Universitat de Girona
Escola Politècnica Superior

Treball final de grau

Estudi: Grau en Enginyeria Informàtica

Títol:

Desenvolupament d'un videojoc de waterpolo

Document: Memòria

Alumne: Arià Serra Felip

Tutor: Dr. Gustavo Patow
Departament: Informàtica i matemàtica aplicada
Àrea:LSI

Convocatòria (mes/any) Juny/2015

Índex de continguts

Capítol 1. Introducció, motivacions, propòsit i objectius del projecte

En aquest capítol introductori explicarem les motivacions del projecte de final de grau, així com els propòsits i objectius. En l'últim apartat del capítol es presentarà un breu resum dels diferents apartats d'aquesta memòria.

1.1. Introducció

Un videojoc és un joc electrònic interactiu, en un suport informàtic, per ser executat en un dispositiu electrònic, amb el qual una persona interactua davant d'una pantalla. L'origen es remunta al 1958, amb el joc *Tennis for Two*, més conegut, més tard com a *Pong*, que va ser el primer videojoc creat.

A partir d'aquí es van anar creant nous videojocs, i la indústria va anar creixent. Es van desenvolupar les maquines recreatives, videoconsoles i videoconsoles portàtils. Actualment la indústria dels videojocs és un sector que està a l'alça i que competeix directament amb la indústria del cinema i de la música, guanyant més que la industria del cinema i la musica juntes.

Les prediccions del futur de la industria dels videojocs és que les tauletes electròniques i els telèfons intel·ligents guanyaran més pes.

Figura 1.1: Mercat global dels videojocs. Font: Newzoo 2013.

Els videojocs esportius són un gènere que simula el joc tradicional de l'esport. És un gènere extremadament popular pels diferents jocs com futbol, futbol americà, boxa, bàsquet, cotxes, etc. Aquest gènere conté diferents subgèneres que serien: simulació, direcció/mànagers i fantasia.

En el cas que ocupa aquest projecte, es tractarà d'un videojoc esportiu col·lectiu on es controlarà un únic jugador i el sistema gestionarà els altres jugadors.

1.2. Motivacions personals

Des de fa molt de temps he encarat la meva formació professional al món dels videojocs. Per tal de d'assolir aquest objectiu, vaig triar com a etapa formativa universitària un grau en enginyeria informàtica, ja que en aquell temps no hi havia cap grau encarat específicament als videojocs. Més endavant es van introduir aquests graus, primer a Barcelona ja fa 2-3 anys i llavors l'any que ve s'estrena el grau de videojocs a la UdG.

Llavors per finalitzar el grau, he encarat el treball de fi de grau en aquest sector. Amb això ho vaig ajuntar amb el meu hobby principal, el waterpolo que des de petit que el practico. A més, l'altre hobby que tinc són els videojocs per ordinador. A partir d'aquí he ajuntat les dues coses que m'agraden més per fer un projecte interessant.

També, com que m'interessa aprendre el funcionament dels motors de videojocs i així com habituar-me a totes les seves funcionalitats, aquest projecte és una manera d'aprendre un motor de videojocs a fons.

1.3. Motivacions de projecte

A part de les motivacions personals, una altre motivació és que els videojocs de waterpolo no s'han explotat comercialment, només hi ha jocs recreatius o mini jocs online. No trobem que hi hagi cap videojoc amb grans acabats o que hagi tingut una gran ressò internacionalment amb el possible mercat que hi pot haver a Itàlia, Sèrbia i Hongria.

1.4. Propòsits

El propòsit del projecte és desenvolupar un videojoc en 3D en tercera persona de l'esport col·lectiu del waterpolo. Això comporta en crear un joc que tingui una intel·ligència artificial encarada a moure un equip de jugadors, així com crear un sistema el qual permetés que el jugador controlés només un jugador de waterpolo i interactuar amb els altres jugadors de l'equip. Aquest videojoc s'implementarà a partir d'un motor de videojocs gratuït.

Un altre factor que s'introduirà al videojoc és que com, que el waterpolo és un joc que té un conjunt de tàctiques diferents, s'ha crear un sistema per configurar jugades tàctiques de waterpolo, per donar al jugador una manera de personalitzar el joc al seu estil, ja que no podrà controlar directament els jugadors del propi equip.

1.5. Objectius

L'objectiu primordial d'aquest projecte és desenvolupar un videojoc de waterpolo. Podem organitzar les tasques del projecte de la següent manera:

- Planificació del joc: definició dels elements d'interacció i les regles de joc.
- Estudi del motor gràfic Unity3D i el llenguatge C#.
- Estudi de l'API de Unity3D.
- Implementació de les accions possibles pel jugador i d'interacció amb l'entorn.
- Disseny i implementació de la Intel·ligència Artificial dels dos equips que participen al partit.
- Verificació i proves dels algorismes desenvolupats.
- Arrodoniment i finalització de la documentació del treball realitzat.

1.6. Capítols de la memòria

Aquesta memòria del projecte de fi de grau està estructurada en 15 capítols, els quals recullen tota la informació del projecte i les conclusions finals.

- Capítol 1. Introducció, motivacions, propòsit i objectius del projecte

S'hi explica el perquè del desenvolupament del projecte, així com els objectius proposats i la organització del document.

- Capítol 2. Estudi de viabilitat

S'especifiquen els paràmetres que fan possible el desenvolupament del projecte.

- Capítol 3. Metodologia

En aquest capítol es defineix la metodologia utilitzada, explicant les raons de la seva utilització.

- Capítol 4. Planificació

Es defineix l'estratègia seguida per desenvolupar i finalitzar els objectius del projecte plantejats així com la temporització.

- Capítol 5. Marc de treball i conceptes previs

Aquest capítol explica els diversos aspectes que envolten el desenvolupament del projecte per permetre entendre millor el projecte.

- Capítol 6. Requisits del sistema

Conté els diferents requisits del sistema pel complet funcionament de l'aplicació.

- Capítol 7. Estudis i decisions

Es descriu el maquinari, llibreries i programari utilitzats durant el desenvolupament del projecte.

- Capítol 8. Anàlisi i disseny del sistema

En aquest capítol es descriu el disseny utilitzat en el sistema, així com les especificacions, esquemes d'implementació, classes i mètodes del projecte.

- Capítol 9. Implementació i proves

S'explica com s'ha implementat i provat les diferents parts del sistema.

- Capítol 10. Implantació i resultats

Descriu fins a quin punt s'han implementat completament o no els diferents objectius del projecte.

- Capítol 11. Conclusions

S'exposen les conclusions obtingudes al finalitzar el projecte.

- Capítol 12. Treball futur

Es descriuen possibles millores i noves funcionalitats per a noves versions del videojoc.

- Capítol 13. Bibliografia

Aquest capítol conté les referències biogràfiques utilitzades pel desenvolupament del projecte.

- Capítol 14. Annexos

Inclou un seguit d'ampliacions per a una millor comprensió del projecte.

- Capítol 15. Manual d'usuari i/o instal·lació

L'últim capítol inclou un manual que explica el funcionament i instal·lació per tal d'utilitzar correctament el videojoc.

Capítol 2. Estudi de viabilitat

En aquest capítol s'explicarà la viabilitat del projecte.

2.1. Recursos necessaris per desenvolupar el projecte

Pel desenvolupament d'aquest projecte no ha estat necessària d'una gran infraestructura. Els ordinadors utilitzats han estat un ordinador de sobretaula i un ordinador portàtil. Dependent de la situació s'ha utilitzat un o altre ordinador.

En les següents taules es mostren les especificacions tècniques de cada equip utilitzat:

	Ordinador sobretaula	Ordinador portàtil
CPU	Intel Core i7 920	Intel Core i7 4720HQ
Placa base	Gigabyte EX58-UD5	MSI MS-16GF
Ram	Kingston 4GB 1333MHz	Kingston 8GB 1600MHz
Targeta gràfica	NVIDIA GeForce 9800 GTX+ 1GB	NVIDIA GeForce GTX 850M 1GB
Disc Dur	Western Digital Green 2TB	HGST 1TB
Sistema operatiu	Windows 8.1 Pro x64	Windows 8.1 Pro x64

Figura 2.1: Especificacions tècniques dels equips utilitzats. Font pròpia.

Per la part de programari s'ha utilitzat *Unity 5 Personal Edition* per el desenvolupament del videojoc, el qual no ha requerit de cap inversió econòmica per la seva llicència. Un altre software utilitzat ha estat Visual Studio 2013.

2.2. Pressupostos inicials

Aquest projecte no requereix de cap inversió inicial.

Per a un futur, si es decideix de publicar-ho a *Steam Greenlight*, només requeriria d'uns 100€ i llavors pagar un 30% dels beneficis a *Steam* un cop es comencés a vendre. El programa utilitzat pel desenvolupament del videojoc és *Unity 5*, que no necessita cap tipus de compra de llicència per a l'ús donat. Llavors un cop es distribuís s'hauria de pagar llicencia professional si s'excedís dels 100000$ anuals.

2.3. Recursos humans

L'equip òptim per a la realització del projecte es requeria d'un cap de projecte, un programador, un dissenyador, un analista i un artista. Amb aquest conjunt de persones es podien definir i dividir les tasques en funció del rol i de les necessitats del moment. En aquest cas el conjunt de rols han estat realitzats per jo mateix, amb l'ajuda del tutor del projecte.

La tasca del programador seria introduir totes les idees del videojoc i transformar-ho amb codi per tal que la funcionalitat treballés correctament.

La tasca d'un dissenyador seria la de definir l'estructura del codi així com els algoritmes utilitzats.

La tasca de l'analista seria triar quina és la millor eina de desenvolupaments, investigar les tendències del mercat i futures tecnologies.

La tasca de l'artista seria crear els escenaris, els models dels personatges, així com les animacions. Altres tasques serien crear la banda sonora i els efectes sonors.

La tasca de cap de projecte seria la de definir prioritats en el projecte i dirigir l'equip.

2.4. Viabilitat tecnològica

El projecte no requereix d'una tecnologia molt avançada. Amb el meu ordinador que té 6 anys ja podia manejar amb facilitat el programa. Per tant, no es necessari un equip molt punter tecnològicament.

2.5. Viabilitat econòmica

La viabilitat econòmica estimada del projecte es pot dividir en dues parts, els costos humans i els costos de maquinària i programari.

Els costos humans serien els següents partint que els costos per hora són els següents:

- Costos analista: 18 €/hora.
- Costos dissenyador: 15 €/hora.
- Costos artista: 15€/hora
- Costos programador: 13 €/hora.

Tasca		Hores	Cost
Investigació	Analista	40	720
Estudi d'Unity3D	Dissenyador	70	1050
Disseny d'algoritmes	Dissenyador	70	1050
Disseny de l'estructura	Dissenyador	5	75
Implementació d'algoritmes	Programador	140	1820
Proves i optimitzacions	Programador	50	650
Banda sonora i efectes	Artista	30	450
Creació models i animació	Artista	140	2100
Creació d'escenaris	Artista	80	1200
Memòria	Analista	80	1440
Total		**705**	**9115€**

Figura 2.2: Costos de recursos humans.

En la part de costos de maquinària i programari s'haurien de tenir en compte els costos dels ordinadors utilitzats i les llicències dels programes utilitzats.

Programari	Preu unitari	Preu
Microsoft Windows 8.1	100	200
Autodesk Maya	4030	4030
Adobe Photoshop	435	435
Ordinadors		
Sobretaula	1000	333
Portàtil	950	317
Total		**5315€**

Figura 2.3: Costos de maquinària i programari.

