TecnoCampus
Escola Superior
Politècnica

Centre adscrit a la
upf. Universitat Pompeu Fabra Barcelona

Grau en Enginyeria Informàtica de Gestió i Sistemes d'Informació

Desenvolupament d'un Framework per programar Utility-Based AI

Memòria

Jordi Romagosa i Mellado

Tutor: Enric Sesa i Nogueras

Curs 2017- 2018

TecnoCampus
Mataró-Maresme

Agraïments

Agraeixo al meu tutor, Enric Sesa, els consells i suport que ha aportat durant el desenvolupament d'aquest.

Abstract

The goal of this final-year project is the creation of a framework for programming Utility AI, easing the design and implementation of this kind of AI for videogame developers. During the development, the framework has been designed and implemented. A simple simulation has been developed with the objective of testing and evaluating the framework. The simulation has also been used for proposing new features and changes, which can be used in future framework implementations. In the end, the goal has been reached.

Resum

Aquest treball de final de grau té com a objectiu la creació d'un framework de programació de Utility AI que faciliti el disseny i la implementació d'aquest tipus de AI als programadors de videojocs. Durant el desenvolupament, s'ha dissenyat i implementat el framework; i després s'ha desenvolupat una simulació senzilla amb l'objectiu de testejar i avaluar el framework. També s'han detectat diverses possibles millores i modificacions durant el testeig, que poden servir com a base per seguir amb el desenvolupament del framework en el futur. Al final del projecte s'ha assolit l'objectiu de forma satisfactòria.

Resumen

Este trabajo de fin de grado tiene como objetivo la creación de un framework de programación de Utility AI que facilite el diseño y la implementación de este tipo de AI a los programadores de videojuegos. En el desarrollo se ha diseñado e implementado el framework; y después se ha testeado y evaluado usando una pequeña simulación. También se han detectado posibles mejoras y modificaciones para poder mejorar el framework en fases de desarrollo posteriores. Al final del proyecto, se ha conseguido cumplir el objetivo.

Índex

Índex de figures

Índex de taules

Glossari de termes

Agent	Es diu de qualsevol element d'un videojoc que estigui controlat per una AI.
AI	Intel·ligència Artificial, de l'anglès Artificial Intelligence
BT	Arbre de comportament, de l'anglès Behavior Tree
FPS	Videojoc de dispars en primera persona, de l'anglès First-Person Shooter. Es diu del gènere de jocs d'acció basats en combat amb armes de foc amb la càmera en primera persona.
FSM	Màquina d'estats finits, de l'anglès Finite State Machine
IDE	Entorn de desenvolupament integrat, de l'anglès Integrated Development Environment.
Indie	Prové de l'anglès Independent video game. Es diu dels jocs que s'han creat sense el suport financer d'un publicador, i acostumen a tenir el focus en la innovació.
Joc de rol	Es diu del gènere de jocs en el què s'assumeix el rol d'un personatge que es va fent més fort, adquirint noves habilitats i/o objectes, quan s'avança en el joc. Normalment va acompanyat d'un sistema de nivells.
MMORPG	De l'anglès Massive Multiplayer Online Role Playing Game. Es diu del gènere de jocs de rol que a més permeten que un gran número de jugadors interactuïn entre ells.
NPC	Personatge no jugador, de l'anglès Non-Player Character. Es diu d'un personatge, normalment d'un videojoc, que no és controlat per cap jugador.
UAI	Intel·ligència Artificial basada en el concepte de utilitat, de l'anglès Utility AI.

1 Introducció

Quan es dissenya o s'implementa una AI per a un personatge d'un videojoc, s'ha de determinar la forma en la qual el personatge decidirà quina és la següent acció que portarà a terme. Dels diferents mètodes de presa de decisions que s'utilitzen en els videojocs, un dels més nous i menys utilitzats, tot i que cada vegada s'utilitza més, és la Utility AI.

La Utility AI no utilitza la forma de decisió més típica basada en condicions, sinó que calcula la utilitat de cada acció i decideix quina és millor per portar a terme. Aquest càlcul es fa en funció de l'estat actual del personatge i de l'entorn, cosa que permet una alta flexibilitat en el comportament del personatge respecte a la decisió per condicions.

Però hi ha molts desenvolupadors de videojocs que no implementen UAI pel desconeixement de la seva existència, o perquè no saben com implementar-la. Així doncs, s'ha decidit que l'**objectiu** d'aquest treball final de grau és **implementar un framework de programació de UAI** per facilitar el disseny i implementació d'aquesta durant el desenvolupament de videojocs.

El primer pas per aconseguir l'objectiu ha estat dissenyar i implementar el framework. S'han redactat i avaluat els requeriments, i s'ha procedit a especificar el funcionament i les classes del framework tenint en compte els requeriments. Finalment, el framework s'ha codificat amb el llenguatge de programació C#.

Durant aquest procés s'ha tingut en especial consideració la completa transparència del framework respecte al videojoc: que no es modifiqui res de l'estat del joc i que tota la informació que necessita el framework s'obtingui directament del joc. Per tal d'assolir-ho, ha estat necessària la definició d'una interfície que el desenvolupador del videojoc haurà d'implementar, anomenada Context.

També s'ha considerat la necessitat de que el framework porti implementades funcions de càlcul d'utilitat, per facilitar-ne l'ús, en cas que el desenvolupador no sàpiga com programar les funcions de càlcul o senzillament es vulgui estalviar el procés. Al final, s'han implementat les funcions de càlcul que s'han considerat bàsiques i que permeten programar als usuaris una gran varietat de funcions de càlcul diferents. Els programadors també disposen de la possibilitat d'implementar les seves pròpies funcions d'utilitat.

S’ha redactat també una guia per facilitar la utilització d’aquest framework, inclosa en aquest mateix document.

El segon pas ha estat dissenyar i implementar una simulació utilitzant el framework per tal de comprovar si s’ha assolit l’objectiu del treball; i aprofitar per fer-ne una valoració de la usabilitat, i detectar millores i problemes del framework. La simulació s’ha realitzat amb la plataforma de videojocs Unity, i els scripts s’han escrit en el llenguatge de programació C#. El codi del framework s’ha incorporat com a .dll.

En aquesta simulació, s’ha implementat la AI d’un personatge, on l’objectiu del personatge és sobreviure el màxim de temps, decidint quina és la següent acció que realitzarà a continuació quan acaba l’anterior. Cada acció del framework correspon a una de les accions del personatge.

Una vegada definit el funcionament de la simulació i dissenyada la UI d’aquesta, s’ha procedit a dissenyar la AI del personatge utilitzant el framework. S’han definit sis accions diferents, una amb cadascuna de les sis funcions d’utilitat implementades en el framework.

La implementació de la interfície Context s’ha fet tenint en compte el funcionament de la simulació. Per facilitar la comunicació amb el framework i la lògica interna, és la mateixa classe que controla l’estat del personatge i, per tant, és la classe que conté la informació que demana el framework durant el càlcul de la utilitat.

Durant el disseny de la simulació s’ha considerat tot el necessari per testejar el funcionament bàsic del framework, així com tots els aspectes per poder fer l’avaluació dels objectius d’aquest. Gràcies a la simulació s’ha pogut confirmar l’assoliment de tots els objectius menys un, que no s’ha arribat a comprovar si es complia.

La conclusió a la que s’ha arribat després de testejar el framework amb la simulació és que s’ha assolit l’objectiu principal del treball. Tot i això, després de les proves s’han detectat diverses millores i canvis per millorar el framework, des del punt de vista d’un desenvolupador de videojocs que utilitzi el framework. Aquestes millores i canvis s’haurien de considerar en cas de seguir treballant amb el framework.

2 Objecte del projecte

Quan es valora un videojoc, el factor més important acostuma a ser el que s'anomena jugabilitat. Tot i que és un terme bastant abstracte, la jugabilitat inclou l'experiència de l'usuari al jugar el joc: la satisfacció, l'aprenentatge, l'efectivitat, la immersió, la motivació, l'emoció i la socialització. Una bona definició és: *"El conjunto de propiedades que describen la experiencia del jugador ante un sistema de juego determinado, cuyo principal objetivo es divertir y entretener de forma satisfactoria y creíble ya sea solo o en compañía de otros jugadores". [1]*

Una gran part de la jugabilitat ve donada per les interaccions amb els elements que no controlen altres jugadors. El fet que aquests elements es comportin de manera realista i fins i tot intel·ligent, controlats per AI (Artificial Intelligence), és un factor determinant per la jugabilitat perquè dona al jugador la sensació de que el que està passant és real.

Per posar un dels exemples més clàssics: en un joc de rol, un jugador es troba amb un monstre. El jugador segur que espera que el monstre reaccioni d'alguna manera a la seva presència: ja sigui atacant, intentant escapar, demanant ajuda, o fins i tot lluitant esquivant els seus atacs. Però si enlloc d'això el monstre està completament quiet, sense fer res, mentre el jugador l'ataca, probablement el jugador deixi de jugar al videojoc perquè "la jugabilitat és horrible".

També passaria el mateix en el cas contrari. El cas en el qual el monstre es converteix en una espècie de malson, quasi impossible de derrotar, perquè és capaç de predir tots els moviments del jugador. L'ideal és trobar un punt entremig, en el qual el jugador estigui a gust, per tal de maximitzar la jugabilitat; i per aconseguir-ho s'ha d'aplicar una AI al monstre.

Tenint en compte tot el que s'ha dit a dalt, podem afirmar que, en la creació de videojocs, un dels factors més importants a tenir en compte és com es comporten els NPC (Non-Player Character), enemics, bots, etc. És a dir, com actuaran tots aquells elements que han de tenir un comportament que es pugui descriure com a intel·ligent, però que no hi hagi cap persona al darrere que el controli. I per aconseguir que els diferents elements tinguin un comportament intel·ligent, es necessita una AI que controli cadascun d'aquests.

Ja fa molts anys que s’aplica la AI en els videojocs, però segueix sent un dels elements més difícils d’aplicar per diversos motius. Un dels motius més importants és degut al fet que una AI que faci massa càlculs pot arribar a absorbir tot el temps de procés del joc, en especial si hi ha molts elements amb AI que requereixin un càlcul simultani. I un altre, també molt important, és que és fàcil sobre-dissenyar el seu comportament, dedicant una quantitat de temps absurda de programació i aconseguint resultats bastant pobres quan la AI està en funcionament.

Considerant el que s’ha descrit de la situació actual dels videojocs i la importància de la AI en aquests, i també dels problemes que presenta una mala gestió de la AI, és important trobar una manera de facilitar la feina als programadors de AI per videojocs.

L’objecte del projecte és **crear un framework de programació que faciliti la programació de la AI dels videojocs, basat en els principis de la Utility AI per fer els càlculs i les decisions**.

El motiu pel qual s’ha triat la Utility AI respecte a altres tipus de AI per videojocs i el funcionament d’aquesta s’explicaran més endavant en el document.

3 Estudi previ

3.1 Tendències principals

La AI aplicada als videojocs engloba molts aspectes diferents, des d'algorismes bàsics que controlen el moviment fins a algorismes que permeten fer tàctiques coordinades entre diferents agents. Però com que l'objecte del projecte està centrat en la presa de decisions de l'agent, l'explicació es centrarà en els mètodes o algorismes de presa de decisions més utilitzats.

3.1.1 Màquines d'estats

La tècnica de presa de decisions més coneguda, més utilitzada i més senzilla és una FSM o Finite State Machine. Com el seu nom indica, una FSM està formada per estats que indiquen el comportament que ha de tenir l'agent; així com de transicions que permeten a l'agent passar d'un estat a un altre.

Com es pot veure en la Figura 3.1, l'agent té 4 possibles estats: **find aid**, **wander**, **evade** i **attack**. I cada estat té una o més accions que li permeten fer una transició cap a un altre estat. Suposant que el personatge comença a l'estat de **wander**, per tal de recórrer els quatre estats primer s'han de complir les següents condicions en aquest ordre: que el **player is near**, després que **player is attacking back** i després que **healthpoints are low**.

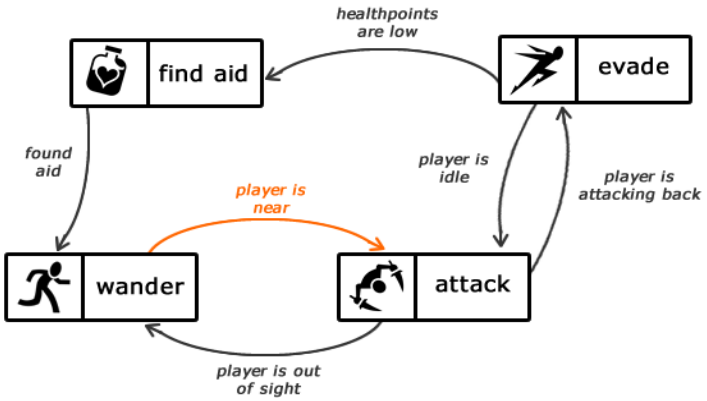


Figura 3.1 Exemple d'una Finite State Machine. Font: Extret de [2].

Una FSM és bastant fàcil d'implementar perquè és molt comú que els programadors ja hagin treballat amb màquines d'estats anteriorment. O en el cas que no sigui així, hi ha moltes guies per internet que indiquen pas per pas com programar una FSM per videojocs. A més, és molt fàcil gestionar el comportament de l'agent perquè en tot moment està en un estat prèviament definit, de manera que a l'hora de l'execució és impossible que faci alguna cosa imprevista. En l'exemple de la figura 1, és impossible que l'agent passi a l'estat **find aid** si no està prèviament a l'estat **evade** i es compleix la condició **healthpoints are low**.

