GraphicEditor



Font: Elaboració Pròpia

Tal com es pot apreciar a la figura 6.14, l'estructura central és la classe *GraphicBehaviourTree*, que en definitiva es l'equivalent editable del *BTEngine.Core.BehaviourTree.* Aquesta nova classe actua com un embolcall per a un objecte de la classe *RelationManager* i una llista de *GraphicNode* (l'equivalent a la classe *Task*).

Figura 6.15. UMLS Sistema de Nodes Gràfics



Font: Elaboració Pròpia

La figura 6.15 mostra l'esquema complert del sistema de nodes. A diferència del sistema de tasques (figura 6.6) en aquest cas tan sols es distingeix entre nodes de flux i nodes "fulla". Aquest fet es deu a que per a l'editor només cal distingir entre els nodes que poden tenir fills, i els que no.

La classe *GraphicNode* actua com a contenidor de la informació que té la tasca que representa. Alhora proveeix les opcions del menú contextual que apareixerà en fer clic dret sobre el node. Les dues classes derivades poden afegir contingut a aquest menú contextual així com controlar com es dibuixarà el node. Tots els nodes es dibuixen com petites finestres

d'editor, per tant, poden contenir informació i funcionalitats. Es decideix fer els nodes amb finestres per tal de facilitar el mostrar informació a l'usuari. Tot i així el gruix de dades es mostren al *Node Inspector* i al *Node's Blackboard Entries* del *GraphicEditor.* Per a que es mostrin els camps a l'editor, aquests han de ser públics i estar marcats amb l'atribut *Tune Parameter* o bé, amb l'atribut *Blackboard Entry* dintre de l'*script* de la tasca.

Els atributs *Tune Parameter* i *Blackboard Entry* formen part del sistema gràfic de pissarres que no s'arriba a implementar. En essència doten d'informació els camps que s'han de mostrar al inspector del node o de la pissarra respectivament. Es decideix mantenir la seva implementació tot i no finalitzar el sistema de pissarres gràfiques ja què el valor informatiu que aporten a l'usuari és substancial. Cal destacar que si bé la diferencia entre ambdós atributs és subtil pot resultar interessant:

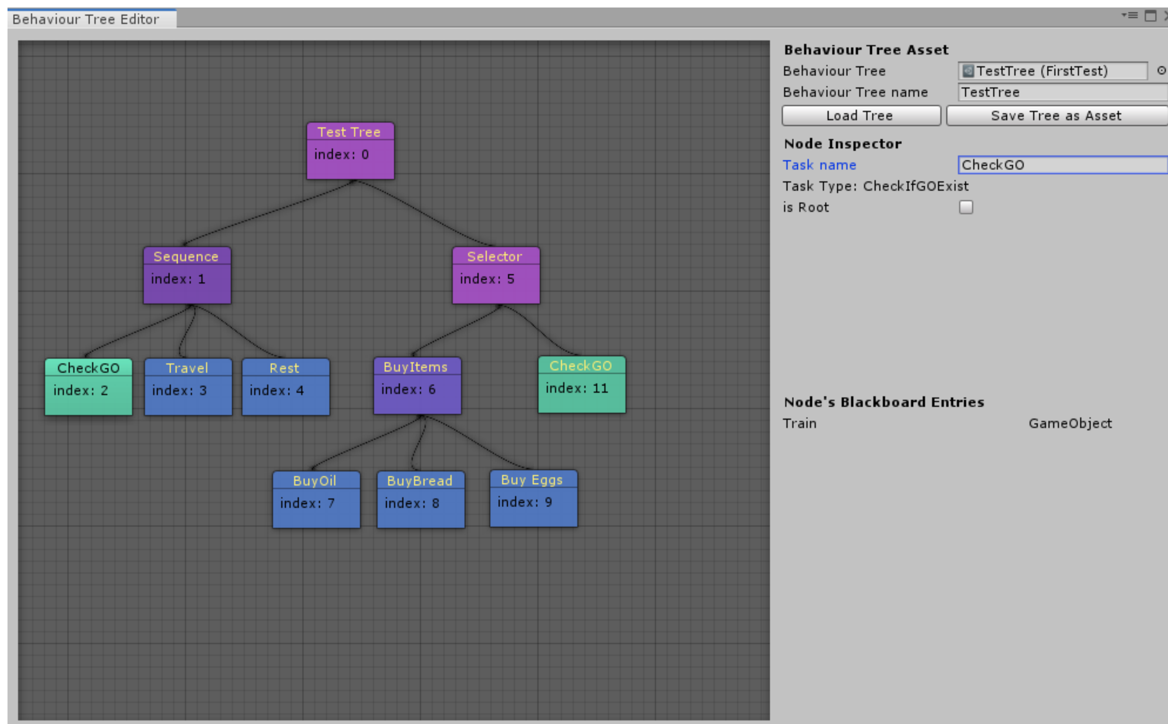
- *Tune Parameter:* s'aplica a aquells camps que permeten modificar un node respecte a diverses instàncies de si mateix. Els camps on s'aplica tan sols afecten al node i no referencien cap objecte.
- *Blackboard Entry:* s'aplica als camps que es consulten a la pissarra en temps d'execució. En altres paraules, emmagatzema referencies a objectes de l'escena, o informació compartida per diversos nodes.