En conclusió el preu estimat per al desenvolupament del projecte seria d'uns 14430€.

Capítol 3. Metodologia

Per al desenvolupament del projecte no s'ha utilitzat una metodologia estàndard, sinó que s'ha usat una metodologia personalitzada plantejada i definida amb el tutor.

Els diferents passos d'aquesta metodologia serien els següents:

1. Triar les eines i llenguatge de programació.
2. Aprendre el llenguatge i les eines escollides.
3. Dividir el projecte en diferents parts.
4. Planificar el treball segons les diferents parts que s'han dividit el projecte.
5. Desenvolupar cada part del projecte.
6. Comprovar el correcte funcionament de cada part del projecte. Si no és correcte, tornaríem al punt anterior.
7. Unir les diferents parts del projecte. Si trobéssim errors es tornaria al punt 5.
8. Generació de diferents escenaris per al correcte funcionament del projecte. Si trobéssim errors es tornaria al punt 5.
9. Matisar la documentació escrita durant els desenvolupament del projecte.

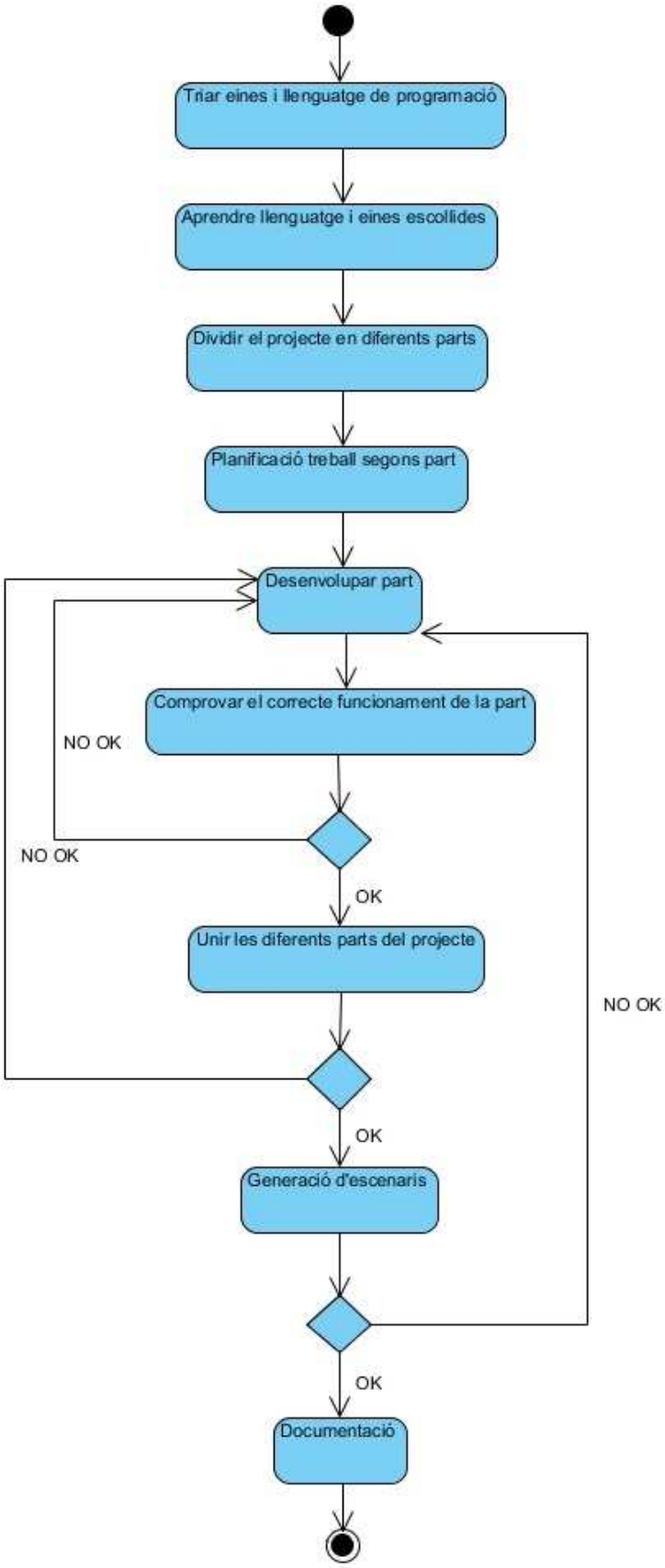


Figura 3.1: Metodologia utilitzada.

Capítol 4. Planificació

En aquest capítol es descriuran les diferents tasques del projecte i el temps estimat per a cada tasca.

4.1. Pla de treball

El pla de treball del videojoc es divideix en diferents tasques, que a grans trets serien, planificació, introducció al motor de joc, implementació i documentació.

4.2. Tasques planificades

4.2.1. Planificació del joc

En aquesta tasca es defineix el joc, les regles i les diferents accions del jugador.

4.2.2. Estudi del motor gràfic Unity3D i el llenguatge C#

S'estudia el funcionament del motor de videojoc *Unity3D*, el qual permet diferents llenguatges de programació com *JavaScript*, *Boo* i *C#*. Es tria el *C#* per la gran semblança que té amb Java i C++.

4.2.3. Estudi de l'API de Unity3D

En aquesta etapa s'estudia l'API del motor per habituar-se a les classes i mètodes claus per al desenvolupament del videojoc.

4.2.4. Implementació de les accions possibles pel jugador i d'interacció amb l'entorn

En aquesta etapa s'implementava els diferents moviments i accions, com xutar i passar, i que el resultat d'aquest interactués amb l'entorn.

4.2.5. Disseny i implementació de la Intel·ligència Artificial dels dos equips que participen al partit

Aquesta etapa es dissenya i s'implementa la intel·ligència artificial, així com un lector de jugades. Cada jugador no controlat es mourà i realitzarà accions a partir de d'intel·ligència artificial.

4.2.6. Verificació i proves dels algorismes desenvolupats

A mesura que s'anaven finalitzant els diferents algorismes, es comprovaven el correcte funcionament.

4.2.7. Arrodoniment i finalització de la documentació del treball realitzat

A mesura que s'avançava el projecte, s'anava documentant tot el procés, idees i mètodes que es requerien. Finalment s'ha arrodonit i empaquetat tot el conjunt de la documentació en la memòria.

4.3. Temps estimat

Es va estimar que el projecte duraria uns 9 mesos i la distribució va ser la que es pot veure a la figura 4.1.:

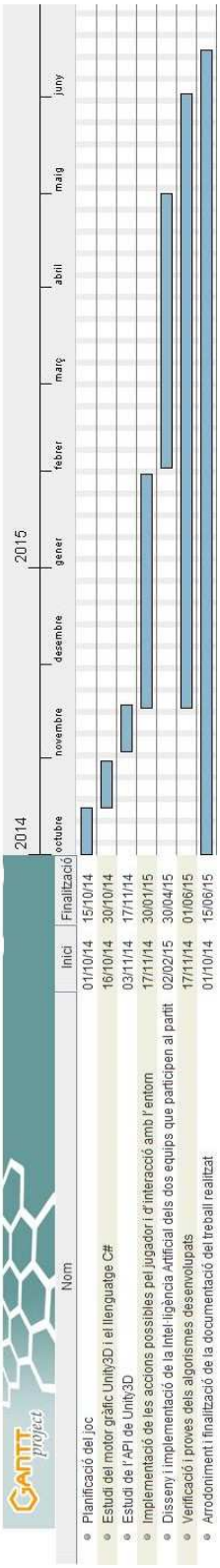


Figura 4.1: Planificació estimada.

4.4. Resultats esperats de cada tasca

4.2.1. Planificació del joc

L'objectiu de la planificació és posar un temps adequat per a cada tasca a fer del joc.

4.2.2. Estudi del motor gràfic Unity3D i el llenguatge C#

S'espera obtenir coneixement del motor gràfic i del llenguatge *C#* per poder implementar fàcilment els diferents mètodes requerits.

4.2.3. Estudi de l'API de Unity3D

Saber aprofitar el màxim possible l'API per estalviar temps i optimitzar el codi.

4.2.4. Implementació de les accions possibles pel jugador i d'interacció amb l'entorn

Un cop finalitzat, poder moure per l'escenari i poder xutar i fer passes sense error.

4.2.5. Disseny i implementació de la Intel·ligència Artificial dels dos equips que participen al partit

L' intel·ligència artificial de cada equip faci que els jugadors es moguin correctament pel camp de joc i que creïn jugades per marcar gol.

4.2.6. Verificació i proves dels algorismes desenvolupats

Obtenir un codi net d'errors i amb tots els algorismes provats.

4.2.7. Arrodoniment i finalització de la documentació del treball realitzat

Finalitzar la memòria indicant totes les decisions preses i canvis fets. La memòria s'ha de poder llegir i comprendre amb facilitat.

Capítol 5. Marc de treball i conceptes previs

5.1. Motors de videojocs

Per desenvolupar un videojoc, normalment s'utilitza un motor de videojocs. Un motor de videojoc és un programari que permet la creació, disseny i representació d'un videojoc. Estan formats per diferents parts, com la renderització, física, col·lisió, so, scripting, animació, intel·ligència artificial, xarxa, streaming, gestió de memòria, multifils, localització i gràfic d'escenes. Aquest programari està pensat per que els desenvolupadors de videojocs no s'hagin de preocupar directament de tocar les llibreries gràfiques del sistema operatiu.

Tanmateix, els motors de videojoc no solament serveixen per fer videojocs, sinó que també serveixen per fer visualitzacions arquitectòniques, simulacions d'entrenament, eines de modelat i simulació física per animacions i escenes realistes de cinema.

5.2. Exemples de motors de videojocs

5.2.1. CryEngine

Figura 5.1: Logo CryEngine.

CryEngine és un motor de videojocs creat per l'estudi *Crytek*. El seu primer videojoc va ser *Far Cry*, joc actualment sota llicència de *Ubisoft*. Aquest motor de videojocs té un simulador d'entorns y materials dels més potents. Però, en contrapartida, és un motor no indicat per a desenvolupadors iniciats perquè té una interfície complicada, documentació escassa i un sistema de programació per a experts.

5.2.2. Unreal engine

Figura 5.2: Logo Unreal engine.

És un motor desenvolupat per *Epic Games*, creadors de la saga *Gears of War*. És un motor indicat per a jocs de consoles i d'ordinadors. La gran diferència d'aquest motor sobre els altres, és que té un sistema de disseny nodal, indicat per a usuaris sense gaire coneixement de programació que vulguin fer servir funcions avançades del sistema. A més té una gran varietat de tutorials i molt de feedback d'altres usuaris.

També és un motor utilitzat per a la simulació de construccions, conducció, previsualització de pel·lícules i de generació de terrenys utilitzats per la NASA.

5.2.3. Id tech

Figura 5.3: Logo Id tech.

Software desenvolupat per *id Software*, el qual s'ha fet la saga *Doom*. Les versions anteriors a la 5 són sota llicència GNU GPL. En la última versió passa a ser propietària fins al llançament de la nova versió de *Id tech*. Una de les propietats que té aquest motor és que suporta unes textures amb una resolució de 128000 x 128000 píxels, anomenat *Virtual Texturing*. Aquest motor automàticament adapta les limitacions de les textures per les diferents plataformes, estalviant així una feina extra al desenvolupador.

5.2.4. Game Maker: Studio

Figura 5.4: Logo GameMaker: Studio.