El limitat número d'estats i de transicions fan d'una FSM una bona opció per totes aquelles AI que siguin bastant senzilles. Fins i tot grans jocs, si no necessiten una gran complexitat de les AI dels personatges, prefereixen aplicar una FSM per estalviar temps i esforços de programació. I la majoria de jocs Indie també utilitzen una FSM pels mateixos motius, a més del fet que és més fàcil d'aprendre a programar respecte als altres estils.

Però de la mateixa manera que una FSM permet programar AI senzilles de forma molt ràpida, també limita molt ràpidament la mida que aquestes poden arribar a tenir. Això és degut a què cada acció que pot fer l'agent ve donada per un estat diferent. Per tant, si es vol aconseguir que l'agent porti a terme una acció més, s'ha d'afegir un nou estat a la FSM. Però afegir un nou estat implica revisar totes les transicions de tots els estats per assegurar que el comportament de la FSM segueix sent el desitjat. El fet de revisar tots els estats s'acaba convertint en una pèrdua de temps molt gran, i com més estats hi ha més augmenta el temps dedicat només a revisar.

I un altre problema molt gran de les FSM és el fet que limita tant les transicions entre estats que moltes vegades no permet assolir la sensació de realitat que molts jugadors esperen dels enemics d'un videojoc. Aquesta falta de realitat moltes vegades es tradueix en avorriment dels jugadors, ja que veuen que és igual el que facin: l'enemic farà sempre el mateix. Seguint amb l'exemple de la Figura 3.1, l'enemic sempre sortirà corrents a buscar una poció de curació o similar quan tingui poca vida; sense importar la distància que hi pugui haver, o si amb un únic atac més és possible que derroti al jugador.

Per a més informació sobre les FSM es poden consultar les referències [2], [3] i [4].

3.1.2 Arbre de comportament

En un nivell més de complexitat que la FSM hi ha el Behavior Tree (Arbre de Comportament). El BT està format per un seguit de nodes de diversos tipus, enllaçats en forma d'arbre. Els enllaços entre els nodes són els que determinaran el comportament de l'agent una vegada estigui la AI en funcionament.

En la Figura 3.2 es pot veure un exemple de BT, en el qual un agent intenta travessar una porta. En aquest exemple es poden veure tres tipus de nodes diferents, que són els més bàsics del BT: els nodes **sequence** (tipus **Composite**), que fan que s'executin tots els nodes del nivell inferior; els nodes **selector** (tipus **Decorator**), que prenen decisió i executen només un dels nodes fills o canvien l'estat dels fills; i els nodes d'**accions** (tipus **Leaf**), que són els que diuen a l'agent el que ha de fer.

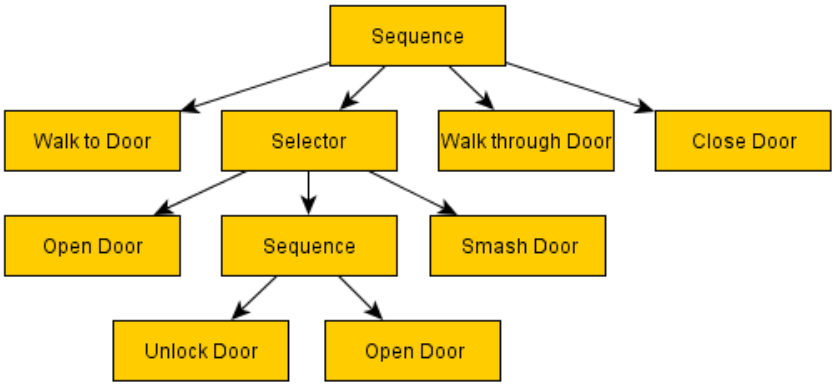


Figura 3.2 Exemple d'un Behavior Tree. Font: Extret de [5].

El principal avantatge del BT respecte la FSM i el motiu pel qual es va començar a utilitzar és que soluciona el problema d'afegir nous estats; eliminant la gran pèrdua de temps i l'augment de complexitat que això suposa en la FSM. Per afegir una nova acció al BT, només cal afegir un nou node d'acció o **Leaf** a l'arbre. A vegades també caldrà afegir algun node **Composite** o **Decorator**, però tota la resta de l'arbre resta inalterada i el comportament de l'agent es manté similar a l'anterior.

Però tot i que el manteniment del BT és molt més senzill que en la FSM, la implementació és molt més complexa i requereix d'un nivell avançat de programació. Per poder implementar un BT, el programador necessita conèixer com funciona una estructura d'arbre i com implementar-la. Això implica que el programador ha de tenir experiència prèvia amb arbres o bé que ha de dedicar un gran esforç en aprendre'n just abans de programar el BT. I si a més tenim en compte que les guies que es poden trobar per internet són menys abundants i més complexes que les de FSM (normalment expliquen només la teoria perquè assumeixen que el programador ja sap programar arbres), és fàcil entendre que només jocs que necessitin una AI avançada o complexa i jocs desenvolupats per companyies experimentades apliquin el BT (fet que descarta quasi del tot els jocs Indie).

El BT també té un altre problema, en comú amb la FSM, que és la poca flexibilitat de l'agent a l'hora de reaccionar al context. En el mateix exemple de la figura 2, una vegada l'agent hagi començat la seqüència de **walk to door** fins a **close door**, és segur que ignorarà qualsevol cosa que hi hagi al seu voltant o que pugui passar fins que acabi la seqüència. I de la mateixa manera que la FSM, les condicions són tan fixes que es pot predir amb facilitat el que farà l'agent a continuació, només cal conèixer el comportament habitual d'aquest per saber què farà a continuació.

3.1.3 Utility AI

Precisament per solucionar aquest problema que tenen en comú el BT i la FSM es va començar a utilitzar el que anomenen Utility AI (UAI). Com el seu nom indica, està basada en la utilitat de les accions enlloc de en estats o seqüències de tries com la FSM o el BT, respectivament. Així doncs, cada vegada que l'agent necessita saber el que ha de fer (quina acció ha de portar a terme), la UAI calcula quina és la millor acció basant-se en el context actual de l'agent.

Els càlculs es fan utilitzant el que s'anomenen funcions d'utilitat. Aquestes funcions agafen els valors del context de l'agent (els paràmetres d'entrada), i utilitzant funcions matemàtiques que contemplen valors entre un màxim i un mínim prèviament establerts, calculen un valor de sortida entre 0 i 1. Aquest valor és el que s'anomena "utilitat" de l'acció i es compara amb la utilitat de les altres accions per triar quina és la millor acció que pot portar a terme l'agent.

En la Figura 3.3 es pot veure un exemple de tres funcions d'utilitat diferents que, en un joc FPS (First-Person Shooter), controlen el comportament d'un NPC. Cada acció té assignada una funció diferent, que depenent dels valors del context retornarà un valor entre 0 i 1 tal com està marcat a l'eix y del gràfic. A l'eix x hi ha representat el **score** que representa el valor que poden tenir els camps d'entrada de la funció d'utilitat, ja que no es pot posar un únic valor com **distància a l'enemic** o **salut** perquè cada funció d'utilitat pot tenir diversos valors d'entrada, i poden ser diferents entre ells.

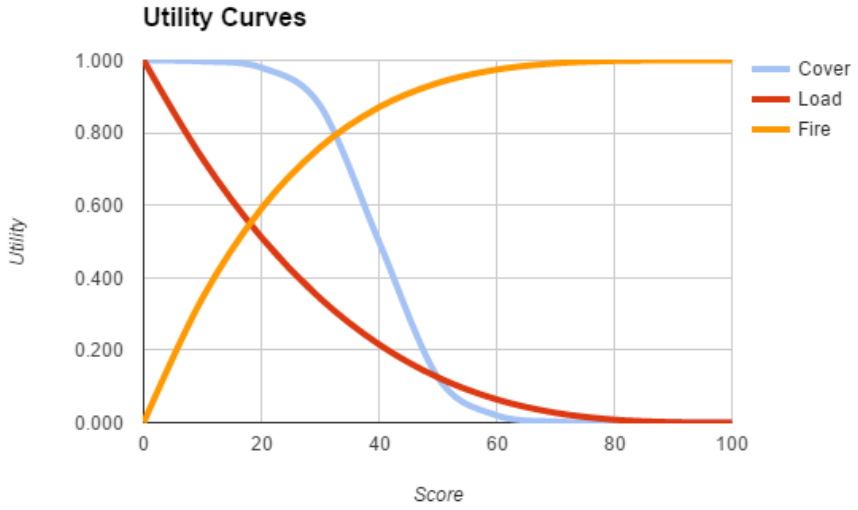


Figura 3.3 Exemple de tres funcions d'utilitat. Font: Extret de [6].

El principal avantatge de la UAI és que permet que l'agent reaccioni de forma dinàmica a l'entorn, creant amb facilitat la sensació de què es comporta de forma intel·ligent i realista. Posant el mateix exemple dels jocs de rol que s'ha fet servir en la FSM, un enemic amb menys de la meitat de vida segurament decideixi no atacar a un jugador si hi ha alguna manera que es pugui curar abans. I també podria reaccionar diferent si enlloc d'un jugador es troba amb un grup de quatre o cinc, fugint d'ells enlloc d'atacar.

Un altre avantatge de la UAI és que és bastant fàcil d'implementar si es té coneixement de com programar les funcions matemàtiques que en formen la base. Al contrari que la FSM i el BT, no s'ha de planificar el comportament des del principi perquè és el mateix agent el que decidirà el que ha de fer en funció del context. A més, afegir o treure una acció és molt senzill perquè cada acció és completament independent de les altres. L'únic que s'ha de tenir en compte és que l'agent ha de tenir els valors d'entrada de les funcions en el context.

Però com que la UAI permet que l'agent decideixi el seu comportament sobre la marxa, implica que la UAI s'ha de testejar i ajustar abans de donar-la per acabada. I aquí és on es pot arribar a perdre molt de temps perquè l'única manera d'ajustar correctament les funcions de càlcul d'utilitat és veient com reacciona l'agent en relació al context. Posant un exemple d'un joc de simulació, unes funcions mal ajustades podrien fer que l'agent, que té molta gana i s'està avorrint molt, decideixi mirar la tele enlloc de menjar i això provoqui que s'acabi morint de gana.

La UAI no fa massa que s'utilitza en els videojocs, però ha anat augmentant el seu ús ràpidament degut a la flexibilitat que dona als agents que l'implementen una vegada el joc està en funcionament. És molt utilitzada en jocs de l'estil Simulador, on és molt important que els personatges es comportin de forma realista. I també hi ha grans companyies que han dedicat esforços en implementar la UAI en els seus jocs, com GuildWars 2, on gràcies a la UAI els enemics suposen un desafiament superior que els de la majoria de jocs MMORPG pel fet que són més difícils de predir.

3.2 Antecedents

3.2.1 Apex Utility AI

Es tracta d'un framework per Unity que facilita la programació de Utility AI per videojocs [7]. És de pagament i costa $65. El seu principal avantatge és una interfície gràfica que facilita molt la programació als usuaris. També conté guies per saber com funciona, i la possibilitat d'ampliar el codi utilitzant llibreries C#.

3.2.2 Crystal AI

Es tracta d'un altre framework de programació de Utility AI, però es pot utilitzar tant en Unity com en .NET 3.5 o superior. Es tracta d'una versió experimental gratuïta, però demanen que es doni crèdit al creador. També té una versió comercial per Unity que costa $48.89. La versió experimental no té una interfície gràfica.

3.2.3 Plataforma de Utility AI de Guild Wars 2

Tot i que no està oberta al públic, en el vídeo *Building a Better Centaur: AI at Massive Scale* [8] es pot apreciar que els dissenyadors tenen una interfície gràfica que els permet afegir la Utility AI als enemics que creen. D'aquesta manera s'estalvien el problema de tenir un programador que faci la feina i el mateix dissenyador ho pot fer.

3.3 Necessitats d'informació

Les fonts que es necessiten per obtenir la informació es tenen a l'abast. Inclouen articles científics i escrits que es poden trobar per internet, i webs especialitzades amb continguts de tècniques de AI i com aplicar-les a videojocs.

Respecte a la Utility AI, es pot trobar informació buscant per internet. També hi ha algun llibre que explica el necessari per entendre com funciona i què s'ha de tenir en compte a l'hora d'aplicar-la.

Per la part tècnica de la implementació, existeixen múltiples manuals i guies de programació de videojocs per internet que es poden consultar lliurement.

Per veure en més detall i exemples concrets dels diferents documents o llocs d'on es traurà la informació, consultar el punt 5.1 d'aquest mateix document.

4 Objectius i abast

4.1 Objectius del projecte

- Facilitar la programació de Utility AI per a programadors de videojocs en un motor de jocs existent.

- Permetre de personalitzar les funcions d'utilitat amb els valors d'entrada, que el framework s'encarregarà de recuperar del context quan els necessiti.

- Obtenir la millor acció que es pot portar a terme d'un conjunt d'accions, on cadascuna té una manera assignada per calcular la utilitat de l'acció.

- Incorporar el framework com a part del motor de jocs; facilitant el seu ús, instal·lació i portabilitat.

- Permetre d'utilitzar les funcions matemàtiques de càlcul d'utilitat que estan prèviament definides, sense que el programador les hagi d'implementar.

- Poder incorporar una funció matemàtica de càlcul d'utilitat personalitzada si aquesta no està prèviament definida.