La classe *RelationManager* és un embolcall que ajuda a gestionar una llista de *SRelation,* una *struct* composta per dos *GraphicNode.* Com el nom de la classe indica s'encarrega de gestionar totes les relacions entre nodes gràfics.

La classe *BTParser* es l'encarregada de convertir els objectes del *Core* en objectes del *GraphicEditor* i viceversa. En altres paraules és la classe encarregada de convertir els *BehaviourTree* en *GraphicBehaviourTree.* Com en el cas de la classe *BTSerializer* és estàtica.

Finalment, la classe BT*GraphicEditor* que hereta de *UnityEditor.EditorWindow* és qui integra tot el sistema dintre d'una finestra del editor. Com es pot observar a la figura 6.15, l'espai es divideix en un espai on crear nodes i editar l'arbre, i una franja dreta que actua a mode d'inspector. Aquesta última mostra una franja superior que permet carregar, canviar el nom, i guardar un BT. Per exemple, l'arbre representat a la figura 6.16 és el BT dissenyat com a test en la primera iteració (figura 6.9).

Figura 6.16. BTEngine



Font: Elaboració Pròpia

Aquesta classe, a banda de tot l'esmentat anteriorment, és qui fa ús de la reflexió per trobar i categoritzar totes les tasques creades per l'usuari per a instanciar-les com un node gràfic. Emprant mecanismes similars, cerca tots els camps públics del node seleccionat per mostrar quins paràmetres són ajustables (*Tune Parameter*) al node inspector (figura 6.16), i quins és consulten a la pissarra en temps d'execució (*Blackboard Entry*), mostrats a la secció *Node's Blackboard Entries* (figura 6.16). Cal destacar que tal com és veu a la figura 6.16 aquests camps no són editables al finalitzar aquesta iteració.

Cal destacar que la idea bàsica de com encarar el renderitzat de l'editor de nodes segueix les explicacions de *Sharp Accent* (2015). Així com, l'ús la llibreria *UnityEditor.Graphs* per la implementació de funcionalitats tal com el desplaçament vertical de l'editor de BTs, o la capacitat d'arrossegar la zona d'edició.

6.2.3. Valoració de la Iteració

La valoració d'aquest segon cicle consta de diversos punts a destacar. El més important però és l'impacte que ha tingut el sistema de serialització de *Unity Engine* sobre el *Core* i el temps d'implementació. A efectes pràctics ja s'ha vist a la secció 6.2.2.1 com han canviat les classes per adaptar-se a aquest sistema. El temps que ha costat trobar un bon sistema per integrar la serialització a l'eina ha estat de dues setmanes. Aquesta demora es deu en gran

part a que és descobreixen les mancances del sistema a mida que s'està implementant tota l'eina en paral·lel. Fet que va generar errors i força dificultats per aïllar el seu origen.

El segon gran punt a destacar és l'assoliment dels objectius que es refereixen a l'edició de BTs. En contraposició d'aquest fet, no s'arriben a assolir els objectius referents a les pissarres. Gran part d'aquesta mancança és fruit de la demora ocasionada pels fets descrits anteriorment.

Pel que fa als objectius secundaris, no es consideren assolits ja què l'eina sempre podrà ser més funcional i estar més polida a nivell estètic. Tot i així, el resultat final és força satisfactori visualment i no es excessivament incòmode.

7. Resultats del Treball

En aquest apartat es mostren els resultats obtinguts en el desenvolupament d'aquest treball. La secció es divideix en dos blocs. El primer mostra els resultats objectius, el segon consisteix en un resum de les valoracions de les iteracions, unes reflexions finals sobre com es podria haver millorat el treball, i planteja diversos escenaris que podrien donar continuïtat al treball.