Anteriorment anomenat *Animo* o *Game Maker*, és un motor creat per *Mark Overmars* i està programat amb *Delphi*. És un motor que està indicat per a desenvolupadors amb uns coneixements nuls de programació, ja que utilitza un sistema de arrossegar i deixar anar objectes. També té un conjunt de biblioteques amb les accions estàndard, que serien per fer el moviment de personatges, etc. Aquesta biblioteca es pot engrandir amb noves accions creades per als usuaris. Per a usuaris més experimentats existeix un llenguatge de programació *Game Maker Language* que permet personalitzar i engrandir les seves característiques.

5.3. Motor triat

Figura 5.5: Logo Unity.

El motor triat per al desenvolupament d'aquest joc és Unity, desenvolupat per *Unity Technologies*. És un motor utilitzat pels desenvolupadors independents perquè està indicat per a plataformes mòbils, web i multi plataforma en general. Té la major comunitat de desenvolupadors que exposen dubtes i solucions per la web. Disposa d'un gran contingut gratuït i de pagament tant en internet com a la botiga virtual. Permet programar en C#, Boo i JavaScript. La versió actual és gratuïta per a desenvolupament, però, si s'arribés a vendre, llavors s'hauria de pagar un tant per cent de la facturació a *Unity Technologies*.

Les raons per la seva utilització en el projecte és, per una banda la gratuïtat inicial d'utilització amb un accés a totes les eines imprescindibles per al desenvolupament del joc. La versió de pagament inclou una llicencia d'ús per un equip de desenvolupadors, analítiques, gestió al núvol, cap que influeixi directament a l'objectiu del desenvolupament del videojoc.

Una altre raó és la gran comunitat de desenvolupadors i la gran quantitat de fòrums que ajuden a poder avançar i no quedar-se encallat en un problema trobat.

5.4. Waterpolo

Figura 5.6: Partit de waterpolo.

És un esport d'equip practicat en un recinte d'aigua, com una piscina. Un partit de waterpolo està configurat per dos equip que només poden tenir 7 jugadors dintre l'aigua, un d'ells com a porter. La finalitat és introduir la pilota a la porteria rival. Cada equip està format fins un màxim de 13 jugadors, dos dels quals són porters.

Un partit té una duració de 4 parts, de 8 minuts cada una, i que els entrenadors poden demanar un temps mort per part de 1 minut de durada. Cada equip disposa de 30 segons per construir una jugada per introduir la pilota a la porteria contraria.

Capítol 6. Requisits del sistema

En aquest capítol es descriuen els requisits que ha de complir el sistema, tant funcionals com no funcionals.

Els requeriments funcionals són els serveis que ofereix l'aplicació independentment de la implementació. En canvi els requeriments no funcionals són les restriccions imposades pel client o pel problema.

6.1. Requeriments funcionals

Els requeriments de l'aplicació serien:

- Iniciar la partida.
- Moure pel camp de joc.
- Interactuar amb els companys del teu equip.
- Xutar o passar la pilota.

6.2. Requeriments no funcionals

Bàsicament els requeriments no funcionals es refereixen a la disponibilitat de recursos, seguretat o interfícies externes. Per la part de seguretat, no hi ha cap requisit de control d'accés al programa. El programa no guarda cap tipus de dades de caràcter confidencial.

La implementació i les proves del videojoc s'han dut a terme en els ordinadors següents:

	Ordinador sobretaula	Ordinador portàtil
CPU	Intel Core i7 920	Intel Core i7 4720HQ
Placa base	Gigabyte EX58-UD5	MSI MS-16GF
Ram	Kingston 4GB 1333MHz	Kingston 8GB 1600MHz
Targeta gràfica	NVIDIA GeForce 9800 GTX+ 1GB	NVIDIA GeForce GTX 850M 1GB
Disc Dur	Western Digital Green 2TB	HGST 1TB
Sistema operatiu	Windows 8.1 Pro x64	Windows 8.1 Pro x64

Figura 6.1: Especificacions tècniques dels equips utilitzats.

Capítol 7. Estudis i decisions

A continuació s'explicaran les llibreries i programes utilitzats per desenvolupar el projecte, així com el sistema operatiu utilitzat.

7.1. Sistema operatiu

El sistema operatiu utilitzat ha estat Windows 8.1 Pro amb una arquitectura de 64 bits. La raó de la seva utilització és perquè és el sistema operatiu que vaig servir normalment per treballar als diferents ordinadors que utilitzo.

7.2. Programari utilitzat

En aquest apartat es descriuen els diferents programes utilitzats per fer el projecte.

7.2.1. C#

És un llenguatge de programació orientat a objectes. Va ser desenvolupat i estandarditzat per *Microsoft* com a part de *.NET*. La sintaxis deriva de *C/C++* i s'assembla amb *Java*. També és anomenat C Sharp, per la traducció musical anglesa de #. A més que sembla que siguin 4 + enganxats.

Inclou moltes llibreries i a més les millora de les versions anteriors. També inclou llibreries d'altres llenguatges, adaptades a l'estructura i sintaxi.

La raó de la utilització és que *Unity* només permet programar 3 llenguatges: *Boo*, *JavaScript* i *C#.* D'aquests tres llenguatges, *C#* és la més semblant als llenguatges que he fet servir. D'aquesta manera podré dedicar més temps al projecte, que aprendre un nou llenguatge.

7.2.2. Visual Studio 2013

Figura 7.1: Logo Visual Studio 2013.

És un entorn de desenvolupament integrat a mans de la companyia *Microsoft*. Permet desenvolupar aplicacions, llocs i aplicacions web amb diversos llenguatges com *Visual C++, Visual C#, Visual J#, ASP.NET i Visual Basic .NET*. També té extensions que suporten altres llenguatges.

La utilització d'aquest IDE i no el *MonoDevelop* que porta el paquet de *Unity* és perquè *MonoDevelop* no és gaire estable, molts de cops es parava i es necessitava reiniciar el programa per poder continuar desenvolupant. A més, per la part de depurar el codi, mostra molt malament les diferents variables i a més no té una drecera de teclat per avançar línia a línia. Per això em vaig decantar a Visual Studio que és molt estable i a més per depurar a *Unity* només cal instal·lar un complement fàcil de trobar per internet.

7.2.3. Unity 5

Figura 7.2: Logo Unity.

La tria d'Unity sobre altres motors gràfics és per la seva gran comunitat i pel seu gran grau de personalització que ofereix. També perquè moltes de les accions que es poden programar en un script.

A l'editor de l'Unity es pot construir tota la part del videojoc, excepte els scripts que necessiten un IDE.

Per desenvolupar un videojoc en Unity, cal dividir-lo per escenes, on cada escena pot representar qualsevol part del joc, com el menú d'inici, un nivell o una part del joc.

Unity permet tenir diferents vistes del videojoc, tant en 2D com en 3D. També, a partir del ratolí i amb una combinació de tecles, permet moure per l'escenari.

7.2.3.1. Conceptes essencials

7.2.3.1.1. GameObject

A l'escenari es poden posar diferents objectes, els quals es poden posicionar, rotar i escalar. Aquest objectes, anomenats *GameObject*, estan composats per un conjunt de *Components*.

Aquesta aplicació també permet crear *Prefabs,* que són *GameObjects* guardats a disc que tenen totes les components i propietats inicialitzades. Per tant, si un *GameObject* es repeteix a l'escenari, es pot guardar com a *Prefab* i llavors inserir-ho a l'escena, tant cops com siguin necessaris, amb tot inicialitzat, per tal de facilitar les tasques al programador.

7.2.3.1.2. Component

Aquests poden ser: objectes 3D, objectes 2D, llum, interfície d'usuari, àudio, càmera, etc. Com a mínim cada *GameObject* ha de tenir la *Component Transform*, que defineix la posició, rotació i escala. Aquests Components es poden afegir en temps d'execució o no. Permeten afegir el comportament d'aquell *GameObject* en concret.

7.2.3.1.3. Camera

És un dispositiu el qual el jugador veu el món. Es pot manipular la càmera per tal de fer efectes únics.

7.2.3.1.4. Mesh Filter

És un component que serveix per agafar una malla dels assets, que és un model en 2D o 3D, i els passa al *Mesh Renderer* per mostrar-ho per pantalla

7.2.3.1.5. Mesh Renderer

Component que agafa la geometria del *Mesh Filter* i la renderitza a la posició definida a la component *Transform*.

7.2.3.1.6. Collider

Component que serveix per delimitar quines parts del objecte poden interactuar amb altres objectes amb *Colliders* i fer alguna cosa a partir d'aquí. Per tal de guanyar rendiment, no s'utilitza la mateixa malla utilitzada per crear l'objecte, si aquest conté molts de polígons.

7.2.3.1.7. Rigidbody

Component que permet actuar sota el control de la física. Pot rebre forces i torçar-se per fer que els objectes es moguin en un sentit realista.

7.2.3.1.8. Material

Component que s'encarrega de l'aparença visual. Aquest pot estar controlat per diferents shaders i normalment conté una textura.

7.2.3.1.9. Textura

És una imatge que transforma visualment un objecte o una GUI.

7.2.3.1.10. Character controller

Component que principalment serveix per controlar, en tercera o primera persona, un objecte per la escena.

7.2.3.1.11. Navigation

Serveix perquè els caràcters es puguin moure intel·ligentment per l'escena i interactuar amb els obstacles.

7.2.3.1.12. Canvas

Component que serveix per representar un espai abstracte que conté el UI.

7.2.3.2. Llibreries utilitzades

7.2.3.2.1. UnityEngine

És la llibreria principal de Unity que conté les classes específiques de Unity. Les classes més importants i utilitzades en el projecte, així com els mètodes i atributs importants, són:

7.2.3.2.1.1. GameObject

Són els objectes fonamentals a Unity que representen caràcters, objectes interactuables i escenaris. Els *GameObject* són caixes buides, que no fan res tots sols. Es fan servir com a contenidors de Components, a on s'implementen la funcionalitat real de l'objecte. Sempre tenen la component *Transform* adjuntada.

Valor de retorn	Variable	Descripció
String	tag	Conté el tag del GameObject.
Transform	transform	Retorna el component Transform.
String	name	Conté el nom del GameObject.
Animation	animation	Conté el component Animation.
AudioSource	audio	Conté el component AudioSource.
Camera	camera	Conté el component Camera.
Collider	collider	Conté el component Collider.
Bool	enabled	Booleà que diu si està actiu o no.
Light	light	Conté el component Light.
Rigidbody	rigidbody	Conté el component Rigidbody.

Valor de retorn	Mètode	Descripció
Component	AddComponent	Afegeix un component al GameObject.
Component	GetComponent	Retorna el component demanat del GameObject.
GameObject	FindGameObjectWithTag	Retorna el GameObject amb el tag entrat.
Void	Destroy	Destrueix l'objecte entrat.
Void	DestroyImmediate	Destrueix l'objecte entrat immediatament.
Object	Instantiate	Fa un clon del objecte entrat.

7.2.3.2.1.2. MonoBehaviour

Classe bàsica per crear un script que volem adjuntar en un *GameObject*. Aquesta classe conté diferents mètodes bàsics els quals sempre es van cridant, depenent de quin, cada frame del videojoc, o quan es crea el *GameObject*.

Valor de retorn	Variable	Descripció
Bool	enabled	Permet que els comportaments siguin cridats.
GameObject	gameObject	Conté el GameObject amb que està afegit.
String	tag	Conté el tag del GameObject.
Transform	transform	Conté el component Transform del GameObject.
String	name	Conté el nom del objecte.