- Aconseguir independència entre els valors del context i els càlculs de la Utility AI.

4.2 Abast

- Els càlculs del framework arribaran fins a la presa de decisió de la millor acció que es pot portar a terme. Qualsevol nivell superior, com podria ser fer un pla d'accions estarà a càrrec del programador usuari del framework.

- El framework no s'encarregarà d'assegurar que els valors d'entrada necessaris existeixen en el context, és el programador el que se n'haurà d'assegurar.

4.3 Objectius del producte i del client

Degut a que el projecte és de desenvolupament, els objectius del projecte i del producte són els mateixos.

Per altra banda, com que el producte de moment està orientat a un sol tipus de client, també coincideixen els objectius del client amb els del producte.

4.4 Públic potencial

- Programadors de AI per videojocs.

5 Metodologia

Per portar a terme el projecte, és necessària una primera fase de recerca d'informació. En aquesta fase es farà la recerca sobre l'estat de l'art, el context en el que es desenvolupa i es pensaran maneres de com implementar la solució. Seguidament, es passarà al desenvolupament del producte.

5.1 Recerca d'informació

La major part de la informació sobre la Utility AI, i com plantejar el framework es traurà del llibre de Utility AI de Mark Dave [9]. També es consultaran els documents [10], [11] i [12].

La recerca d'informació es centra en webs de solvència contrastada, especialitzades en el desenvolupament de videojocs. Gràcies a aquestes es podrà veure l'estat actual del mercat, solucions que s'han trobat al problema i possibles implementacions, entre altres. Alguns exemples de possibles webs on es buscarà informació són:

- Fòrums de desenvolupadors de videojocs, com 3DJuegos, Gamedev [13] o Gamasutra [14].

- Llocs web especialitzats en el desenvolupament de videojocs, com Ludust [15] o Teamups [16].

- Vídeos de conferències del GDC (Game Developers Conference) [17], la conferència anual més important respecte als videojocs.

5.2 Metodologia de desenvolupament

La metodologia de desenvolupament que es farà servir serà una metodologia pròpia adaptada a les necessitats del projecte. Aquesta metodologia està basada en el mètode de desenvolupament de software per prototipatge [18], ja que en cada iteració es produirà un prototip funcional del framework. El procés es dividirà en les següents fases:

- Primer, es farà la base del framework: tot allò necessari per tal que es pugui fer un càlcul d'utilitat i obtenir l'acció que s'ha de portar a terme.

- Es dissenyarà i programarà un videojoc senzill que serveixi per testejar la funcionalitat, usabilitat i comoditat del framework.

- Utilitzant el videojoc i l'experiència obtinguda d'utilitzar el framework, s'analitzaran els problemes detectats per tal de modificar i/o millorar el framework.

- Es modificarà el framework seguint les observacions del punt anterior, i es retornarà al segon punt d'aquesta llista.

Per motius de temps, només s'ha pogut arribar fins a la producció del primer prototip del framework i al seu posterior testeig i avaluació. No s'ha arribat a la quarta fase mencionada, ni a la creació del segon prototip del framework. Però el mètode de treball descrit a dalt es considera el millor per seguir aplicant en cas que es segueixi treballant en el framework en un futur.

6 Requeriments

6.1 Requeriments funcionals

- Acoblar el framework al motor de jocs per tal d'utilitzar-lo.

- Poder programar funcions d'utilitat, indicant quins valors s'han d'utilitzar.

- Assignar a cada acció una funció d'utilitat.

- Niar o composar les funcions d'utilitat per crear funcions complexes.

- Calcular l'acció i triar-la amb una sola crida al framework.

- Obtenir els valors d'entrada de les funcions d'utilitat des del context en el moment en què es necessiten.

- Obtenir els valors des del context. El mètode ha de retornar un error o un valor no vàlid en el cas que el valor no es pugui obtenir.

- Reutilitzar les accions en diferents crides de càlcul.

- Poder afegir funcions d'utilitat escrivint el codi, si és necessari.

- Utilitzar funcions d'utilitat prèviament implementades, de forma que només s'hagin de definir els valors d'entrada.

6.2 Requeriments no funcionals

- El temps de càlcul del framework no pot fer que el joc vagi lent. Es considera que el framework causa que el joc vagi lent si fa que els jugadors siguin capaços de percebre el temps de càlcul de la AI. Si el temps de càlcul supera el límit establert, que ha de ser prou petit com perquè una persona no ho noti, el framework ha de portar a terme una acció escollida de forma prèviament definida, per evitar una experiència negativa en els jugadors.

6.3 Requeriments tecnològics

- Accés a un ordinador de sobretaula d’última generació.

- Accés a un videojoc per poder testejar el framework.

- Llicència del motor de jocs que s’hagi triat.

7 Disseny del Framework

7.1 Requeriments no funcionals

Els requeriments no funcionals no s'han implementat en aquesta primera iteració de desenvolupament del framework, perquè s'ha considerat que en la primera iteració era més important implementar el funcionament bàsic del framework.

7.2 Desenvolupament dels requeriments funcionals

En aquest apartat s'identifica i es valora, per a cada requeriment funcional del sistema, com s'ha d'implementar en la fase de codificació.

Acoblar el framework al motor de jocs per tal d'utilitzar-lo:

El framework s'ha d'incorporar com a .dll dins del sistema de Unity. Una vegada incorporat com a llibreria, s'ha de poder utilitzar amb facilitat dins dels scripts del joc.

Poder programar funcions d'utilitat, indicant quins valors s'han d'utilitzar:

A l'hora de programar les funcions, s'han d'introduir els noms dels valors del context que s'han d'utilitzar en el càlcul de la utilitat.

Assignar a cada acció una funció d'utilitat:

S'ha d'assignar una funció d'utilitat en el moment de creació de les accions.

Niar o composar les funcions d’utilitat per crear funcions complexes:

En el moment de la definició de les funcions d’utilitat es podran utilitzar funcions de composició, preparades en la implementació del framework.

Calcular l’acció i triar-la amb una sola crida al framework:

En el moment que el joc ho desitgi, es farà una única crida al framework que incorporarà totes les accions a contemplar.

Aquesta crida consistirà en un mètode o funció, que contindrà les accions en els paràmetres i que escollirà l’acció de forma diferent segons la funció que s’hagi utilitzat. El funcionament de les crides es defineix en el punt 8.2.4 d’aquest document.

La crida del mètode ha de poder saber quina és l’acció que s’està portant a terme en aquell moment (o l’última que s’ha portat a terme). S’introdueix l’acció actual com a un dels paràmetres de la funció.

Les accions han de poder contenir un factor d’inèrcia per ajudar a decidir si cal canviar d’acció o si és millor mantenir l’acció actual. En cas que l’acció no la tingui definida, no es considerarà l’acció actual de forma diferent a les altres. Aquest valor només s’utilitza en algunes de les funcions de tria d’accions, no en totes.

Obtenir els valors d’entrada de les funcions d’utilitat des del context en el moment en què es necessiten:

El retorn dels valors des del context es farà per defecte de forma predefinida, amb els valors màxim, mínim i actual. És responsabilitat del programador de la AI programar com s’obtenen aquests tres valors.

El disseny del context ha de facilitar al programador la implementació de com obtenir els valors necessaris per fer els càlculs.

Obtenir els valors des del context. El mètode ha de retornar un error o un valor no vàlid en el cas que el valor no es pugui obtenir:

En el cas que un valor del context no existeixi quan es vol obtenir per fer els càlculs, automàticament es descartaran els càlculs d'utilitat d'aquella acció.

Reutilitzar les accions en diferents crides de càlcul:

Ni les accions ni les funcions de càlcul d'utilitat no es modifiquen en les crides de càlcul, permetent la seva reutilització.

El programador del videojoc s'encarrega de poder recuperar o reutilitzar les accions en diferents moments del joc en cas que sigui necessari.

Poder afegir funcions d'utilitat escrivint el codi, si és necessari:

Les funcions de càlcul d'utilitat es podran definir en el moment de la creació o es podran utilitzar funcions bàsiques ja definides.

Les funcions de càlcul d'utilitat definides escrivint el codi han d'estendre i implementar els mètodes de la classe abstracta que defineix les funcions d'utilitat.

Utilitzar funcions d'utilitat prèviament implementades, de forma que només s'hagin de definir els valors d'entrada:

Quan s'utilitzen funcions de càlcul d'utilitat predefinides en el framework, només cal posar el nom del valor que s'ha d'obtenir en els paràmetres de la funció.

Els tipus de funcions definides en el framework i el seu funcionament s'han definit en l'apartat 7.3.2 d'aquest document.

7.3 Especificació de les classes del domini

Després d'analitzar els requeriments, s'ha decidit que es necessita de les següents classes a l'hora de codificar el framework: **Context , UtilityFunction**, **Action**, **BestActionSelector**. En la Figura 7.1 mostra el diagrama de classes complet del framework.

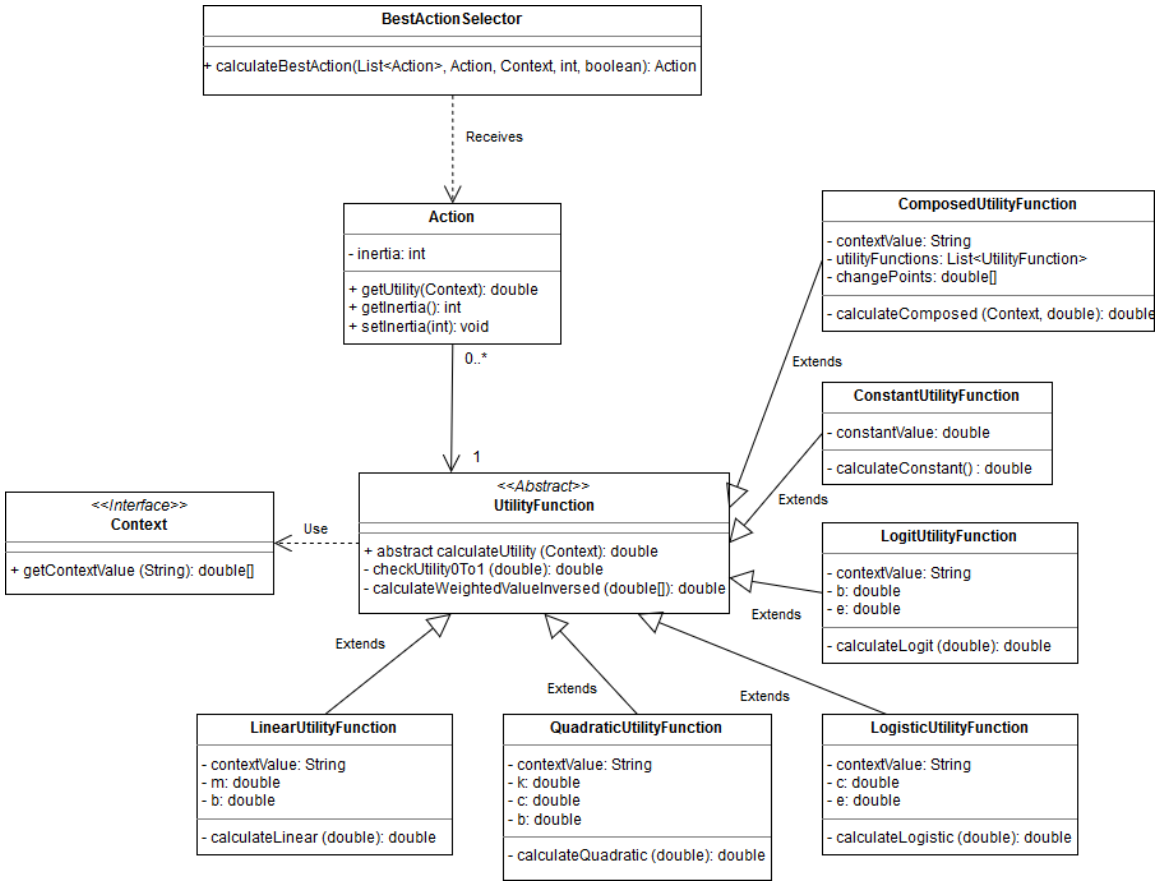


Figura 7.1 Diagrama de classes del framework.

7.3.1 Context

Context es tracta d'una Interfície. És l'única part del framework que el programador de la AI ha de codificar obligatòriament, ja que fa de nexe entre l'estat actual del joc i el framework.

Les classes que implementin Context hauran de codificar l'únic mètode públic d'aquesta interfície: el mètode **getContextValue**. Aquest, rep com a paràmetre un *String* que coincideix amb el nom del valor que es vol recuperar des del framework. I ha de retornar un *array* de tres *double*, que són el valor màxim, el valor mínim i el valor actual del *String* que s'ha rebut com a paràmetre.

Així doncs, l'obtenció dels valors de l'estat actual del joc queden sota la responsabilitat del programador del videojoc o de la AI.

7.3.2 UtilityFunction

La classe UtilityFunction es tracta d'una classe abstracta, i les classes que heretin d'aquesta, hauran d'implementar el mètode abstracte **calculateUtility**. També dos mètodes ja codificats que es diuen **checkUtility0To1** i **CalculateWeightedValueInversed**.