7.1. Resultats Objectius

Taula 7.1. Resultats Finals

Funcionalitats Desitjades	
ID	**Funcionalitats**
1	Guardar els BTs com un asset dintre del projecte
2	Ús d'una pissarra per desvincular les tasques de la resta
3	Pissarra amb variables estàtiques
4	Nodes clàssics
5	Creació de Selectors propis
6	Creació de Sequences propies
7	Creació de Decorators propis
8	Creació de Actions propies
9	Creació de Conditions propies
10	Customització del node
11	Element extern que executa l'arbre
12	Els parametres d'execució del BT formen part del ent executor
13	Parametritzar la frequencia d'execució del arbre
14	Parametritzar les tasques màximes per Tick
15	Diferents colors segons el tipus de node en l'editor visual
16	Index del node visible en l'editor visual
17	Pila de crides
18	Visualització de la execució en temps real
19	Comentaris visibles en l'editor visual
20	Hinted BTs

Font: Elaboració Pròpia

La taula 7.1 mostra les funcionalitats que es desitjaven implementar en l'editor gràfic. Tots els camps que es ressalten en verd s'han implementat amb èxit dintre del temps esperat. Els ressaltats en vermell són objectius que s'han descartat definitivament degut a que el temps d'implementació i/o la seva complexitat tècnica fa impossible la seva implementació en el període establert per entregar el treball. Pel que fa als camps ocres, són funcionalitats que el sistema permet implementar de forma ràpida i és contempla la seva implementació de cara a la defensa oral del treball.

Pel que fa a l'eina final, a nivell de llibreria és totalment funcional i ampliable tal com es descriu a l'apartat 6. En quant a l'apartat gràfic permet la creació de BTs des de zero així com modificar arbres existents pel que es considera els objectius del treball assolits. Tot i així és cert que requereix una iteració més per acabar d'implementar totes les funcionalitats d'edició gràfica que s'esperen d'una eina de característiques similars.

7.2. Valoracions Finals

En primer lloc cal destacar els diversos errors comesos a l'hora d'establir el cronograma. El calendari establert a la figura 5.2 contemplava les iteracions com quelcom definit per un espai temporal. Si bé no és una mala aproximació per se, al disposar d'un marge tan reduït per a realitzar la implementació, hagués estat millor definir les iteracions com a versions finals del producte i ponderar el temps dedicat a cadascuna segons el esforç que suposa assolir-les. Tal i com es fa en el segon cronograma (figura 5.4). Seguint la mateixa línia de pensament, les estimacions haurien de cobrir un marge de temps dedicat exclusivament a compensar els imprevistos. En altres paraules, el pla de producció no hauria de necessitar tots els dies possibles. Si es així qualsevol demora suposa un retràs no negligible i que pot ser difícil de recuperar. En aquest cas, els dos cronogrames esmentats deixaven marges de temps per a poder recuperar aquests imprevistos, tot i així el segon cop el marge no ha estat suficient i no s'han pogut assolir tots els objectius desitjats.

En segon lloc, la producció durant la segona iteració va resultar més caòtica degut a dos factors. El desconeixement de les limitacions del sistema de serialització de *Unity Engine,* i el creixement en paral·lel de tot el projecte. Si bé aquest segon factor era ineludible, si que es podria haver plantejat de forma més cautelosa pensant en com depurar l'execució des de un inici.

Pel que fa als resultats, si bé és una eina que assoleix tots els objectius del treball, està lluny d'estar complerta. Gran part de les mancances que presenta venen de tres fonts, el desconeixement previ al treball sobre els sistemes de reflexió, així com de la llibreria de *Graphs* de *UnityEditor,* i el sistema de serialització. Amb els coneixements adquirits però produir una eina d'aquestes característiques (o replantejar la producció de la mateixa) hauria de resultar relativament senzill.

En vistes de la valoració dels resultats és menester parlar sobre les eines emprades. Tan *Unity Engine* com C# estan ben documentades i busquen que l'usuari tingui una entrada el més suau possible al seu ús. Inclús en opcions d'ús tan avançades com poden ser la reflexió o l'extensió de l'editor resulta senzill trobar informació. Si bé al llarg del treball s'ha esmentat de forma recorrent els problemes ocasionats pel sistema de serialització de *Unity Engine,* en general les limitacions que aquest té són per dotar l'usuari de més llibertat a l'hora d'implementar els seus sistemes i evitar els casos ambigus. La falta de control sobre quan s'activa el sistema si que es considera quelcom millorable.

De cara a la continuïtat acadèmica del treball, hi ha diverses línies prou interessants com per desenvolupar treballs emprant aquest com a base. Una d'elles és afegir la utilitat com a mecanisme de decisió dels arbres que produeix l'eina (com s'exposa a l'apartat 3.3.3.1), una altra possible via d'expansió seria la modificació dels BTs en temps d'execució (tal com s'explica en l'apartat 3.3.3.4) , o dotar els arbres de memòria (tal com s'introdueixen a la secció 3.3.2.3).

8. Referències

Buckland, M. (2005). *Programming Game AI by Example.* Plano, Texas, EUA: Wordware Publishing Inc.