Valor de retorn	Mètode	Descripció
Void	Invoke	Invoca un mètode entrat en un temps específic.
Coroutine	StartCoroutine	Crea una corutina.
Void	StopCoroutine	Para una corutina.
Void	Awake	Mètode cridat quant l'instancia del script és carregada.
Void	FixedUpdate	Mètode cridat cada frame fixat.
Void	LateUpdate	Mètode cridat al final d'un frame.
Void	Start	Mètode cridat al primer frame que el script està habilitat.
Void	Update	Mètode cridat cada frame.
Component	GetComponent	Retorna el component demanat del GameObject.

7.2.3.2.1.3. Transform

Classe que dona informació de la posició, rotació i escala de l'objecte.

Valor de retorn	Variable	Descripció
Vector3	localPosition	Posició del objecte amb el pare.
Quaternion	localRotation	Rotació relativa amb el pare.
Vector3	localScale	Escala relativa amb el pare.
Transform	parent	Pare del objecte.
Vector3	position	Posició relativa amb l'espai.
Quaternion	rotation	Rotació segons amb l'espai.

7.2.3.2.1.4. Collider

Classe bàsica per a tots els colliders. D'aquesta classe se'n deriven les classes *BoxCollider*, *SphereCollider*, *CapsuleCollider* i *MeshCollider*, entre d'altres, els quals tenen una estructura bàsica predefinida, com el nom de cada classe.

Valor de retorn	Variable	Descripció
Bool	enabled	Diu si està actiu o no.
Bool	isTrigger	Si és trigger no col·lisionarà amb objectes Rigidbody, només amb Collider.

Valor de retorn	Mètode	Descripció
Void	OnCollisionEnter	Cada cop que es detecti una col·lisió Collider /Rigidbody d'entrada, entrarà a aquesta funció.
Void	OnCollisionExit	Cada cop que es detecti una col·lisió Collider /Rigidbody de sortida, entrarà a aquesta funció.
Void	OnCollisionStay	Cada frame que es detecti una col·lisió Collider /Rigidbody, entrarà a aquesta funció.
Void	OnTriggerEnter	Cada cop que es detecti una col·lisió Collider d'entrada, entrarà a aquesta funció.
Void	OnTriggerExit	Cada cop que es detecti una col·lisió Collider de sortida, entrarà a aquesta funció.
Void	OnTriggerStay	Cada frame que es detecti una col·lisió Collider, entrarà a aquesta funció.

7.2.3.2.1.5. Input

És la interfície amb l'input del sistema.

Valor de retorn	Mètode	Descripció
Float	GetAxis	Retorna el valor de l'axis virtual entrat.
Bool	GetKey	Retorna cert tot el temps que es pressiona la tecla entrada per paràmetre.
Bool	GetKeyDown	Retorna cert al moment que es pressiona la tecla entrada per paràmetre.

7.2.3.2.1.6. Application

Classe que té accés a la informació en temps real de l'aplicació.

Valor de retorn	Mètode	Descripció
Void	LoadLevel	Carrega un nivell per nom o índex.
Void	Quit	Surt de l'aplicació.

7.2.3.2.1.7. NavMeshAgent

Component adjuntada a un caràcter mòbil en un joc que li permet navegar per l'escena fent servir el *NavMesh*.

Valor de retorn	Variable	Descripció
Float	acceleration	Màxima acceleració del agent.
Bool	autoRepath	Intentar aconseguir un nou camí si l'actual és invàlid.
Vector3	desiredVelocity	Velocitat desitjada per l'agent.
Vector3	destination	Lloc de l'espai on l'agent hauria d'arribar.
Bool	hasPath	Retorna un booleà si actualment l'agent té un camí.
Float	height	Alçada màxima que un agent pot passar sobre un obstacle
Float	speed	Maxima velocitat que pot tenir l'agent.

Valor de retorn	Mètode	Descripció
Void	Resume	Segueix el camí.
Bool	SetDestination	Introdueix la destinació del agent i es calcula el camí. Retorna cert si la destinació és factible.
Void	Stop	Para el moviment del agent.

7.2.3.2.2. System

Llibreria fonamental de C# que conté les classes més usuals i classes bàsiques que defineixen els valors i referencies de tipus de data més comuns així com events, interfícies, atributs i excepcions.

7.2.3.2.3. System.Collections

Llibreria que conté interfícies i classes que defineixen nombroses col·leccions d'objectes, com llistes, cues, taules i diccionaris.

7.2.3.2.4. System.Collections.Generic

Conté interfícies i classes que defineixen col·leccions genèriques, les quals permeten als usuaris crear col·leccions fortament tipades. Aquestes permeten una millor eficiència que les col·leccions no genèriques fortament tipades.

7.2.3.2.5. System.IO

Conté tipus que permeten llegir i escriure fitxers així com fluxos de dades. També conté tipus que ajuden a interactuar amb fitxers i directoris.

7.2.3.2.6. System.Math

Conté constants i mètodes estàtics per trigonometria, logarítmica i altres funcions matemàtiques comunes.

7.2.3.2.7. JSONObject

Classe creada per *Matt Schoen* que permet llegir i escriure més fàcilment fitxers JSON.

7.2.3.2.8. HLSLSupport.cginc

Fitxer inclòs automàticament a un shader, que conté macros auxiliars i definicions per compilar shaders multi plataforma.

7.2.3.2.9. UnityShaderVariables.cginc

Fitxer inclòs automàticament a un shader, que conté variables globals comunament més utilitzats.

7.2.3.2.10. UnityCG.cginc

Fitxer amb funcions d'ajuda més comunament utilitzats per shaders.

7.2.4. Microsoft Word 2013

Figura 7.3: Logo Microsoft Word 2013.

És un processador de textos comercialitzat per *Microsoft*. Va ser desenvolupat per *Richard Brodie* pel computador *d'IBM* sota un sistema operatiu *DOS* en 1983. Actualment és un paquet *Microsoft Office*.

La utilització d'aquest processador de textos i no *OpenOffice* és que actualment estic pagant un paquet de Microsoft per tenir un terabyte al núvol (*OneDrive*) amb el qual regalen *Microsoft Office*. Per tant aprofito les eines que estic pagant.

7.2.5. Visual Paradigm for UML

Figura 7.4: Logo Visual Paradigm for UML.

És una eina *UML CASE* que suporta *UML 2, SysML* i *Business Process Modeling Notation* del *Object Management Group*. Aquesta eina també està preparada per generar codi a partir dels diagrames creats, així com generar diagrames a partir de codi.

Suporta diferents tipus de diagrames, com el de classe, casos d'ús, seqüència, comunicació, diagrama d'estats, activitats, components, desplegament, de paquets, d'objectes, estructura composta, de temps i global d'interaccions.

La utilització d'aquest programa és perquè, a més de tenir una versió per estudiants, és la que ens han ensenyat a treballar a la universitat. D'aquesta manera no cal perdre temps amb aprendre una nova eina semblant.

Capítol 8. Anàlisi i disseny del sistema

En aquest capítol es descriurà com serà el videojoc i es mostraran els element més importants per l'enteniment del disseny del videojoc.

8.1. Descripció

El videojoc tractarà de ser un jugador de waterpolo que està jugant un partit amb el seu equip, contra un altre equip. Per tal de poder guanyar, el jugador principal haurà d'interactuar amb els jugadors del seu equip per poder marcar gols i acabar el partit amb més gols que l'equip rival.

El jugador veurà als jugadors de waterpolo en 3D en una vista de tercera persona.

8.2. Anàlisi

En l'anàlisi es descriurà els actors del programa, casos d'ús i les fitxes de casos d'ús.

8.2.1. Actors

Un actor és un rol fet per un usuari o un altre sistema que interactua amb el subjecte. En aquest cas només trobem un únic rol que interactuarà amb la aplicació que seria el usuari o jugador del videojoc. No trobaríem cap tipus de manteniment del sistema o un altre sistema que interactuï amb aquest.

8.2.2. Casos d'ús

Un cas d'ús és una seqüència d'interaccions que es desenvolupen entre un sistema i els actors en resposta a un esdeveniment que inicia l'actor.

En aquesta aplicació trobem casos d'ús dividits en dues parts: menú principal i partit.

8.2.2.1. Menú principal

En el menú principal trobem que el jugador del videojoc és l'actor i que només pot fer dues accions per interactuar amb el sistema: iniciar un partit i sortir de l'aplicació.

Figura 8.1: Casos d'ús del menú principal.

8.2.2.2. Partit

En el partit trobem que el jugador fa diferents accions: moure i saltar (fer un petit salt per sobre l'aigua) pel camp de joc, passar la pilota, xutar la pilota, demanar la pilota, robar pilota/pressionar al contrincant i pausar la partida.

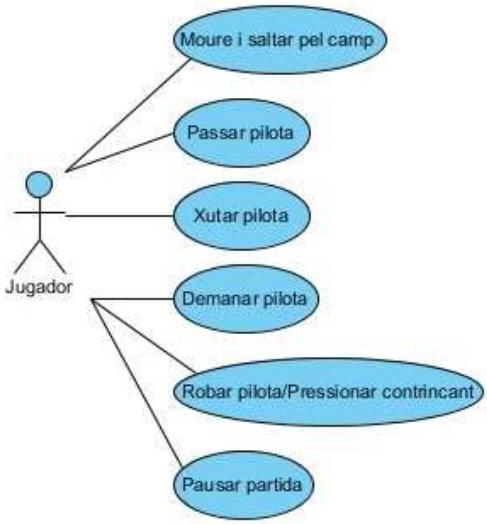


Figura 8.2: Casos d'ús del partit.

8.2.3. Fitxes de casos d'ús

Una fitxa de cas d'ús descriu el cas d'ús, actor, precondicions, fluxos principals i alternatius, postcondicions i observacions trobades. En aquesta aplicació trobem les següents fitxes de casos d'ús:

8.2.3.1. Iniciar un partit

Fitxa de cas d'ús: Iniciar un partit	
Descripció	Quan el jugador inicia el programa i pot triar entre iniciar un partit o sortir.
Actor	Jugador
Precondició	S'ha d'haver iniciat el programa
Flux principal	1. Triar iniciar un partit.
Flux alternatiu	Cap.
Postcondició	S'inicia el partit.
Observacions	Cap.

8.2.3.2. Sortir aplicació

Fitxa de cas d'ús: Sortir de l'aplicació	
Descripció	Quan el jugador inicia el programa i pot triar entre iniciar un partit o sortir.
Actor	Jugador
Precondició	S'ha d'haver iniciat el programa
Flux principal	1. Triar sortir de l'aplicació
Flux alternatiu	Cap.
Postcondició	Es tanca l'aplicació.
Observacions	Cap.

8.2.3.3. Moure i saltar pel camp

Fitxa de cas d'ús: Moure i saltar pel camp	
Descripció	El jugador s'ha de poder moure pel camp i poder saltar per agafar pilotes, xutar o passar.
Actor	Jugador
Precondició	S'ha d'haver iniciat el partit i la pilota ha d'estar en joc.
Flux principal	1. Moure el jugador de waterpolo consultant el dispositiu d'entrada.
Flux alternatiu	Cap.
Postcondició	Actualitzar la posició del jugador de waterpolo.
Observacions	Cap.