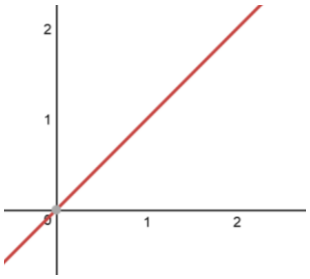
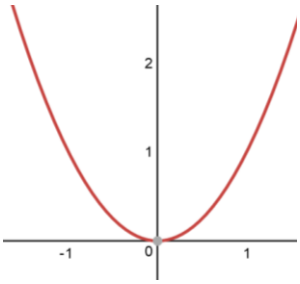
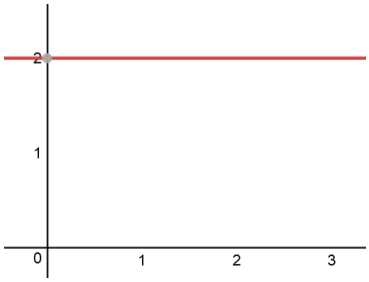
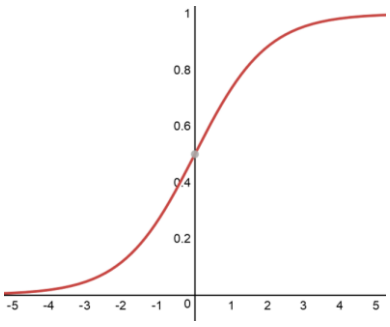
El mètode **checkUtility0To1** s'utilitza per comprovar que mai es retorna una utilitat inferior a zero o superior a 1. Abans de retornar la utilitat, s'ha de cridar aquest mètode i retornar el valor que retorna.

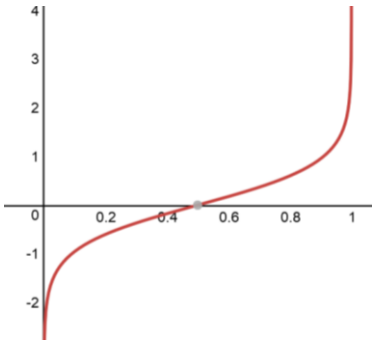
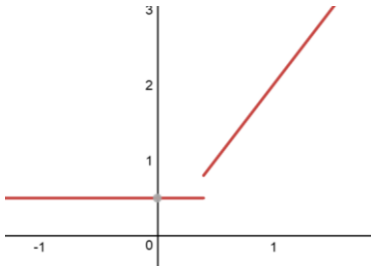
El mètode **CalculateWeightedValueInversed** rep com a paràmetre l'array de double que retorna la funció de Context, i en calcula el valor actual ponderat entre el mínim i el màxim transformat entre 0 i 1, a punt per utilitzar en les funcions de càlcul de utilitat de les implementacions de UtilityFunction. Com més a prop del màxim estigui el valor actual, més petit (més a prop del 0) serà el valor de retorn. Així s'aconsegueix que les funcions d'utilitat, on el valor de **y** (la utilitat) augmenta en funció de que **x** (el valor actual) augmenta: com més a prop del valor mínim, més gran serà la utilitat.

La implementació del mètode **calculateUtility** és el que determina com es calcula la utilitat. Utilitza el paràmetre Context que se li passa com a paràmetre per buscar els valors de l'estat del joc que necessita per calcular la utilitat i retorna la utilitat després de passar pel mètode **checkUtility0To1**. En el framework hi ha un seguit de classes que hereten de la classe UtilityFunction i que ja tenen implementat aquest mètode. En el cas que el programador de la AI vulgui una implementació concreta, la responsabilitat de la implementació recau sobre ell.

Les classes que hereten de UtilityFunction i que ja estan programades en el framework són les següents: **LinearUtilityFunction**, **QuadraticUtilityFunction**, **ConstantUtilityFunction**,

LogisticUtilityFunction, **LogitUtilityFunction** i **ComposedUtilityFunction**. En la Taula 7.1 es poden veure les diferents implementacions amb la funció matemàtica que calcula la utilitat. En les fórmules, la x és el valor que es busca al context i la y és el valor de la utilitat.

Nom	Funció de càlcul	Representació gràfica
Linear	$y = mx + b$	
Quadratic	$y = (x - c)^k + b$	
Constant	$y = b$	
Logistic	$y = \frac{1}{1+e^{-x+c}}$	

Logit	$y = \log_e\left(\frac{x}{1-x}\right) + b$	
Composed	$y = \begin{cases} mx + b, & x < 0{,}5 \\ b, & x \geq 0.5 \end{cases}$	

Taula 7.1 Classes que hereten de UtilityFunction. Font: Les funcions extretes de [9] i els gràfics elaboració pròpia amb l'eina online Desmos [19].

A l'hora de decidir quines funcions matemàtiques incorporar, s'ha decidit optar per les més simples i comuns, de manera que es pugui abastar el màxim de possibles necessitats per part dels programadors que utilitzin el framework.

Els paràmetres de les funcions que no són **x**, com **m** o **b** en la funció linear, es defineixen en el constructor de la classe. S'ha considerat d'aconseguir el seu valor a través del context per permetre canviar el comportament depenent de l'individu o de l'estat actual, però s'ha descartat i es pot considerar com una futura millora.

7.3.3 Action

La classe Action conté la informació necessària per calcular la utilitat de l'acció. Té dos atributs anomenats **inertia** i **utilityFunction**, i tres mètodes públics anomenats **getUtility**, **getInertia** i **setInertia**.

L'atribut **inertia** (inèrcia) consisteix en un valor *int*. S'utilitza per indicar que una acció té prioritat a l'hora de decidir quina acció s'ha de portar a terme, si aquesta ja s'està portant a terme en aquest moment. Com que no totes les accions han de tenir prioritat obligatòriament, el valor de la inèrcia no és obligatori en el constructor de les Accions, i s'inicialitza automàticament amb el valor zero. La inèrcia es pot utilitzar en el càlcul de la classe BestActionSelector i es pot modificar durant l'execució del joc.

L'atribut **utilityFunction** és un objecte que hereta de la classe UtilityFunction. S'utilitza per calcular la utilitat de l'acció i es defineix en el moment de la creació de l'Action.

El mètode **getUtility** necessita un paràmetre, que és una implementació de la classe Context. Quan es crida aquest mètode, la classe Action crida el respectiu mètode **calculateUtility** de l'atribut **utilityFunction**, i retorna el mateix valor que rep.

El mètode **getInertia** serveix per recuperar el valor en el cas que el BestActionSelector el necessiti, i el mètode **setInertia** serveix per canviar el valor de l'atribut **inertia**.

7.3.4 BestActionSelector

La classe BestActionSelector s'encarrega de calcular la millor acció que es pot portar a terme d'un grup d'accions. Té un únic mètode públic anomenat **calculateBestAction**.

El mètode **calculateBestAction** és el que s'invocarà quan es vulgui calcular la millor acció que es pot portar a terme en aquell moment. Per tal que funcioni, ha de rebre els següents paràmetres:

Llista d'accions: és el conjunt d'accions que es poden portar a terme i entre les quals s'ha de triar la millor.

Acció actual: podria ser un valor nul. És l'acció actual que s'està portant a terme. És la que es retorna per defecte en el cas que no es superi el límit de temps establert pel seu càlcul i l'única a la qual se li afegeix el valor de la inèrcia. En cas que sigui nul·la, es retorna una acció aleatòria si s'acaba el temps.

Context: objecte que implementa la interfície Context. És el que es passarà a la funció de càlcul per obtenir els valors pertinents.

int: el mode de càlcul de l'acció. Els diferents modes de càlcul estan definits com a atributs constants de la classe i són els que diuen com triar l'acció.

boolean: indica si s'ha d'aplicar o no la inèrcia en la tria de l'acció actual.

7.4 Diagrama de seqüència

La Figura 7.2 mostra el diagrama de seqüència del framework. Aquest, mostra el comportament usual entre el joc i el framework i les principals funcions que s'utilitzen durant l'execució de la AI.

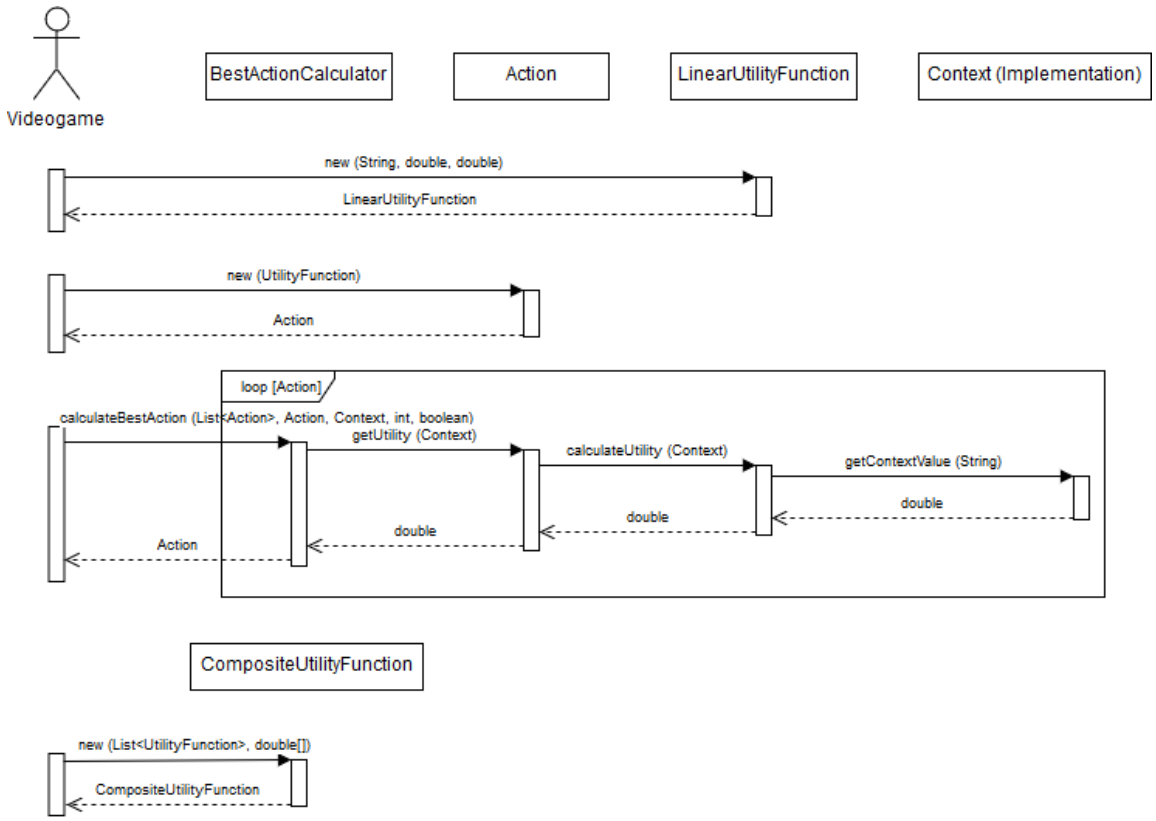


Figura 7.2 Diagrama de seqüència del framework.

7.5 Codificació

El framework s'ha desenvolupat amb l'aplicació Microsoft Visual Studio i amb el llenguatge de programació C#. S'ha decidit utilitzar aquesta aplicació i llenguatge perquè C# és el llenguatge natiu de l'aplicació Unity. En la instal·lació de la plataforma Unity normalment s'instal·la l'aplicació Microsoft Visual Studio com a editor per defecte dels scripts. D'aquesta manera només ha estat necessari utilitzar un llenguatge de programació i una aplicació de desenvolupament per programar el framework i per escriure els scripts del joc.

8 Guia d'ús del Framework

Aquesta guia s'ha redactat per tal d'utilitzar correctament el framework. A continuació s'exposen els passos a seguir per programar una AI i quines consideracions s'han de tenir en compte.

Els exemples que apareixen són en el llenguatge de programació C#.

8.1 Les accions de l'agent

El primer que cal fer per utilitzar el framework és definir les accions que ha de portar a terme l'agent del qual es vol programar la AI, així com la funció d'utilitat que es vol utilitzar d'entre les implementacions de UtilityFunction que venen incorporades amb el framework. A la Taula 7.1 es poden veure les diferents implementacions.

Cada una de les UtilityFunction té un cert número de paràmetres que s'han de completar, definits en el seu propi constructor. El més important d'aquests i l'únic en comú, excepte en el cas de la ConstantUtilityFuncion és el **String amb el nom del valor** que es buscarà en la implementació de Context durant el càlcul de la utilitat.

Una vegada decidida l'acció i la funció de càlcul de la utilitat que es vol utilitzar, es pot passar a codificar. En el següent exemple, es crea l'acció anomenada **eat**, utilitzant la funció d'utilitat **LinearUtilityFunction**. I segons els paràmetres de la funció d'utilitat es buscarà el valor anomenat Hunger del Context, i el càlcul es farà seguint la funció matemàtica y=1*x+0:

```
Action eat = new Action(new LinearUtilityFunction("Hunger", 1, 0));
```

Per ajustar les funcions d'utilitat només cal considerar la part de la funció de l'eix x entre els valors 0 i 1, ja que són els únics valors que es consideraran després de la utilització de la funció **CalculateWeightedValueInversed**.

A continuació es mostra un altre exemple, aquesta vegada utilitzant la funció d'utilitat **ConstantUtilityFunction**, que és l'única que no necessita obtenir un valor del context:

```
Action sleep = new Action(new ConstantUtilityFunction(0.2));
```

8.2 Inèrcia de les accions

La inèrcia d'una acció s'utilitza per evitar que l'agent canviï contínuament l'acció que està portant a terme. Si la inèrcia de l'acció actual no és zero, l'agent tindrà tendència a continuar realitzant la mateixa acció, ja que al calcular la utilitat aquesta estarà bonificada.