Champanard, A. J. (2007). *Using a Static Blackboard to Store World Knowledge*. Recollit de AiGameDev.com: https://aigamedev.com/open/article/static-blackboard/

Colledanchise, M., & Ögren, P. (2018). *Behavior Trees in Robotics and AI.*

Epic Games. (2019). *Behaviour Trees*. Recollit de Unreal Engine 4 Documentation: https://docs.unrealengine.com/en-us/Engine/AI/BehaviorTrees

Flórex-Puga, G., Gómez-MArtín, M., Díaz-Agudo, B., & González-Calero, P. (2008). Dynamic Expansion of Behaviour Trees. *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Ghallab, M., Nau, D., & Traverso, P. (2016). *Automated Planning and Acting.* Cambridge: Cambridge University Press.

Isla, D. (2005). *GDC 2005 Proceeding: Handling Complexity in the Halo 2 AI*. Recollit de Gamasutra: http://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php

Marks, S., Windsor, J., & Burkhard, W. (2007). Evaluation of Game Engines for Simulated Surgical Training. *GRAPHITE '07* (p. 273-280). Perth, Australia: ACM.

Microsoft. (20 / 7 / 2015). *Introduction to the C# Language and the .NET Framework*. Recollit de C# Guide: https://docs.microsoft.com/es-es/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework

Millington, I., & Funge, J. (2009). *Artificial Intelligence for Games.* Burlington: Elsevier.

Ocio, S. (2012). Adapting AI Behaviours to Players in Driver San Francisco Hinted-Execution Behaviour Trees. *Proceedings, The Eight AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Padaone Games. (2018). *Behaviour Bricks Documentation*. Recollit de http://bb.padaonegames.com/doku.php

Rabin, S. (2002). *AI Game Programming Wisdom.* Hingham: Charles River Media.

Rabin, S. (2014). *Game AI Pro: Collected Wisdom of AI Game Professionals.* Boca Raton: CRC Press, Taylor & Francis Group.

SharpAccent. (3 / 3 / 2015). *Unity Tutorial Node Editors part 1 [Advanced].* Recollit de Youtube: https://youtu.be/gHTJmGGH92w

Sommerville, I. (2011). *Software Engineering.* EUA: Pearson Education INC.

Thompson, T. (3 / Ocutbre / 2016). *Behind The Systemic AI of Far Cry | AI and Games.* Recollit de Youtube: https://www.youtube.com/watch?v=Q7of5BPmiUs

Thompson, T. (15 / Agost / 2016). *The AI of Alien: Isolation | AI and Games.* Recollit de Youtube: https://www.youtube.com/watch?v=Nt1XmiDwxhY&t

Thompson, T. (2017). *Outsmarting The Covenant: The AI of Halo 2*. Recollit de Cube: https://medium.com/the-cube/theaiofhalo2-33e824209a4c

Thompson, T. (14 / Agost / 2017). *The AI of BioShock Infinite's Elizabeth | AI and Games.* Recollit de Youtube: https://www.youtube.com/watch?v=Yht4Oojfuvs

Thompson, T. (6 / Setembre / 2018). *Building Companion AI in Far Cry Primal | AI and Games.* Recollit de Youtube: https://www.youtube.com/watch?v=oNE1eyqeJ0c

Thompson, T. (2 / Gener / 2019). *Behaviour Trees: The Cornerstone of Modern Game AI (AI 101) | AI and Games.* Recollit de Youtube: https://www.youtube.com/watch?v=6VBCXvfNlCM

Unity. (2019). *Extending the Editor*. Recollit de Unity Documentation: https://docs.unity3d.com/Manual/ExtendingTheEditor.html

Unity. (2019). *The world's leading real-time creation platform*. Recollit de Unity: https://unity3d.com/es/unity

Unity. (2019). *Unity Documentation*. Recollit de Unity: https://docs.unity3d.com/Manual/index.html

Wang, P. (2008). What Do You Mean by "AI"? *Artificial General Intelligence* (p. 362-372). IOS Press.

Grau en Disseny i Producció de Videojocs

Annex

Ricard Perea Ros

Tutor: Dr. Enric Sesa i Nogueras

2018/2019

1.1. Diagrames UML

Ruta CD: Annexos\Diagrames\UML_BTEngine.Core.pdf

Ruta CD: Annexos\Diagrames\UML_BTEngine.GraphicEditor.pdf

1.2. Eina Implementada

Ruta CD: BTEngine\BTEngine.unitypackage

1.3. Codi Font

Ruta CD: Annexos\BTEngine.rar

1.4. Instruccions d'Ús de l'Eina

Ruta CD: Annexos\Documentació\Instruccions_BTEngine.pdf