8.2.3.4. Passar pilota

Fitxa de cas d'ús: Passar pilota	
Descripció	El jugador pot passar la pilota tant sigui a un jugador com a un espai del camp.
Actor	Jugador
Precondició	Estar la pilota en joc i el jugador ha de tenir la pilota.
Flux principal	1. Capturar acció i direcció del dispositiu d'entrada. 2. Avaluar si hi ha un jugador del mateix equip al voltant del lloc triat. 3. Passar al lloc volgut si no hi ha company.
Flux alternatiu	3. Redireccionar la direcció de la pilota al company.
Postcondició	Es fa un passe al lloc triat.
Observacions	Cap.

8.2.3.5. Xutar pilota

Fitxa de cas d'ús: Xutar pilota	
Descripció	El jugador pot xutar una pilota allà on vulgui.
Actor	Jugador
Precondició	Estar la pilota en joc i el jugador ha de tenir la pilota.
Flux principal	1. Capturar acció i direcció del dispositiu d'entrada. 2. Xutar a la direcció triada.
Flux alternatiu	Cap.
Postcondició	Es fa un xut.
Observacions	Cap.

8.2.3.6. Demanar pilota

Fitxa de cas d'ús: Demanar pilota	
Descripció	Un jugador pot demanar la pilota a un company.
Actor	Jugador
Precondició	Estar la pilota en joc i la pilota la té un company.
Flux principal	1. Demanar la pilota. 2. El jugador en possessió te la passa.
Flux alternatiu	Cap.
Postcondició	Rebre la pilota.
Observacions	Cap.

8.2.3.7. Robar pilota o pressionar contrincant

Fitxa de cas d'ús: Robar pilota o pressionar contrincant	
Descripció	Quan l'equip del jugador no té la pilota, pot pressionar a un contrincant i robar la pilota, si aquest la té.
Actor	Jugador
Precondició	Pilota en possessió de l'equip contrari.
Flux principal	1. Pressionar al contrari. 2. Si aquest té pilota, pot robar-li.
Flux alternatiu	Cap.
Postcondició	Obtenció de la pilota o fer retrocedir al contrincant.
Observacions	Cap.

8.2.3.8. Pausar partida

Fitxa de cas d'ús: Pausar partida	
Descripció	Durant un partit el jugador pot parar la partida.
Actor	Jugador
Precondició	Estar en un partit.
Flux principal	1. Parar el partit 2. Mostrar el menú d'opcions per sortir o retornar al partit. 3. Triar una opció. 4. Si tria continuar el partit, el partit es reprèn.
Flux alternatiu	4. Si tria sortir, tornar al menú principal.
Postcondició	Fer la opció triada.
Observacions	Cap.

8.2.4. Diagrames d'activitat

8.2.4.1. Menú principal

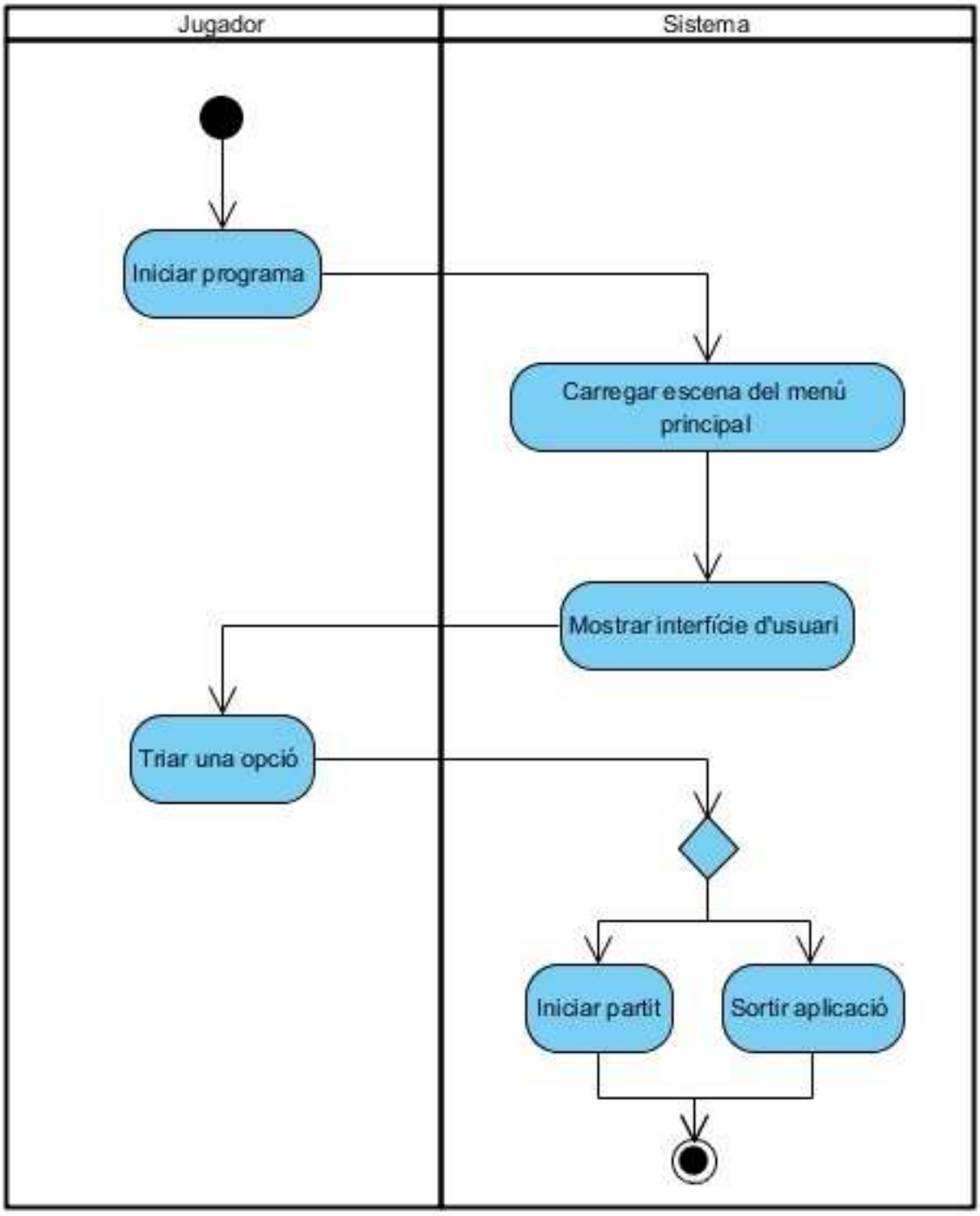


Figura 8.3: Diagrama d'activitat del menú principal.

8.2.4.2. Partit

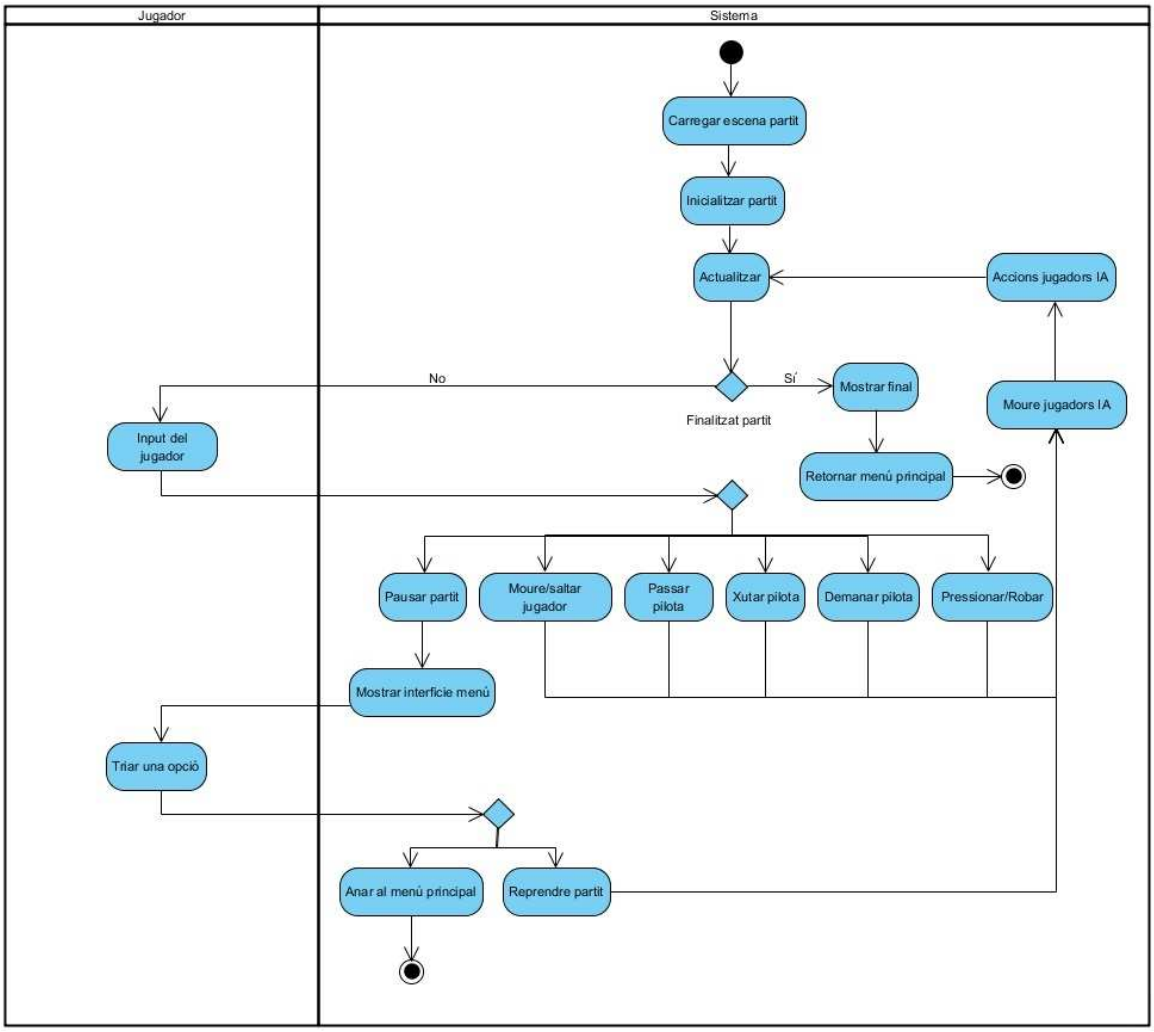


Figura 8.4: Diagrama d'activitat del partit.

8.3. Disseny

8.3.1. Interfícies d'usuaris

En aquest videojoc trobem un seguit d'interfícies d'usuari. En total en trobem 4 interfícies que es mostraran seguidament, aquests són les interfícies del menú principal, del partit, del menú de pausa i el de finalització del partit.