Posant un exemple amb números per facilitar la comprensió, si l'acció actual té **una utilitat de 0.5 i una inèrcia de 0.5**, la utilitat total es converteix en 0.5 + 0.5*0.5 = **0.75**. Per tant, l'agent només canviarà d'acció si una altra acció té una utilitat superior a 0.75.

No és obligatori utilitzar inèrcia en les accions, ja que **per defecte** el valor és **zero**. Si es vol utilitzar un altre valor, es pot afegir com a paràmetre del constructor en la creació de l'acció. En el següent exemple es crea una acció amb una inèrcia de **0.2**:

```
Action eat = new Action(0.2, new LinearUtilityFunction("Hunger", 1, 0));
```

També es pot modificar després de la creació de l'acció. En el següent exemple es mostra com fer-ho. Aquest exemple i l'anterior acaben amb una acció idèntica:

```
Action eat = new Action(new LinearUtilityFunction("Hunger", 1, 0));
eat.Inertia = 0.2;
```

El **valor màxim de la inèrcia** de les accions que es considera a l'hora de fer els càlculs a través del BestActionCalculator és **1**; duplicant la utilitat i convertint la utilitat de l'acció en el rang de 0 a 2, enlloc del rang bàsic de 0 a 1. Si una inèrcia és negativa, s'ignorarà a l'hora de fer els càlculs.

La modificació de la utilitat deguda a una inèrcia massa gran pot causar un comportament no desitjat en l'agent. Tot i que estan permesos, **no es recomana posar valors pròxims a 1** a la inèrcia, ja que l'objectiu de la inèrcia és evitar el canvi continu entre dues accions d'utilitat semblant. Amb una inèrcia massa gran es genera un **comportament compulsiu** per part de l'agent, que només canviarà d'acció quan la utilitat de l'acció actual sigui molt petita i la utilitat d'una altra acció sigui molt gran.

8.3 La funció d'utilitat ComposedUtilityFunction

La funció d'utilitat ComposedUtilityFunction és un cas especial entre les funcions que estan definides per defecte en el framework, perquè utilitza altres UtilityFunction per fer els càlculs.

Per poder-la utilitzar, se li ha de passar una **List de UtilityFunction** i un **array de double** que indica els punts de canvi entre una UtilityFunction i la següent. És important que tant la List com l'array estiguin ordenats, ja que per fer els càlculs els agafarà per índex.

La tria de la funció d'utilitat que s'utilitza es fa segons el valor que es busca al Context. Si la ComposedUtilityFunction té dues UtilityFunction amb la separació en el punt 0.4, utilitzarà la primera de la List si el valor obtingut és més petit que 0.4, i la segona si és més gran que 0.4.

En el següent exemple es mostra com crear efectivament una ComposedUtilityFunction. En aquest cas es crea l'acció anomenada **eat**, combinant una **ConstantUtilityFunction** i una **LinearUtilityFunction** per fer que l'agent ignori la gana si aquesta no supera el **0.4**. La Figura 8.1 mostra el gràfic resultant d'aquesta funció d'utilitat composta.

```
List<UtilityFunction> composedFunctions = new List<UtilityFunction> {
       New ConstantUtilityFunction(0);
       new LinearUtilityFunction("Hunger", 1, 0) };
double[] cutPoints = new double[] {0.4};
Action eat = new Action(new ComposedUtilityFunction("Hunger", composedFunctions,
cutPoints));
```

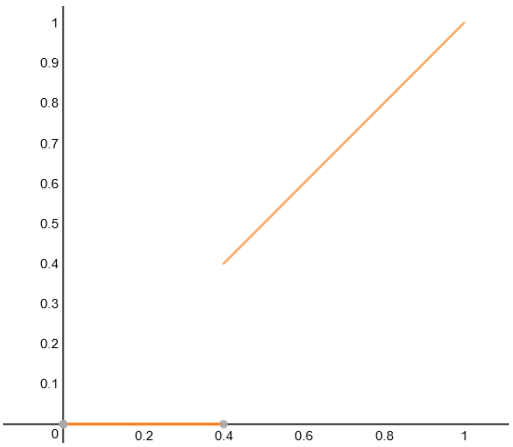


Figura 8.1 Exemple gràfic de ComposedUtilityFunction. Font: Elaboració pròpia amb [19].

Cal remarcar que el valor que es busca en el Context en cada una de les UtilityFunction dins de la ComposedUtilityFunction no té per què ser el mateix que el que s'utilitza per fer la tria de quina UtilityFunction utilitzar. Però el resultat pot resultar impredictible si no s'ha dissenyat correctament, causant un comportament inesperat en l'agent.

8.4 Les funcions d'utilitat personalitzades

Per utilitzar una funció d'utilitat personalitzada s'ha de crear una classe que hereti de la classe UtilityFunction, i codificar el mètode abstracte **CalculateUtility**. UtilityFunction té dues funcions implementades que poden facilitar la codificació.

CalculateWeightedValueInversed agafa els tres valors que retorna el Context i retorna un únic valor entre 0 i 1. Aquest valor representa la relació del valor actual amb els valors mínim i màxim, sent 0 el màxim i 1 el mínim.

CheckUtility0To1 es pot utilitzar per assegurar que mai es retorna una utilitat inferior a 0 ni superior a 1. És important utilitzar aquest mètode abans del retorn de la funció **CalculateUtility** perquè totes les utilitats haurien d'estar en el mateix rang.

És important tenir en compte que, en cas necessari, s'ha de poder obtenir el valor des del **Context** que es rep com a paràmetre del mètode **CalculateUtility**. Es recomana utilitzar una variable string amb el nom del valor per tal d'aconseguir el valor del Context.

En el següent exemple es mostra una part de la codificació per implementar una funció d'utilitat personalitzada.

```
Public class CustomUtilityFunction : UtilityFunction {
       private string contextValue;
       public override doublé CalculateUtility(Context context) {
              double[] values = context.GetContextValue(contextValue);
              …
              return CheckUtility0to1(utilityValue);
} }
```

8.5 Implementar Context

La codificació de la o les classes que implementin la interfície Context és independent del funcionament del framework. Només s'ha de tenir en compte que el retorn del mètode **GetContextValue** ha de retornar els valors màxim, mínim i actual en un array de double i en exactament aquest ordre per no interferir amb el funcionament del framework.

Durant la implementació de Context i la definició i creació de les accions, és important assegurar que els valors que es buscaran per fer els càlculs d'utilitat, els paràmetres String de la funció **GetContextValue**, es podran obtenir al Context. És fàcil de detectar perquè qualsevol error durant l'obtenció dels valors fa que es provoqui una excepció i no continua el càlcul de les utilitats de les accions.

8.6 Calcular la pròxima acció

Per a calcular la pròxima acció s'utilitza el mètode **CalculateBestAction** de la classe **BestActionCalculator**. Aquest mètode conté els paràmetres necessaris per permetre el màxim de flexibilitat en la decisió i càlcul de la pròxima acció. El mètode, amb els paràmetres inclosos, és el següent:

```
public static Action CalculateBestAction(List<Action> actions, Action
      currentAction, Context context, int chooseMethod, Boolean useInertia) {…}
```

El primer paràmetre és una **llista de les accions que pot portar a terme l'agent** i que s'han de considerar a l'hora de fer la tria. L'acció escollida serà una d'aquesta llista, però el mètode de tria dependrà de la resta de paràmetres.

El segon paràmetre és **l'acció actual de l'agent**, o l'última que ha realitzat si el càlcul es fa posteriorment. El correcte funcionament del mètode no depèn de si aquesta acció es troba o no en la llista de les accions del primer paràmetre, o si es passa un objecte *null*. Però s'ha de tenir en consideració que en qualsevol dels dos casos no serà possible aplicar inèrcia, encara que així s'hagi indicat en el cinquè paràmetre, com s'explica més avall.

El tercer paràmetre és la referència a un objecte d'una classe que implementi la interfície **Context**. El fet de que es passa com a paràmetre permet canviar de Context en diferents crides del mètode o utilitzar crides idèntiques del mètode canviant el Context per controlar a diversos agents amb la mateixa AI, entre altres.

El quart paràmetre és el **mètode de tria de l'acció**. Hi ha tres mètodes implementats: **Best Action**, **Weighted Random** i **Random Best 3**.

El **Best Action** calcula la utilitat de cada acció i retorna l'acció amb la utilitat més alta. Aquest mètode de tria és ideal quan es vol aconseguir que l'agent es comporti de la manera més **intel·ligent** possible, ja que sempre realitzarà l'acció amb més utilitat. Però el principal problema és que és **difícil ajustar les funcions d'utilitat**, ja que només que una estigui desajustada causarà que contínuament o mai realitzi aquella acció.

El **Weighted Random** calcula la utilitat de cada acció i retorna una acció aleatòria tenint en compte la proporció de la utilitat entre les accions. És més fàcil d'entendre amb un exemple: si hi ha tres accions amb les utilitats 0.1, 0.3 i 0.6, cadascuna tindrà respectivament un 10%, un 30% i un 60% de ser escollida. Aquest mètode de tria permet **més llibertat a l'hora d'ajustar les funcions d'utilitat**, perquè per probabilitats en algun moment realitzarà l'acció, i amb més facilitat si aquesta té una utilitat alta. Però per contra pot ser que l'agent realitzi **accions inesperades** i amb una utilitat realment baixa.

El **Random Best 3** calcula la utilitat de cada acció i retorna una acció aleatòria d'entre les tres amb la utilitat més alta. Aquest mètode de tria estaria **entre els dos anteriors**, ja que no s'assoleix un comportament ideal com amb el Best Action però no és tant estricte d'ajustar, i tampoc realitzarà mai les accions que tenen una utilitat molt baixa. És recomanable abstenir-se d'utilitzar aquest mètode de tria si no es té una llista d'accions de com a mínim 4 accions.

El cinquè paràmetre és un boolean que indica si es **vol utilitzar la inèrcia** en l'acció actual. Per tal que tingui sentit posar el paràmetre com a *True* és necessari que l'acció actual no sigui *null* i que aquesta estigui en la llista d'accions del primer paràmetre, ja que en cas contrari no es pot aplicar inèrcia i el resultat serà el mateix que si el paràmetre és *False*. Les accions que tenen inèrcia amb valor 0 tampoc alteren el resultat. En qualsevol altre cas, la inèrcia s'aplica després del càlcul d'utilitat de l'acció, **augmentant el valor d'utilitat** que s'utilitza en els mètodes de tria del quart paràmetre.

En el següent exemple es mostra com utilitzar el mètode CalculateBestAction, utilitzant les accions **eat** i **sleep**, amb l’acció actual **eat**, el mètode **Best Action** i **sense inèrcia**.

```
List<Action> actions = new List<Action> { eat, sleep };
Action bestAction = BestActionCalculator.CalculateBestAction(actions, eat,
contextImplementation, BestActionCalculator.BEST_ACTION, false);
```

9 Disseny i desenvolupament de la simulació per testejar el Framework

A l'hora de dissenyar el joc, s'ha vist que no és necessari que hi hagi interacció entre el jugador i el personatge per tal de testejar la AI. Com que facilita la programació, però no afecta a la AI, s'ha decidit optar per aquest camí. Sumat al fet que el joc és de tipus **simulació**, es deixa d'anomenar "joc" i es passa a anomenar "simulació".

9.1 Objectiu

L'objectiu de la simulació és poder testejar el framework per programar Utility AI. Per tant, la simulació ha de ser senzilla i ha d'incorporar una AI que utilitzi el framework.

A continuació es descriu la simulació que s'ha dissenyat i implementat per testejar el primer prototip del framework.

9.2 Descripció

Un dels gèneres de videojocs que utilitzen més la Utility AI per controlar els NPC és el de **simulació**, i és per aquest motiu que s'ha basat la simulació en aquest gènere de videojocs.

Tenint el compte que l'objectiu és **testejar la AI i no fer una simulació interessant**, la part visual de la simulació consisteix en un seguit de blocs de text que mostren l'estat actual del personatge principal. Aquests blocs estan dividits en dos tipus: el bloc principal, que mostra els valors generals; i un seguit de blocs secundaris que mostren els diferents valors de l'estat del personatge.

En la Figura 9.1 es pot veure una captura de la simulació una vegada aquesta ha estat programada, on es pot veure que és molt simple d'aspecte.

Figura 9.1 Captura de pantalla de la simulació.

Respecte al funcionament de la simulació, cada període de temps preestablert, els diferents valors de l'estat del personatge baixen o pugen, depenent de l'acció que estigui portant a terme. I aquest haurà de mantenir-se viu el màxim de temps o bé aconseguir no morir-se mai. Cada activitat té associada una duració.

La responsabilitat d'aconseguir que el personatge no es mori, recau en el programador de la AI, que ha de programar una AI suficientment bona per aconseguir que sobrevisqui.

9.3 Especificació

A continuació es descriuen els diferents elements de la simulació. S'han tingut en compte els objectius de la simulació, el funcionament del framework i el funcionament de la plataforma de programació Unity per fer l'especificació de la simulació.

9.3.1 Funcionament de la simulació

El personatge té diversos **atributs** que defineixen l'**estat actual** d'aquest. Cadascun d'aquests atributs té definit un valor **mínim** i un **màxim**, i agafa com a **valor actual inicial** el valor màxim. Cada atribut té també definit un **modificador**, un número que indica com de ràpid es redueix (o augmenta) el valor actual de l'atribut si no s'està portant a terme una acció que el

modifica directament. El modificador és negatiu en cas que es redueixi el valor, i positiu en cas contrari.

La simulació té una **unitat de temps establerta**, i cada vegada que aquest temps passa, s'actualitza l'estat de la simulació. En aquest moment s'apliquen els modificadors als diferents atributs, o l'**incrementador de l'acció** que s'està portant a terme en cas que l'acció modifiqui l'atribut en qüestió. També es comprova la duració de l'acció, i si aquesta ja s'ha acabat, per si cal fer el càlcul de quina és la pròxima acció que el personatge ha de portar a terme. El valor de la unitat de temps és de **un segon**.

Entre dues accions hi ha una **pausa d'una unitat de temps**, que representa el temps que el personatge triga en canviar d'una acció a la següent.

El personatge té definides diverses **accions** que pot dur a terme. Aquestes accions corresponen a les diferents accions que estan definides a la AI. Cadascuna d'elles té un valor que indica la **duració** de l'acció en unitats de temps, i un llistat de parelles de valors que indiquen quin és **l'atribut que modifica** com a part de l'acció i el **valor del modificador**. En tot moment, el personatge ha de saber quina és **l'acció que està portant a terme**.

El personatge es **morirà** en el moment en el qual algun dels atributs arribi al valor mínim. Quan això passa, es mostra un missatge que ho anuncia i s'atura la simulació.

9.3.2 UI

La interfície gràfica de la simulació, està formada per blocs de text.

El **bloc principal** es troba en la part superior i ocupa la pantalla d'esquerra a dreta. Dins d'aquest bloc es mostren els següents valors:

- **Nom**: un camp de text editable, utilitzat per identificar la prova que s'està duent a terme. És especialment útil en cas de captures de pantalla o gravacions.

- **Felicitat**: mostra la felicitat global del personatge, en un valor sobre 100.

- **Temps**: mostra el temps que ha passat des que s’ha iniciat la simulació, és a dir, el temps que ha sobreviscut el personatge. En cas que el personatge es mori, s’ha d’aturar. El temps es mostra en minuts i segons.

- **Activitat**: mostra el nom de l’acció que està portant a terme el personatge actualment.

Els **blocs secundaris** es poden veure a sota del bloc principal. Fins a un total de 3 blocs diferents podran ocupar la pantalla d’esquerra a dreta. Dins de cada bloc es mostra el següent:

- **Atribut**: mostra el nom de l’atribut del personatge que representa el bloc.

- **Valor actual**: mostra el valor actual de l’atribut. Al costat del valor es mostra un número entre parèntesi, que mostra quin canvi té en cada unitat de temps en funció de l’acció que està portant a terme el personatge.

- **Màxim**: mostra el valor màxim que pot prendre l’atribut. És un valor fix.

- **Mínim**: mostra el valor mínim que pot prendre l’atribut. És un valor fix.

La Figura 9.2 mostra el disseny de la interfície gràfica de la simulació, amb valors d’exemple en els diferents textos.

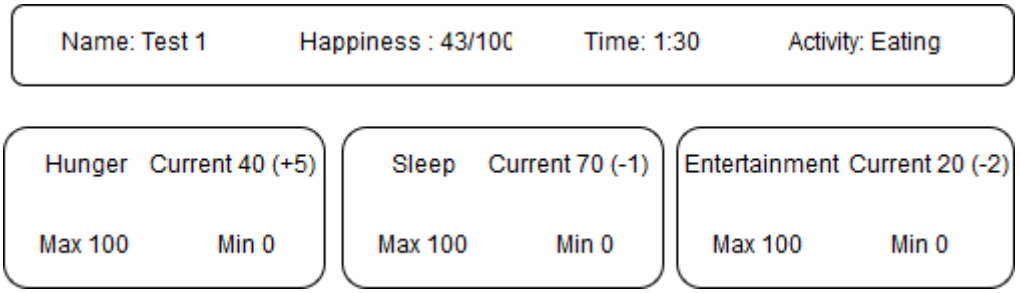


Figura 9.2 Disseny de la UI de la simulació.

9.3.3 Atributs i Accions

A la Taula 9.1 es mostren els valors dels atributs que defineixen l'estat del personatge. Per facilitar la programació, la interpretació i l'avaluació dels resultats, tots els atributs tenen el mateix màxim i mínim i els seus valors són 100 i 0 respectivament.

Atribut	Màxim	Mínim	Modificador
Fullness	100	0	-3
Awake	100	0	-1
Relax	100	0	-1
Clean	100	0	-1
Money	100	0	-2
Energy	100	0	+1

Taula 9.1 Atributs del personatge.

A la Taula 9.2 es mostren les accions que pot portar a terme el personatge, i els valors i atributs associats a aquestes.

Acció	Unitats de Temps	Atribut	Modificador
Eat	4	Fullness	+10
Sleep	8	Awake	+5
		Relax	+1
Work	6	Money	+8
		Relax	-4
Exercise	4	Relax	+15
		Clean	-5
		Energy	-10
Watch TV	2	Relax	+10
Shower	3	Clean	+40

Taula 9.2 Accions del personatge.

9.3.4 AI

Per tal de testejar i avaluar el framework, s'ha decidit aplicar a cada una de les accions del personatge una de les diferents **UtilityFunction** actualment implementades en el framework. A continuació s'exposa quina implementació de **UtilityFunction** proveïda pel framework utilitza cada acció, i quins valors agafa per fer el càlcul.

L'acció **Eat** utilitza la classe **LogisticUtilityFunction**. El nom del valor del context que busca per fer els càlculs és **Fullness**, el valor de c és **1** i el de e és ***e*** (la constant matemàtica, base dels logaritmes neperians).

L'acció **Sleep** utilitza la classe **LogitUtilityFunction**. El nom del valor del context que busca per fer els càlculs és **Awake**, el valor de b és **1** i el de e és ***e*** (la constant matemàtica, base dels logaritmes neperians).

L'acció **Work** utilitza la classe **LinearUtilityFunction**. El nom del valor del context que busca per fer els càlculs és **Money**, el valor de m és **0.8** i el de b és **0.2**.

L'acció **Exercise** utilitza la classe **ComposedUtilityFunction**. El **punt de tall** entre les dues funcions que utilitza és **0.6**. La primera funció és **ConstantUtilityFunction**, amb valor de **0**. La segona funció és **LinearUtilityFunction**, on el nom del valor del context que busca per fer els càlculs és **Relax**, el valor de m és **2** i el de b és **-1.2**.

L'acció **Watch TV** utilitza la classe **ConstantUtilityFunction**, amb valor de **0.3**.

L'acció **Shower** utilitza la classe **QuadraticUtilityFunction**. El nom del valor del context que busca per fer els càlculs és **Clean**, el valor de k és **5**, el valor de c és **0** i el de b és **0**.

9.4 Implementació en Unity

La implementació de la simulació a la plataforma de videojocs Unity s'ha fet seguint les especificacions del punt anterior. En aquest punt es detalla la part tècnica de la implementació.

Quan es fa referència a elements o objectes de la simulació, aquests es poden observar a la pestanya de **Hierarchy**, amb els seus components a la pestanya **Inspector**. I tota la informació dels scripts es pot trobar a la pestanya **Project**, a la carpeta **Scripts**.

Per **acoblar el framework** amb la simulació creada amb la plataforma de Unity, s'ha compilat el codi del framework en format **.dll** i s'ha afegit a una carpeta anomenada **Plugins**. La carpeta Plugins es pot veure a la pestanya Project. Aquest és el funcionament per defecte de Unity en respecte a **Managed Plugins**: plugins que no depenen de la plataforma en la que el producte de Unity es llençarà. Per més informació sobre Managed Plugins i plugins de Unity podeu mirar les referències [20], [21] i [22].

9.4.1 Implementació de la UI

La part gràfica de l'aplicació utilitza exclusivament la interfície gràfica incorporada en Unity anomenada **Canvas**. Els elements estan organitzats jeràrquicament, ja que facilita el control durant la programació. L'estructura jeràrquica també coincideix amb el funcionament dels scripts, que es detallen en el següent punt.

Dins del **Canvas** de la simulació, hi ha set **Panel** diferents, i cadascun conté un o més **Text** que mostren una part de la informació de l'estat del personatge:

- El Panel anomenat **Main Info** és el que mostra la informació del **bloc principal**.

- Els sis Panel anomenats **Atribute** mostren la informació d'un dels atributs del personatge.

- El Panel **Death Panel** només es mostra quan el personatge s'ha mort, per anunciar-ho.

Per crear la UI del joc, primer s'han mirat els vídeos de la part de UI Components de la referència [23] per tal d'entendre el funcionament dels diferents elements de UI en Unity.

9.4.2 Controladors de la UI

Els controladors de la UI s'utilitzen per enllaçar els elements de la UI amb l'script que controla la simulació, per tal que es mostri la informació de l'estat actual del personatge per pantalla.

En Unity, les **variables públiques** dels scripts es mostren en l'**Inspector** com a camps editables una vegada s'ha afegit l'script com a component d'un objecte [24]. S'ha utilitzat aquest comportament per referenciar els components gràfics especificats en el punt anterior i utilitzar-los després en els controladors de la UI.

A la simulació s'han incorporat diversos objectes buits, utilitzant la funció **Create Empty** de Unity. Els objectes buits de Unity s'utilitzen quan no es vol o no es necessita un objecte que tingui un comportament predefinit. S'han utilitzat set objectes buits pels diferents controladors de UI, organitzats i anomenats per mantenir l'estructura lògica i amb referències als elements del Canvas mencionats en el punt anterior.

Els objectes anomenats **UIAtribute** contenen un script UIAtribute. En aquest script hi ha les variables públiques per fer referència als **Text** d'un dels Panel **Atribute** del personatge. L'script s'encarrega de mostrar els valors actuals en cada un dels Text referenciats. En la Figura 9.3 es poden veure les variables públiques del script, que aquest apareix com a component del objecte, i els noms dels text que s'utilitzen en les variables públiques de l'script.

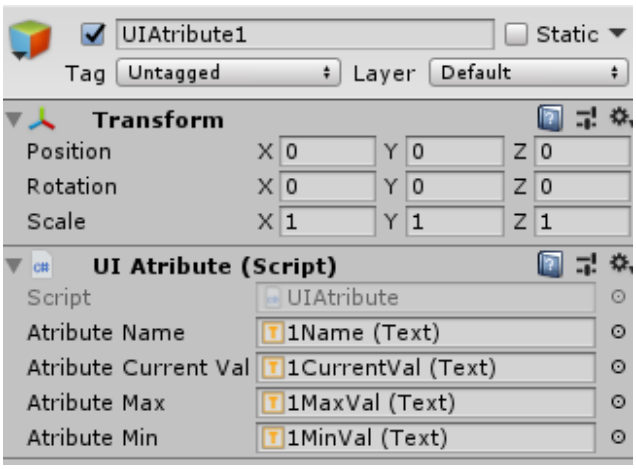


Figura 9.3 Captura de pantalla d'un UIAtribute de la simulació.

L'objecte anomenat **UIController** conté un script UIController. En aquest script hi ha les variables públiques per fer referència als objectes que contenen els script **UIAtribute**, així com

als **Text** del Panel **Main Info** i al Panel **Death Panel**. Durant la crida de Start() de l'script, s'obté la referència als scripts de UIAtribute dels objectes referenciats i es guarden en una llista. L'script s'encarrega de controlar els UIAtribute, de modificar els text de Main Info, i de mostrar o ocultar el Death Panel.

A la Figura 9.4, igual que a la figura anterior, es poden veure les variables públiques del script. En aquest cas, a part dels textos de l'estat del joc, també es veuen les referències als UIAtribute i al panell DeathPanel.

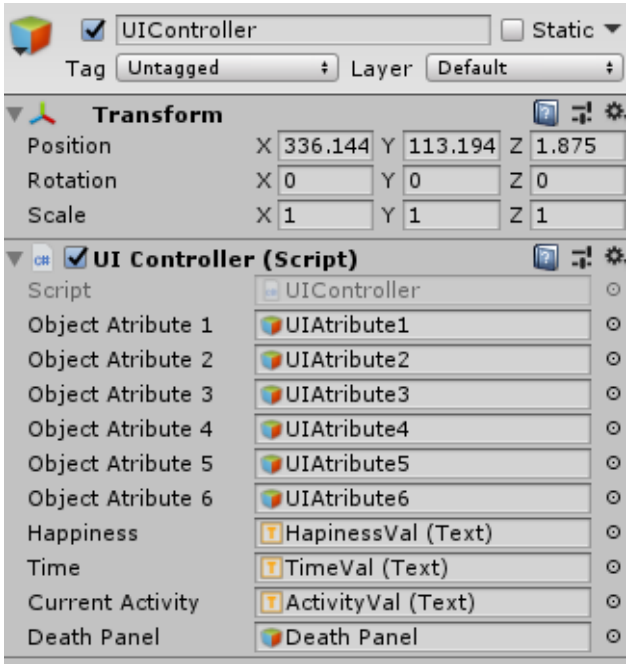


Figura 9.4 Captura de pantalla del UIController de la simulació.

9.4.3 Classes i Scripts

La part funcional de la simulació està controlada per codi de programació escrit en el **llenguatge C#**, ja que és el llenguatge natiu de Unity. A continuació s'expliquen les classes que s'han utilitzat, i la utilització de les classes del framework per implementar la AI. La implementació ha seguit la descripció de la simulació de l'apartat 9.3.1.