8.3.1.1. Interfície menú principal

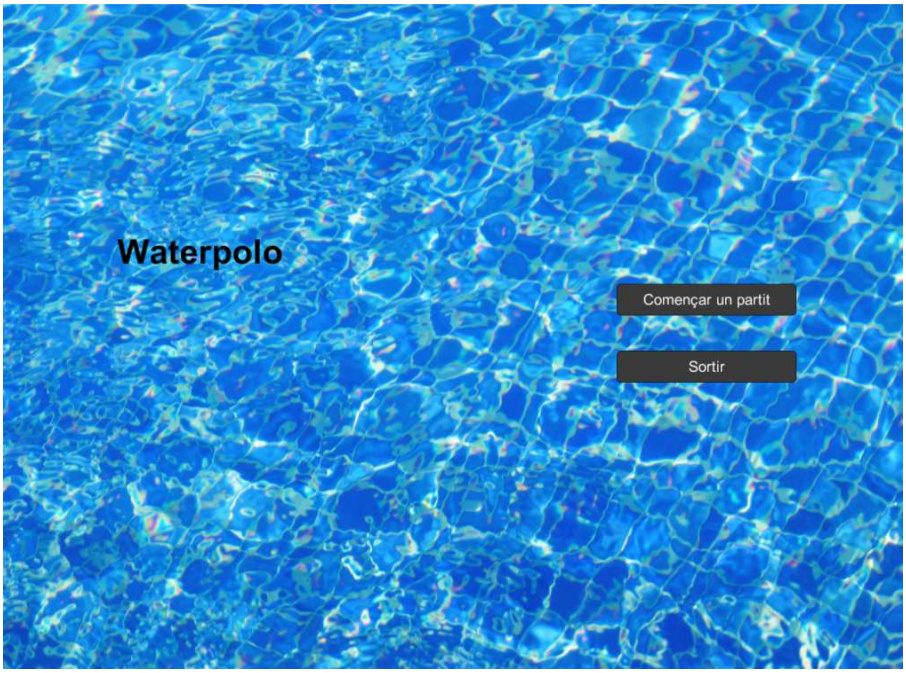


Figura 8.5: Menú principal del joc.

En la interfície del menú principal del videojoc permet començar un partit i sortir de l'aplicació.

8.3.1.2. Interfície partit

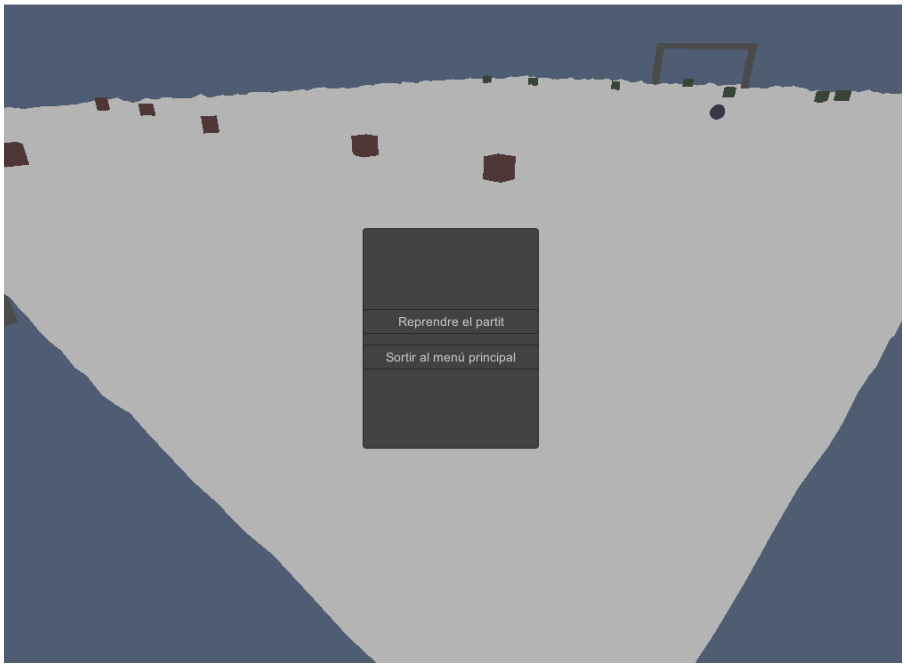


Figura 8.6: Interfície durant una pausa en un partit.

Quan premem Escape s'atura el partit i apareix un menú que permet retornar al partit o acabar el partit i anar al menú principal.

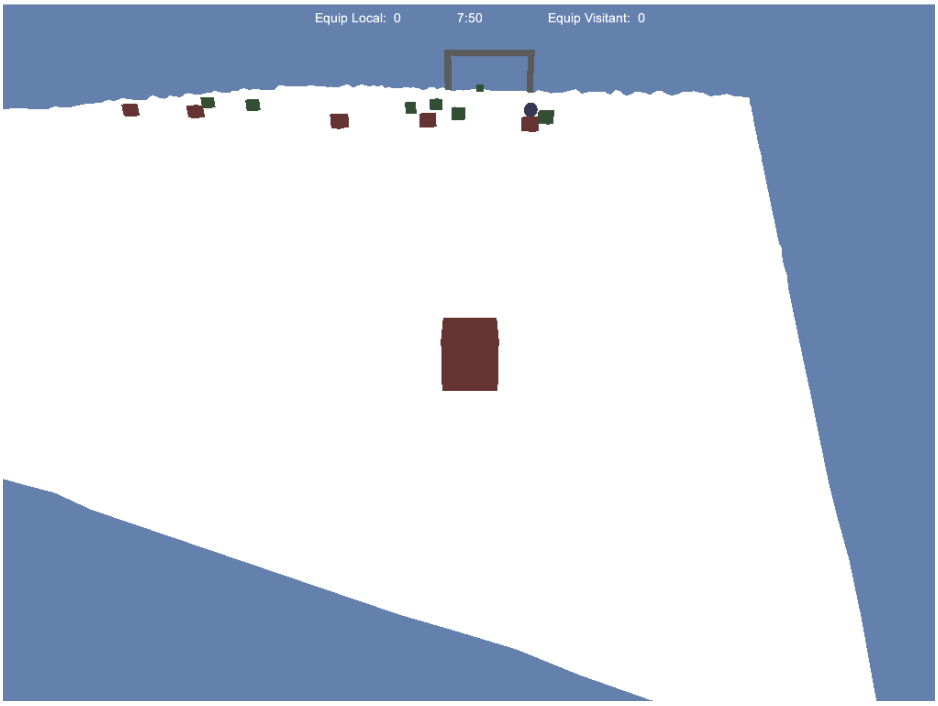


Figura 8.7: Interfície en un partit.

Durant un partit podem veure a la part superior de la pantalla el temps restant perquè acabi el joc i el marcador del partit.

Figura 8.8: Interfície un cop hem acabat el partit.

Un cop s'ha finalitzat un partit, es mostrarà aquesta interfície, on el marcador informarà si hem guanyat, perdut o empatat. També dóna la possibilitat de tornar al menú principal per poder fer un altre partit o sortir de l'aplicació.

8.3.2. Diagrama de classes

En el següent diagrama es mostra la distribució de les diferents classes de l'aplicació i les relacions entre elles. Està distribuït amb diferents colors per tal de facilitar l'enteniment. També s'hi poden veure els shaders utilitzats, que en aquest cas serien: *WaterShader, WaterShaderApplyChanges* i *WaterShaderMaterial.*

Està organitzat en tres parts de diferent color. El color verd és el mòdul que s'encarrega de moure els jugadors i la intel·ligència artificial. El color rosa són les classes inicialitzadores per les diferents escenes del videojoc. El color blau és el mòdul on estan contingudes les classes que s'encarreguen de l'aigua de la piscina.

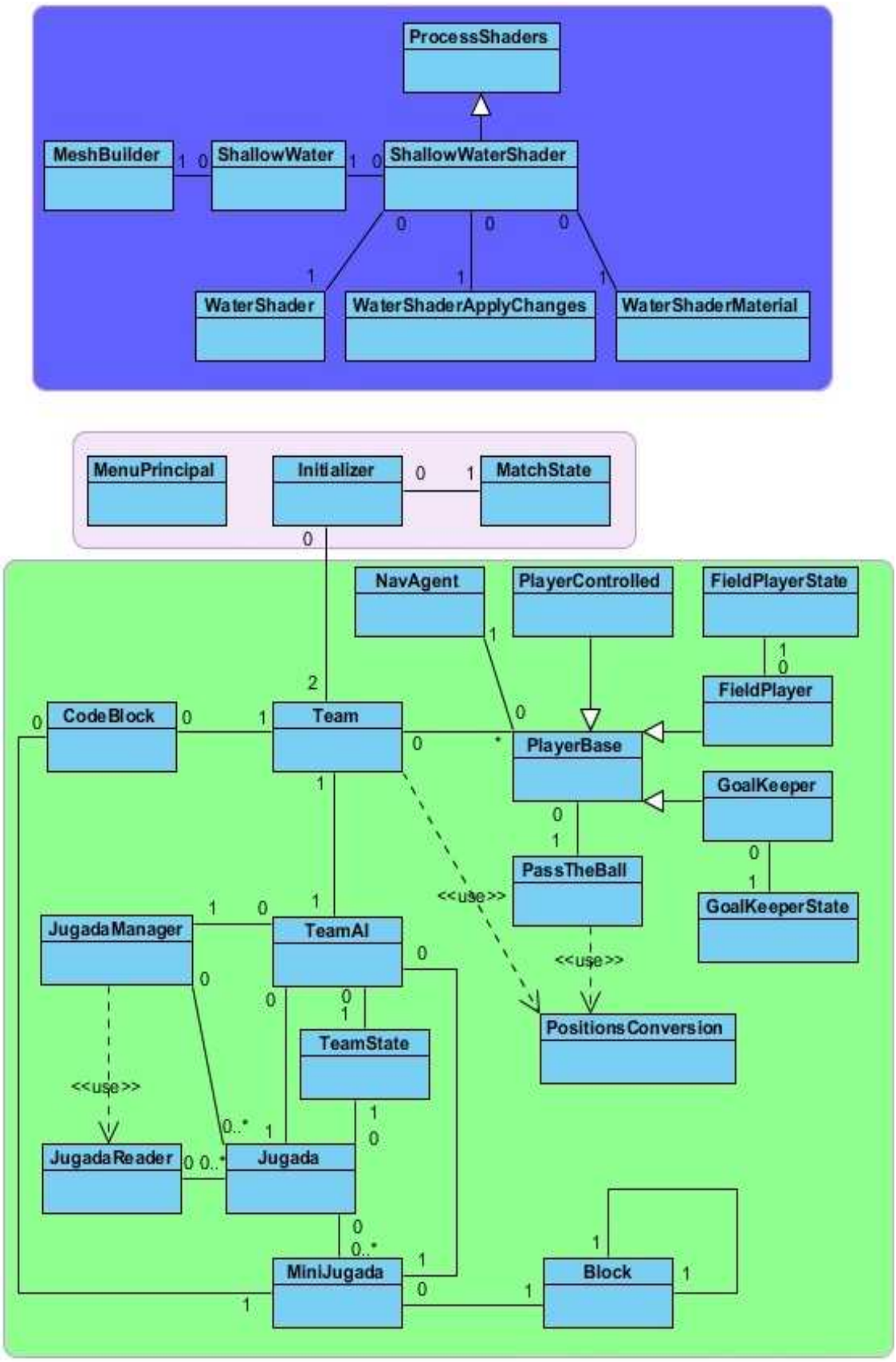


Figura 8.9: Diagrama de classes del videojoc.

8.3.2.1. Aigua

8.3.2.1.1. MeshBuilder

-List<Vector3> punts
-List<Vector3> normal
-List<Vector2> m_UVs
-List<int> index
+void AddTriangle(int index0, int index1, int index2)
+Mesh CreateMesh()

Aquesta classe permet crear malles a partir de triangles.

Atributs:

-List<Vector3> punts: Llistat que conté els diferents punts a l'espai.

-List<Vector3> normal: Llistat que conté les normals.

-List<Vector2> m_UVs: Llistat les UV.

-List<int> index: Llistat que té les posicions de cada punt del triangle.

Mètodes:

+void AddTriangle(int index0, int index1, int index2): Mètode que afegeix els índex dels punts d'un triangle.

+Mesh CreateMesh(): Mètode que crea una malla a partir del llistat de punts, normals, índex i UV.