Es pot veure el diagrama de classes en la Figura 9.5, on també es poden veure els controladors de la UI. Pel funcionament de Unity, les classes que s'utilitzen com a scripts han d'estendre la classe MonoBehavior.

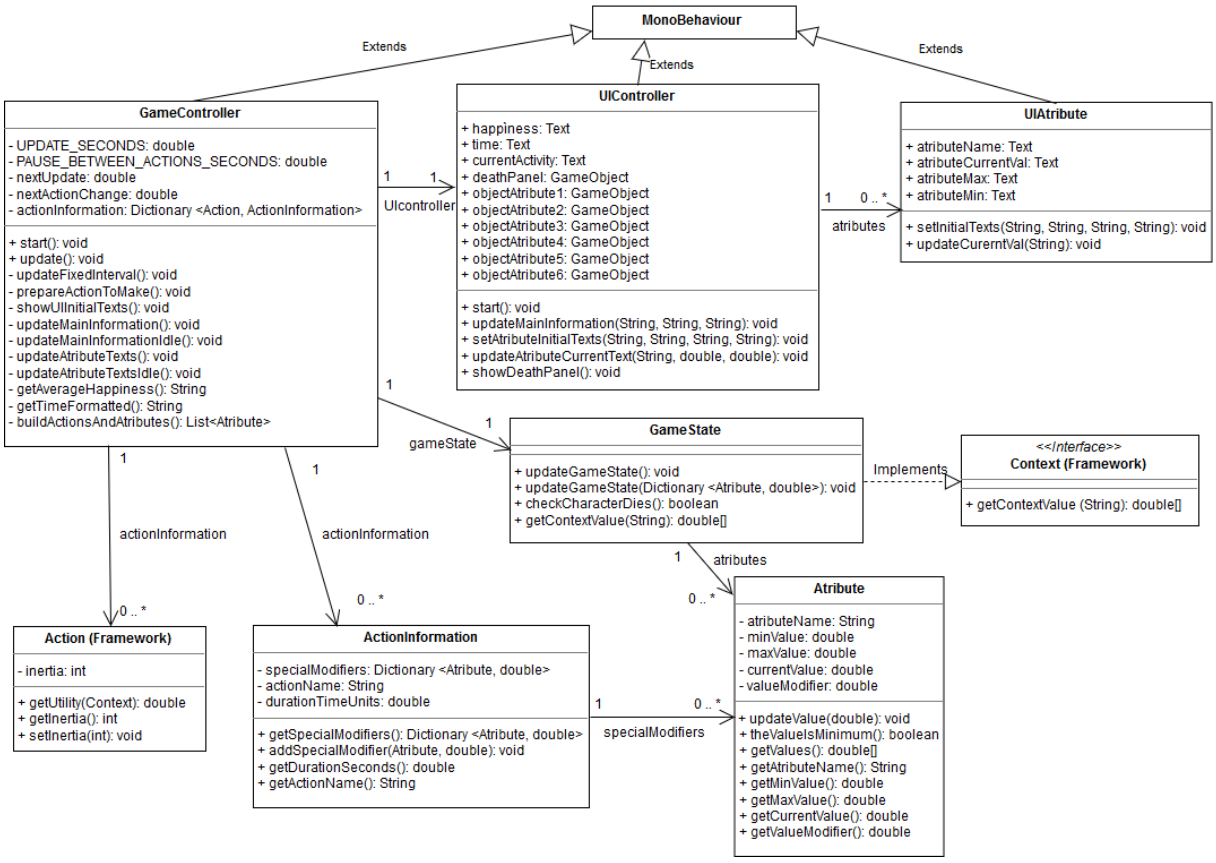


Figura 9.5 Diagrama de classes de la simulació.

La classe **Atribute** conté la informació sobre un dels atributs del personatge. És l'encarregada de modificar el valor actual de l'atribut.

La classe **GameState** és la classe que implementa la interfície del framework **Context** i el mètode **GetContextValue**. Hi ha dos motius per a aquesta decisió: el primer és que per definició, la classe que té la **informació de l'estat de la simulació** s'anomena context; i la segona és que GameState conté el **llistat d'Atribute**, convertint-se en la classe que té la informació per passar al framework quan aquest ho necessiti. GameState també és la classe que **actualitza l'estat de la simulació** modificant els valors dels Atribute i **comprova si el personatge mor**.

La classe **ActionInformation** s'utilitza per guardar la informació especial de cada una de les accions que pot portar a terme el personatge. Aquesta classe s'ha considerat necessària perquè la classe Action del framework no conté informació que no sigui la imprescindible pel càlcul de la utilitat. ActionInformation conté la informació sobre la **duració** de l'acció, el **nom** d'aquesta i el conjunt de **modificadors especials** dels Atribute per aquella acció.

La classe **GameController** és l'única classe de la part funcional de la simulació que es pot considerar un Script, perquè és l'única que hereta de la classe **MonoBehaviour** de Unity [25]. Com el seu nom indica, és la classe que controla el funcionament de la simulació i la que gestiona i controla la resta de classes. L'objecte anomenat **GameController** de la simulació és un objecte buit que conté l'script GameController, per tal que la simulació funcioni.

Per enllaçar amb la part gràfica de la simulació, la UI, GameController té una variable pública que rep la referència al **UIController**. També té com a variables el **GameState**, un diccionari [26] que conté les **Action** del framework amb la seva respectiva **ActionInformation**, i l'**Action actual**. Com que és la classe que conté les **Action**, també és on es creen.

GameController és la classe encarregada de **controlar el temps**, i per això té les variables necessàries que controlen el canvi de les accions, i les constants definides per l'interval de temps i per la pausa entre accions.

10 Prova i anàlisi del Framework

L'objectiu de la simulació era testejar el framework i obtenir les primeres valoracions des del punt de vista d'un usuari del framework per programar Utility AI. A continuació s'exposa un resum de les parts que s'han testejat i es discuteixen les possibles modificacions i/o noves funcionalitats per millorar el framework.

10.1 Què s'ha provat/testejat?

S'ha testejat la funcionalitat bàsica del framework, des del disseny de les accions de l'agent fins a les crides per escollir les accions i fer que l'agent les porti a terme.

La funcionalitat bàsica inclou:

- **Provar les sis funcions de càlcul d'utilitat**, és a dir, les sis implementacions de UtilityFunction incorporades en el framework.
- **Crear accions** amb les diverses funcions de càlcul d'utilitat.
- **Implementar una classe que implementi Context** i la funció GetContextValue.
- **Calcular la pròxima acció** amb la funció CalculateBestAction, utilitzant el mètode de tria Best Action i sense inèrcia.

La resta de funcionalitats queden pendents de testejar.

10.2 Avaluació dels objectius del Framework

A continuació s'exposen els objectius del projecte del punt 4.1 d'aquest document i es valora, des del punt de vista d'un programador de videojocs, si s'han complert els objectius del projecte.

Facilitar la programació de Utility AI per a programadors de videojocs en un motor de jocs existent.

El framework facilita la programació de la Utility AI per a agents de videojocs de forma satisfactòria. Al utilitzar el framework no és necessari programar el funcionament intern de la AI, ja que aquest està controlat pel framework. No cal conèixer a fons el funcionament de la Utility AI per tal de programar la AI utilitzant el framework gràcies a la guia del punt 8 d’aquest document.

Objectiu assolit.

Permetre de personalitzar les funcions d’utilitat amb els valors d’entrada, que el framework s’encarregarà de recuperar del context quan els necessiti.

És possible decidir quins valors es volen utilitzar a les funcions de càlcul d’utilitat de les accions, i aquests valors s’obtenen correctament des del context.

Objectiu assolit.

Obtenir la millor acció que es pot portar a terme d’un conjunt d’accions, on cadascuna té una manera assignada per calcular la utilitat de l’acció.

Una vegada es tenen les accions codificades, és molt senzill fer el càlcul i que retorni la millor acció, dins d’un conjunt d’accions, que pot realitzar l’agent segons el context actual d’aquest. A més, el càlcul no modifica l’estat de les accions, permetent que les mateixes accions es puguin utilitzar tantes vegades com sigui necessari, per fer càlculs posteriors.

Objectiu assolit.

Incorporar el framework com a part del motor de jocs; facilitant el seu ús, instal·lació i portabilitat.

El framework no dona cap problema respecte a acoblar-lo amb el motor de jocs Unity, i es pot utilitzar correctament. Només ha calgut afegir el .dll a la carpeta corresponent de Unity per tal de poder utilitzar el codi del framework des dels scripts del joc.

Objectiu assolit.

Permetre d'utilitzar les funcions matemàtiques de càlcul d'utilitat que estan prèviament definides, sense que el programador les hagi d'implementar.

Hi ha sis funcions de càlcul d'utilitat definides en el framework, i aquestes cobreixen les necessitats bàsiques per programar una Utility AI. No només es poden utilitzar funcions prèviament definides sinó que aquestes són suficients per programar un agent amb el comportament desitjat, sense necessitat de programar cap funció matemàtica.

Objectiu assolit.

Poder incorporar una funció matemàtica de càlcul d'utilitat personalitzada si aquesta no està prèviament definida.

Segons la documentació i la guia del punt 8 d'aquest document, és possible incorporar funcions de càlcul d'utilitat personalitzades. I l'estructura del framework està pensada per treballar amb polimorfisme i afegir una funció d'utilitat personalitzada no hauria d'alterar-ne el funcionament. Però no s'ha utilitzat cap funció personalitzada durant la codificació del joc perquè no era necessari, i no es pot confirmar el seu correcte funcionament.

Objectiu no testejat.

Aconseguir independència entre els valors del context i els càlculs de la Utility AI.

A l'hora de programar la classe que implementa Context, s'ha pogut codificar lliurement, només era necessari tenir en compte el retorn del mètode GetContextValue. També és transparent al framework la nomenclatura dels valors, ja que aquesta s'introdueix a la creació de les funcions d'utilitat i no es modifica ni altera fins que s'utilitza per buscar el valor al context. Per tant, es pot afirmar que hi ha completa independència entre els valors del context i els càlculs de la Utility AI.

Objectiu assolit.

10.3 Problemes i millores

Durant la programació de la AI del personatge del videojoc, s'han trobat diversos problemes i inconvenients, i s'han trobat alguns aspectes que es podrien millorar. La següent llista mostra el recopilatori d'aquests problemes i, si s'ha trobat, una solució o millora proposada:

- **Les funcions d'utilitat són massa difícils d'ajustar**, perquè s'ha de considerar només l'interval de l'eix x entre 0 i 1. És necessari facilitar la forma d'ajustar-les, en especial en els casos de les funcions Logistic i Logit.

- La funció d'utilitat **ComposedUtilityFunction és incòmoda de codificar i poc intuïtiva**. Seria millor poder afegir les diverses funcions internes després de la creació en lloc de en el moment de creació, i s'hauria de modificar el funcionament dels punts de canvi entre funcions.

- **El retorn del mètode GetContextValue** de la interfície Context **no aporta facilitat a la codificació**. Un objecte amb els atributs Min, Max i Current seria més fàcil de programar i entenedor que l'array de double que es retorna actualment.

- No hi ha una forma de controlar, a l'hora de fer la tria de la pròxima acció, que **algunes accions sempre tinguin prioritat respecte a unes altres si sobrepassen una certa utilitat**. En un exemple, els humans es poden morir de gana, però no d'avorriment. Per tant, un agent amb una utilitat de l'acció *menjar* de 0.8, hauria de preferir menjar en lloc de mirar la tele encara que la utilitat de *mirar la tele* sigui de 0.9. Així es pot evitar que els agents donin prioritat a certes accions quan altres són més importants per a la seva supervivència.

10.4 Què queda pendent per una pròxima avaluació?

A continuació s'escriu una llista dels elements del framework que queden pendents de testejar en una pròxima avaluació:

- Comprovar el funcionament del framework si hi ha accions amb **inèrcia**.
- Codificar i testejar una **funció d'utilitat personalitzada**.
- Comprovar que el funcionament és l'esperat si **més d'un agent utilitza la mateixa AI** però utilitzen diferents Context.
- Provar el correcte funcionament dels mètodes de tria **Weighted Random** i **Random Best 3** de la classe BestActionCalculator.
- Testejar qualsevol **modificació** de les funcionalitats bàsiques del framework que es realitzin seguint els **nous requeriments**, en cas de modificació d'aquest.

11 Eines que s’han utilitzat

En aquest apartat es descriuen les principals eines que s’han utilitzat en el desenvolupament del projecte.

11.1 Microsoft Visual Studio

Visual Studio és un IDE de la companyia Microsoft, utilitzat per desenvolupar diferents tipus de programes, aplicacions web i aplicacions mòbils, entre altres. Per a més informació, consultar la referència [27].

Un dels principals motius pels quals s’ha decidit utilitzar Visual Studio, com ja s’ha mencionat, és perquè permetia tant el desenvolupament del framework com del joc per testejar-lo. El motiu decisiu ha estat que la llicència de l’edició Community és **gratuïta**, eliminant així qualsevol tipus de despesa per la llicència.

11.2 Unity

Unity és un motor de videojocs de la companyia Unity Technologies. Com es pot veure a la referència [28] és el motor de videojocs més ben posicionat juntament amb Unreal Engine [29]. Els dos permeten crear jocs per múltiples plataformes de forma molt senzilla al moment de fer la compilació del programa, i els dos tenen una llicència gratuïta.

A l’hora de decidir quin dels dos motors de videojocs utilitzar s’ha considerat el llenguatge típic dels scripts dels dos motors. I s’ha decidit utilitzar Unity perquè el llenguatge dels scripts és **C#**, que és molt similar al Java, i per tant ha resultat bastant ràpid i fàcil d’aprendre gràcies a l’experiència prèvia amb Java.

Per aprendre a utilitzar el motor de jocs Unity s’han seguit diverses guies de programació i explicacions dels diferents elements de la pàgina de *Unity Learn Tutorials* [30].

Per a més informació sobre el motor de jocs Unity, consulteu les referències [31] i [32].

11.3 Desmos

Desmos és una aplicació web i mòbil per calcular gràfics escrita en Javascript [19]. Per tal de realitzar els gràfics de la Taula 7.1, es va buscar una aplicació web que permetés mostrar els gràfics de les funcions.

Després de valorar diverses opcions, es va decidir utilitzar Desmos perquè és l’eina que donava més llibertat de caracteritzar els gràfics: afegir etiquetes, canviar el color de les funcions, afegir més d’una funció, afegir variables, etc.

Desmos també s’ha utilitzat per ajudar al disseny de la AI de la simulació, i per ajustar els valors dels paràmetres de les implementacions de UtilityFunction.

12 Conclusions

En aquest treball de fi de grau s'ha dissenyat i implementat un primer prototip del framework per facilitar la programació de Utility AI per a videojocs. També s'ha dissenyat i implementat una simulació per a poder fer una valoració d'aquest prototip des del punt de vista de l'usuari del framework.

S'han assolit tots els objectius del projecte menys un, que no s'ha arribat a provar en la simulació perquè durant el disseny d'aquesta no s'ha considerat necessari implementar una funció d'utilitat personalitzada. Per la naturalesa del treball, els objectius del producte i del client coincideixen amb els objectius del projecte, i per tant també s'han assolit tots excepte l'objectiu que no s'ha provat en la simulació.

L'eina Microsoft Visual Studio ha servit al seu propòsit i no ha aportat cap problema en el seu ús. Ha estat una bona tria com a eina de desenvolupament del framework perquè només ha estat necessari aprendre a utilitzar una eina de codificació per programar tant el framework com els scripts de la simulació. Visual Studio també permet triar el tipus de projecte durant la creació d'aquest, i això ha permès codificar el framework com a llibreria per facilitar-ne la importació des de la plataforma de videojocs (que és un dels objectius del projecte).

L'eina de treball Unity ha demostrat ser una eina molt potent, que justifica que sigui una de les més utilitzades en la programació de videojocs. Tot i que només s'han utilitzat la part gràfica de la UI i els scripts de programació, Unity ha facilitat enormement la programació de la simulació.

L'eina Desmos s'ha utilitzat tant per complementar la redacció d'aquesta memòria com per dissenyar i ajustar el comportament de la AI en la simulació. Tot i que s'ha triat perquè s'ha vist com l'eina més útil de les que s'han provat, ha resultat que algunes funcionalitats eren més complexes d'utilitzar del que semblava en un principi. Com a conseqüència s'ha hagut de dedicar més temps de l'esperat a l'hora d'utilitzar aquesta eina.

El framework ha demostrat complir amb l'objecte del projecte: facilitar la programació de Utility AI als desenvolupadors de videojocs. El principal punt fort del framework és la independència amb el codi del videojoc i l'estat d'aquest, perquè permet al programador del

videojoc programar l'estat del joc i/o el context del personatge com ell prefereixi i només cal que implementi un mètode d'una interfície per tal d'utilitzar el framework. El segon punt fort és el tractament de les funcions del càlcul d'utilitat com a objectes, perquè permet al programador de programar les seves funcions amb facilitat, i crear i canviar les funcions en temps d'execució, entre altres.

Però com ja s'ha indicat en l'avaluació del framework, aquest presenta diversos problemes. D'entre tots aquests, el més destacable és la dificultat d'ajustar les funcions d'utilitat, ja que coincideix amb un dels desavantatges de la Utility AI. La conseqüència d'aquest problema durant les proves amb la simulació ha estat que el personatge sempre es moria. Normalment es moria per l'estrès, però si es modificaven els valors d'alguna funció perquè no es morís d'estrès, aleshores es moria de gana. En la Figura 12.1 es mostra un exemple de l'execució.

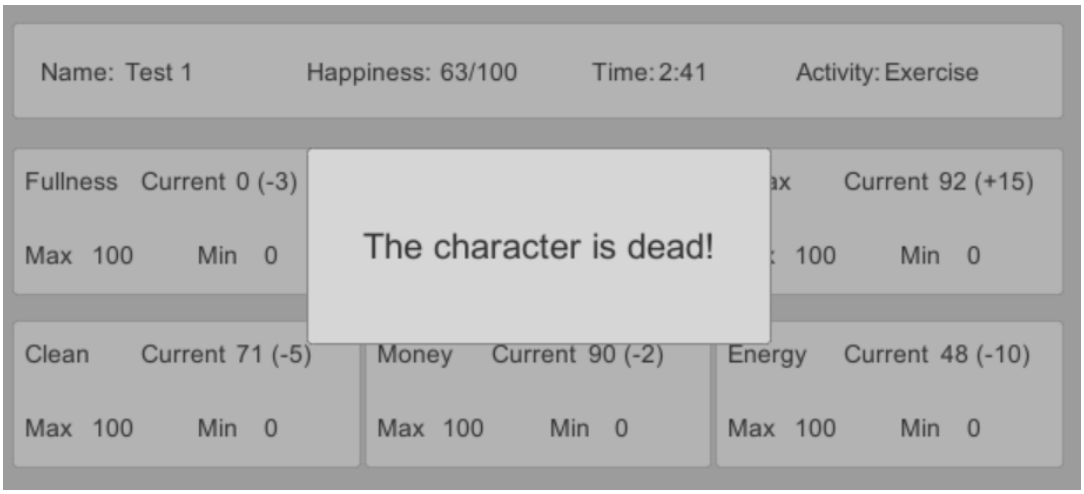


Figura 12.1 Resultat de l'execució de la simulació.

El resum valoratiu del projecte és que el resultat ha estat satisfactori, però que s'hauria de seguir treballant en el projecte per tal de millorar el framework i eliminar els principals problemes que presenta en aquest primer prototip.

A petita incursió personal, vull remarcar que ha estat un repte el disseny del framework per tal d'aconseguir la independència amb el joc. I també que ha estat divertit, en especial a l'hora d'ajustar les funcions de càlcul d'utilitat de la simulació, encara que al final el personatge sempre es morís.

13 Possibles ampliacions

A continuació s'exposa una llista de les possibles ampliacions a realitzar en el framework. En aquesta llista s'han d'afegir els problemes i ampliacions de l'apartat 10.3 d'aquest document.

- Aplicar un mecanisme de control del temps en el càlcul de la pròxima acció, per fer que s'interrompi el càlcul si es supera el temps establert. És un requeriment no funcional que no s'ha codificat en aquesta iteració de prototipatge, consultar l'apartat 6.2 d'aquest document.

- Escriure en un log la llista de les accions amb les utilitats corresponents, en cada una de les crides a la classe BestActionCalculator del framework, per tal que el programador de la AI pugui veure els valors obtinguts i pugui ajustar amb més facilitat la AI.

- Permetre que la UtilityFunction de les Action del framework es pugui canviar amb un mètode set. D'aquesta manera es pot modificar el comportament de la AI fàcilment durant l'execució, en cas que el programador de la AI així ho vulgui.

- Proporcionar una eina/funcionalitat dins del framework per visualitzar les corbes d'utilitat de les UtilityFunction, ja sigui per separat o més d'una alhora. D'aquesta manera es facilita l'ajust de les funcions per als programadors de la AI, i no és necessari utilitzar una eina externa al framework.

14 Bibliografia

[1] González Sánchez, J. L., Zea, N. P., & Gutiérrez, F. L. (2010). From Usability to Playability: Introduction to Player-Centred Video Game Development Process. Proceedings of First International Conference, HCD 2009 (Held as Part of HCI International), San Diego, CA, USA.

[2] Finite-State Machines: Theory and Implementation [en línia] [consulta: 9 novembre de 2017]. Disponible a https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867

[3] Finite-state Machine [en línia] [consulta: 9 novembre de 2017]. Disponible a https://en.wikipedia.org/wiki/Finite-state_machine

[4] Game Programming Patterns – State [en línia] [consulta: 9 novembre de 2017]. Disponible a http://gameprogrammingpatterns.com/state.html

[5] Behavior trees for AI: How they work [en línia] [consulta: 10 novembre de 2017] Disponible a https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

[6] Apex Utility AI [en línia] [consulta: 13 novembre de 2017]. Disponible a http://apexgametools.com/products/apex-utility-ai-2/

[7] Crystal AI [en línia] [consulta: 13 novembre de 2017]. Disponible a https://github.com/igiagkiozis/CrystalAI

[8] Mark, D., Lewis, M. (2015). Building a Better Centaur: AI at Massive Scale [Vídeo]. Recuperat de https://www.gdcvault.com/play/1021848/Building-a-Better-Centaur-AI

[9] Mark, Dave: *Behavioral Mathematics for Game AI*. Course Technology, Boston, USA, 2009, primera edició. ISBN: 978-1-58450-684-3.

[10] An Introduction to Utility Theory [en línia] [consulta: 11 novembre de 2017]. Disponible a http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter09_An_Introduction_to_Utility_Theory.pdf

[11] At-a-glance funcions for modelling utility-based game AI [en línia] [consulta: 11 novembre de 2017]. Disponible a https://alastaira.wordpress.com/2013/01/25/at-a-glance-functions-for-modelling-utility-based-game-ai/

[12] AI: Utility vs Behavior Trees [en línia] [consulta: 11 novembre de 2017]. Disponible a https://discourse.urho3d.io/t/ai-utility-vs-behaviortrees/3092

[13] GameDev.net [en línia] [consulta: 18 Novembre de 2017]. Disponible a https://www.gamedev.net/

[14] Gamasutra – The Art & Business of Making Games [en línia] [consulta: 18 Novembre de 2017]. Disponible a http://www.gamasutra.com/

[15] Ludust – Game Development Community [en línia] [consulta: 18 Novembre de 2017]. Disponible a http://ludust.com/

[16] Team up – Collaborate, Create and Innovate [en línia] [consulta: 18 Novembre de 2017]. Disponible a https://www.teamups.net/

[17] Game Developers Conference [en línia] [consulta: 18 Novembre de 2017]. Disponible a http://www.gdconf.com/

[18] SDLC Software Prototype Model [en línia] [consulta: 5 Novembre de 2017]. Disponible a https://www.tutorialspoint.com/sdlc/sdlc_software_prototyping.htm

[19] Desmos | Beautiful, Free Math [en línia] [consulta: 4 Març de 2018]. Disponible a https://www.desmos.com/

[20] Unity – Manual: Managed Plugins [en línia] [consulta: 10 Març de 2018]. Disponible a https://docs.unity3d.com/Manual/UsingDLL.html

[21] Unity – Manual: Native Plugins [en línia] [consulta: 10 Març de 2018]. Disponible a https://docs.unity3d.com/Manual/UsingDLL.html

[22] Mike Geig (2014). Writing Plugins – Unity [Vídeo]. Recuperat de https://unity3d.com/es/learn/tutorials/topics/scripting/writing-plugins

[23] User Interface (UI) – Unity [en línia] [consulta: 5 Maig de 2018]. Disponible a https://unity3d.com/es/learn/tutorials/s/user-interface-ui

[24] Unity – Manual: Variables and the Inspector [en línia] [consulta: 5 Maig de 2018]. Disponible a https://docs.unity3d.com/Manual/VariablesAndTheInspector.html

[25] Unity – Scripting API: MonoBehaviour [en línia] [consulta: 3 Maig de 2018]. Disponible a https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

[26] Dictionary<TKey, TValue> Class [en línia] [consulta: 3 Maig de 2018]. Disponible a https://msdn.microsoft.com/en-us/library/xfhwa508(v=vs.110).aspx

[27] IDE de Visual Studio, editor de código, Team Services y Mobile Center [en línia] [consulta: 20 Maig de 2018]. Disponible a https://www.visualstudio.com/

[28] Best Game Engine Software [en línia] [consulta: 20 Maig de 2018]. Disponible a https://www.g2crowd.com/categories/game-engine

[29] What is Unreal Engine 4 [en línia] [consulta: 4 Març de 2018]. Disponible a https://www.unrealengine.com/en-US/what-is-unreal-engine-4

[30] Unity Learn Tutorials [en línia] [consulta: 4 Març de 2018]. Disponible a https://unity3d.com/es/learn/tutorials

[31] Unity [en línia] [consulta: 4 Març de 2018]. Disponible a https://unity3d.com/es/

[32] Unity (game engine) – Wikipedia [en línia] [consulta: 4 Març de 2018]. Disponible a https://en.wikipedia.org/wiki/Unity_(game_engine)#cite_note-1

[33] Jugabilidad [en línia] [consulta: 8 novembre de 2017]. Disponible a https://es.wikipedia.org/wiki/Jugabilidad

[34] Behavior tree (artificial intelligence, robòtics and control) [en línia] [consulta: 10 novembre de 2017] Disponible a https://en.wikipedia.org/wiki/Behavior_tree_(artificial_intelligence,_robotics_and_control)

[35] An Introduction to Behavior Trees [en línia] [consulta: 10 novembre de 2017] Disponible a http://blog.renatopp.com/2014/07/25/an-introduction-to-behavior-trees-part-1/