8.3.2.1.2. ShallowWater

-int totalSegments = 30
-float lengthMesh = 10f
-float widthMesh = 10f
-float maxHeightMesh = 10f
-bool done = false
+void createPlane ()
+void BuildQuadForGrid(MeshBuilder meshBuilder, Vector3 position, Vector2 uv, bool buildTriangles, int vertsPerRow)

Classe que crea la malla de l'aigua el qual utilitza la classe MeshBuilder per crear el conjunt de triangles que configura la malla de l'aigua.

Atributs:

-int totalSegments = 30: Conté el total de segments que es farà en amplada i en llargada.

-float lengthMesh = 10f: Llargada total de la malla.

-float widthMesh = 10f: Amplada total de la malla.

-float maxHeightMesh = 10f: Alçada màxima de la malla.

-bool done = false: Quan s'ha creat la malla es posa a cert.

Mètodes:

+void createPlane (): Mètode per crear un pla amb els atributs de la classe.

+void BuildQuadForGrid(MeshBuilder meshBuilder, Vector3 position, Vector2 uv, bool buildTriangles, int vertsPerRow): Mètode per crear cada rectangle de la malla, els quals estan formats per dos triangles.

8.3.2.1.3. ProcessShaders

-Shader [] shaders
-Material [] materials
+void InitiationMaterials ()
+void InitiationMaterials (string[] shaderNames)
+void SimpleShader(RenderTexture source, RenderTexture destination, int numShader)

Classe encarregada per cridar shaders amb els corresponents materials.

Atributs:

-Shader [] shaders: Taula on s'hi desen els shaders.

-Material [] materials: Taula on s'hi desen els materials per cada shader.

Mètodes:

+void InitiationMaterials (): Inicialitza tots els materials, tants com shaders.

+void InitiationMaterials (string[] shaderNames): Inicialitza materials per als shaders entrats per paràmetre.

+void SimpleShader(RenderTexture source, RenderTexture destination, int numShader): Executa una vegada el shader.

8.3.2.1.4. ShallowWaterShader

-RenderTexture render
-Vector3[] heightMap
-Texture2D heightMapTex
-Texture2D newHeightMap
-RenderTexture newHeightMap2
-RenderTexture heightMapTex2
-float velocityPropagation = 2f
-float velocityMovement = 20f
-float time = 0.5f
-int totalSegments
-float maxHeightMesh
-float rescale = 3f
-bool doneInit = false
+void Start ()
+IEnumerator waitWaterDone()
+void Update()
+Vector3 GetVertex(int i, int j)
+void SetVertex(int i, int j, float value)

Classe encarregada de cridar els shaders que mouen l'aigua i que hereta de ProcessShader.

Atributs:

-RenderTexture render: RenderTexture temporal que es fa servir a la sortida del shader que calcula les noves altures.

-Vector3[] heightMap: Llistat de Vector3 dels vèrtex de la malla.

-Texture2D heightMapTex: Textura utilitzada per passar els valors del heightMap a una textura2D.

-Texture2D newHeightMap: Textura utilitzada per passar els valors del heightMap a una textura2D.

-RenderTexture newHeightMap2: Conté les noves altures dels vèrtex de la malla.

-RenderTexture heightMapTex2: Conté els valors actuals del vèrtex de la malla.

-float velocityPropagation = 2f: Velocitat de propagació de les altures als vèrtexs del voltant.

-float velocityMovement = 20f: Velocitat del moviment d'oscil·lació de les altures.

-float time = 0.5f: Velocitat del moviment en general.

-int totalSegments: Total de segments de cada costat de la malla.

-float maxHeightMesh: Altura total dels vèrtex de la malla.

-float rescale = 3f: Rescalació de la malla.

-bool doneInit = false: Un cop inicialitzat la malla retorna cert.

Mètodes:

+void Start (): Inicialitza els atributs de la classe i crida waitWaterDone.

+IEnumerator waitWaterDone(): Inicialitza els valors dels vèrtexs dels RenderTexture.

+void Update(): Introdueix els materials i fa les crides als shaders encarregats de moure la malla.

+Vector3 GetVertex(int i, int j): Obté el vèrtex de la posició entrada.

+void SetVertex(int i, int j, float value): Modifica l'altura de la posició entrada.

8.3.2.1.5. WaterShader

```
-float _totalSegments;
-sampler2D _MainTex;
-sampler2D _newHeightMap;
-float _velocityPropagation;
-float _maxHeightMesh;
-float _columnWidth;
-float _time;
+float4 frag (fragmentInput input)
```

No és una classe en sí, és un shader que s'encarrega de retornar les noves altures de la malla.

Atributs:

-float _totalSegments: Total de segments de cada costat de la malla.

-sampler2D _MainTex: Malla actual.

-sampler2D _newHeightMap: Textura que conté les altures de cada vèrtex.

-float _velocityPropagation: Velocitat de propagació de les altures als vèrtexs del voltant.

-float _maxHeightMesh: Altura màxima dels vèrtex de la malla.

-float _columnWidth: Velocitat del moviment d'oscil·lació de les altures.

-float _time: Velocitat del moviment en general.

Mètodes:

+float4 frag (fragmentInput input): Mètode que s'encarrega de calcular les noves altures.

8.3.2.1.6. WaterShaderApplyChanges

```
-sampler2D _MainTex
-sampler2D _newHeightMap
+float4 frag (fragmentInput input)
```

Shader que posa les noves altures a la textura principal.

Atributs:

-sampler2D _MainTex: Malla actual.

-sampler2D _newHeightMap: Textura que conté les noves altures de cada vèrtex.

Mètodes:

+float4 frag (fragmentInput input): Mètode que posa les altures a la textura principal.

8.3.2.1.7. WaterShaderMaterial

```
-sampler2D _MainTex
-sampler2D _newHeightMap
-float _maxHeightMesh
-float _rescale
+fragmentInput vert( vertexInput v )
```

Shader que canvia els vèrtexs de la geometria per les noves alçades entrades.

Atributs:

-sampler2D _MainTex: Malla actual.

-sampler2D _newHeightMap: Textura que conté les noves altures de cada vèrtex.

-float _maxHeightMesh: Altura màxima dels vèrtex de la malla.

-float _rescale: Rescalació de la malla.

Mètodes:

+fragmentInput vert(vertexInput v): Mètode que canvia les alçades de la geometria per les alçades de newHeightMap.

8.3.2.2. Inicialtizadors

8.3.2.2.1. MenuPrincipal

```
+void ExitApplication()
+void ComencarPartit()
```

Classe que s'encarrega de la interfície del menú principal.

Mètodes:

+void ExitApplication(): Mètode que sort de l'aplicació.

+void ComencarPartit(): Mètode que carrega el partit.

8.3.2.2.1. MatchState

Enumeració dels estats que pot estar un partit, com per exemple: PrimeraPart, SegonaPart, TerceraPart, QuartaPart, Descans, TempsMort.

8.3.2.2.1. Initializer

-GameObject localTeam
-GameObject visitorTeam
-GameObject player
-GameObject playerLocal
-GameObject playerVisitant
-GameObject goalKeeperLocal
-GameObject goalKeeperVisitant
-GameObject ball
-GameObject menu
-GameObject gui
-bool pause = false
-float time
-int scoreLocal = 0
-int scoreVisitant = 0
-Transform water
-MatchState matchState
+void EndedActionVisitors(GameObject player)
+void EndedActionLocals(GameObject player)
+void EndedBallPass()
+void Awake ()
+void Update ()
+void GoMainMenu()
+void ReturnPlay()
+void AddScoreLocal()
+void AddScoreVisitant()
+void RefreshTime()
+void FinalPartit()

Classe encarregada de crear els equips, l'aigua i de controlar la IA. També serveix per rebre els diferents events dels jugadors.

Atributs:

-GameObject localTeam: Contenidor de l'equip local.

-GameObject visitorTeam: Contenidor de l'equip visitant.

-GameObject Player: Contenidor del jugador que és controlat pel jugador de l'aplicació.

-GameObject playerLocal: Contenidor d'un jugador local.

-GameObject playerVisitant: Contenidor d'un jugador visitant.

-GameObject goalKeeperLocal: Contenidor d'un porter local.

-GameObject goalKeeperVisitant: Contenidor d'un porter visitant.

-GameObject ball: Contenidor de la pilota.

-GameObject menu: Contenidor de la interfície del menú de pausa.

-GameObject gui: Contenidor de la interfície del temps i marcador

-bool pause = false: Si és fals no estem al menú de pausa, al contrari sí.

-float time: Temps faltant per arribar a 0.

-int scoreLocal = 0: Marcador local.

-int scoreVisitant = 0: Marcador visitant.

-Transform water: Component transform de l'aigua per posar-la al lloc que li correspon.

-MatchState matchState: Estat actual del partit.

Mètodes:

+void EndedActionVisitors(GameObject player): event que indica que un jugador visitant ha arribat a destí.

+void EndedActionLocals(GameObject player): event que indica que un jugador local ha arribat a destí.

+void EndedBallPass(): event que ens indica que un passe s'ha finalitzat.

+void Awake (): Inicialitza els atributs entrats, com els equips.

+void Update (): Captura si s'ha premut la tecla escape per anar o no al menú de pausa i crida al mètode RefreshTime per si s'ha acabat el temps.

+void GoMainMenu(): Carrega el menú principal.

+void ReturnPlay(): Continua el joc un cop retornat del menú de pausa.

+void AddScoreLocal(): Afegeix un gol a l'equip local.

+void AddScoreVisitant(): Afegeix un gol a l'equip visitant.

+void RefreshTime(): S'encarrega d'actualitzar el temps que falta per acabar el partit.

+void FinalPartit(): Mètode encarregat de posar la interfície final que ens digui si s'ha guanyat, perdut o empatat.

8.3.2.3. Intel·ligència artificial

8.3.2.3.1. Team

-List<GameObject> listPlayers
-List<int> listPositionsToPlayers
-GameObject playerPrefabControlled
-GameObject playerPrefabNotControlled
-GameObject goalKeeper
-int numberPlayersControlledByIA = 6
-GameObject playerWithBall
-PassTheBall passTheBall
-bool ballOurTeam = false
-bool ballVisitantTeam = false
-TeamState teamState = TeamState.ComencarPart
-TeamAI teamAI
-bool matchStarted = false
-bool localTeam
-GameObject mainObject
+void Update ()
+bool TeamHasTheBall()
+bool PlayerInTeam(GameObject player)
+void MovePlayerTo(int positionPlayer, string position)
+GameObject GetCorrectPlayerFromList(int positionPlayer)
+void Setup(GameObject initializator)
+GameObject AddPlayer(int i, string preName, int zPosition, GameObject prefab)
+bool SetAlertPlayer(int i, bool alert)
+void DisableAlertAllPlayers()
+bool PassarPilota(int player, int playerToPass)
+bool XutarPilota(int player)
+int GetPositionPlayerWithBall()

Classe que conté el conjunt dels jugadors de l'equip. També serveix de connexió entre l'executor de les Jugades(CodeBlock) i els Jugadors.

Atributs:

-List<GameObject> listPlayers: Llistat dels jugadors del equip.

-List<int> listPositionsToPlayers: Llistat de les posicions dels jugadors.

-GameObject playerPrefabControlled: Contenidor dels jugadors controlats per IA.

-GameObject playerPrefabNotControlled: Contenidor del jugador no controlat per IA.

-GameObject goalKeeper: Contenidor del porter.

-int numberPlayersControlledByIA = 6: Nombre total de jugadors controlats per IA.

-GameObject playerWithBall: Jugador que té la pilota.

-PassTheBall passTheBall: Classe de passar i xutar la pilota.

-bool ballOurTeam = false: És cert si la pilota és de l'equip.

-bool ballVisitantTeam = false: És cert si la pilota és de l'equip contrari.

-TeamState teamState = TeamState.ComencarPart: Estat actual de l'equip.

-TeamAI teamAI: Intel·ligència artificial de l'equip.

-bool matchStarted = false: Cert si el partit ha començat.

-bool localTeam: Cert si l'equip és l'equip local del partit.

-GameObject mainObject: Contenidor del objecte inicialitzador.

Mètodes:

+void Update (): Comprova en tot moment de qui es la pilota i actualitza l'estat del equip.

+bool TeamHasTheBall(): Cert si el jugador que te la pilota és de l'equip.

+bool PlayerInTeam(GameObject player): Cert si el jugador forma part de l'equip.

+void MovePlayerTo(int positionPlayer, string position): Mou el jugador a la posició indicada.

+GameObject GetCorrectPlayerFromList(int positionPlayer): Obté el jugador a partir de la posició en el camp.

+void Setup(GameObject initializator): Crea tots els jugadors i els inicialitza, així com els atributs de la classe.

+GameObject AddPlayer(int i, string preName, int zPosition, GameObject prefab): Crea un jugador, l'inicialitza i l'afegeix a la llista de jugadors i posicions.

+bool SetAlertPlayer(int i, bool alert): Posa el valor d'alerta al jugador amb la posició entrada per paràmetre.

+void DisableAlertAllPlayers(): Anul·la totes les alertes dels jugadors.

+bool PassarPilota(int player, int playerToPass): Passa la pilota al jugador en la posició entrada.

+bool XutarPilota(int player): El jugador de la posició entrada xuta a porteria.

+int GetPositionPlayerWithBall(): Obté la posició del jugador amb pilota.

8.3.2.3.2. TeamAI

-Team team
-TeamState teamState = TeamState.Res
-JugadaManager jugadaManager
-Jugada actualJugada
-MiniJugada actualMiniJugada
-List<GameObject> waitingThisPlayers
-int totalPlayersWaiting = 0
-bool waitingReachPosition = false
-bool waitingBallMovement = false
+void Update ()
+void ResetListeners()
+void CaptureEvents()
+void ActionFinishedLocal(GameObject player)
+void ActionFinishedVisitant(GameObject player)
+void DisableMovementEvents()
+void CreateListeners()
+void BallReachedDestination()
+void CaptureEventsBall()
+void DisableEventsBall()

Classe que s'encarrega del control de tot el conjunt de jugadors d'un equip, i decideix quina Jugada es fa i quan es passa d'una fase de la jugada a la següent.

Atributs:

-Team team: Equip que controla.

-TeamState teamState = TeamState.Res: Estat de l'equip.

-JugadaManager jugadaManager: Classe que conté el contenidor de jugades.

-Jugada actualJugada: Jugada actual.

-MiniJugada actualMiniJugada: Fase actual de la jugada.

-List<GameObject> waitingThisPlayers: Jugadors que s'està esperant que arribin a la posició donada.

-int totalPlayersWaiting = 0: Total de jugadors que s'està esperant.

-bool waitingReachPosition = false: Cert si s'estan esperant jugadors que arribin a una posició.

-bool waitingBallMovement = false: Cert si s'està esperant que la pilota acabi el moviment.

Mètodes:

+void Update (): Depenent de l'estat es decideix fer una nova jugada o continuar la següent MiniJugada.

+void ResetListeners(): Torna a l'estat inicial els mètodes que llegeixen els events.

+void CaptureEvents(): Activa els lectors dels events.

+void ActionFinishedLocal(GameObject player): Treu de la llista de jugadors esperant en un equip local.

+void ActionFinishedVisitant(GameObject player): Treu de la llista de jugadors esperant en un equip visitant.

+void DisableMovementEvents(): Desactiva els lectors d'events de moviment.

+void CreateListeners(): Crea els lectors de moviment.

+void BallReachedDestination(): Crida a la següent minijugada.

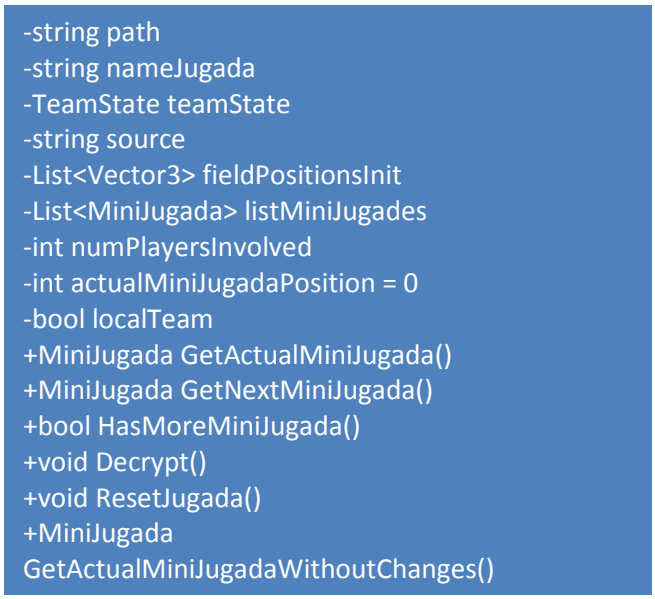
+void CaptureEventsBall(): Activa el lector d'events.

+void DisableEventsBall(): Desactiva el lector d'events.

8.3.2.3.3. TeamState

Enumeració dels estats d'un equip que poden ser: Atacant, Defensant, HomeDeMes, HomeDeMenys, Descans, ComencarPart i Res.

8.3.2.3.4. Jugada



Classe que conté una Jugada sencera que un equip pot executar. Està formada per un conjunt de fases que serien MiniJugades.

Atributs:

-string path: Camí del fitxer en el sistema de fitxers.

-string nameJugada: Nom de la Jugada.

-TeamState teamState: Estat de l'equip per fer la Jugada.

-string source: Contingut del fitxer.

-List<Vector3> fieldPositionsInit: Llistat de les posicions inicials.

-List<MiniJugada> listMiniJugades: Llistat de les MiniJugades.

-int numPlayersInvolved: Total de jugadors implicats.

-int actualMiniJugadaPosition = 0: Comptador de la fase actual de la Jugada.

-bool localTeam: Cert si l'equip és local.

Mètodes:

+MiniJugada GetActualMiniJugada(): Retorna l'actual MiniJugada i posa el comptador a la posició següent.

+MiniJugada GetNextMiniJugada(): Retorna la següent MiniJugada.

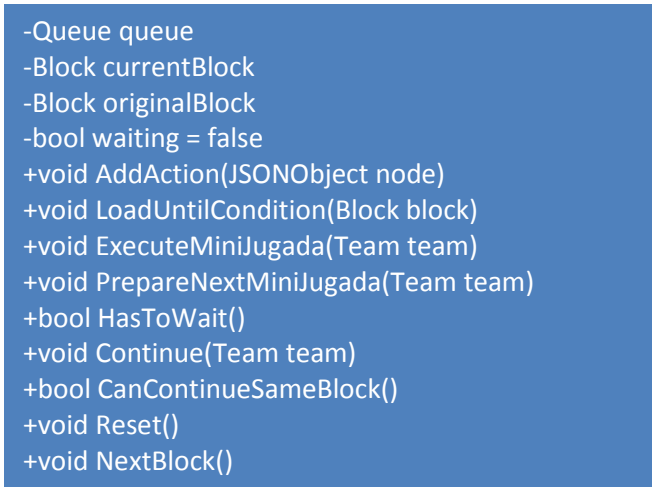
+bool HasMoreMiniJugada(): Cert si hi ha més MiniJugades per fer.

+void Decrypt(): Inicialitza els atributs amb la informació del fitxer JSON.

+void ResetJugada(): Reinicia la jugada a la MiniJugada inicial.

+MiniJugada GetActualMiniJugadaWithoutChanges(): Retorna la següent MiniJugada sense posar el comptador a la següent posició.

8.3.2.3.5. MiniJugada



Classe encarregada de guardar la fase actual d'una Jugada i d'ordenar al CodeBlock que executi la MiniJugada.

Atributs:

-Queue queue: Cua dels Blocks per executar.

-Block currentBlock: Block actual.

-Block originalBlock: Block inicial de la MiniJugada.

-bool waiting = false: Cert si s'està esperant un event de fora.

Mètodes:

+void AddAction(JSONObject node): Inicialitza la MiniJugada amb el codi JSON.

+void LoadUntilCondition(Block block): Carrega a la cua tots els blocs fins trobar una condició.

+void ExecuteMiniJugada(Team team): Executa el bloc actual fins acabar o que s'hagi d'esperar.

+void PrepareNextMiniJugada(Team team): Executa el bloc actual.

+bool HasToWait(): Cert si el block actual diu que s'ha d'esperar.

+void Continue(Team team): Continuar l'execució dels blocs.

+bool CanContinueSameBlock(): Cert si hi ha més Blocks a executar.

+void Reset(): Reinicia al block inicial la MiniJugada.

+void NextBlock(): Passa al següent Block.

8.3.2.3.6. JugadaReader

```
-string pathToPositionsReaderFolder = "Jugades"
-List<Jugada> listJugades
+JugadaReader(bool isLocalTeam)
```

Classe encarregada de llegir totes les Jugades entrades en format JSON, el qual, agafa tota la informació dels fitxers i crea un llistat de Jugades.

Atributs:

-string pathToPositionsReaderFolder = "Jugades": Carpeta on estan les Jugades a llegir.

-List<Jugada> listJugades: Conjunt de Jugades llegides.

Mètodes:

+JugadaReader(bool isLocalTeam): Llegeix tots els fitxers de la carpeta i els afegeix al llistat de Jugades.

8.3.2.3.7. JugadaManager

```
-Dictionary<TeamState, List<Jugada>> mapJugades
+public JugadaManager(bool isLocalTeam)
+public Jugada
GetRandomJugadaByTeamState(TeamState state)
```

Classe que emmagatzema el conjunt de Jugades disponibles ordenada per TeamState.

Atributs:

-Dictionary<TeamState, List<Jugada>> mapJugades: Diccionari on per cada estat de l'equip trobem un llistat de Jugades disponibles per fer servir.

Mètodes:

+public JugadaManager(bool isLocalTeam): Inicialitza la classe i emplena el diccionari de Jugades.

+public Jugada GetRandomJugadaByTeamState(TeamState state): Obté una Jugada aleatòria a partir d'un estat d'equip.

8.3.2.3.8. Block

-bool isLoop = false
-bool hasCondition = false
-bool isWait = false
-JSONObject loopCode
-JSONObject loopCondition
-JSONObject sourceCode
-Block nextChildTrueBlock
-Block nextChildFalseBlock
-Block next
-Block parent
-string action
+Block(JSONObject source, Block parent, Block next)
+void ExtractInfoBlock(JSONObject source)
+void ExtractBrothers(JSONObject source)
+Block ExtractChildren(JSONObject source)
+JSONObject CreateJSONObject(string key, JSONObject itemList)
+JSONObject TransformToList(JSONObject source)

Classe que conté una línia de codi d'una MiniJugada. Cada block està relacionat amb el següent block, que seria la següent línia de codi.

Atributs:

-bool isLoop = false: Cert si el block és un for o un while.

-bool hasCondition = false: Cert si el block té una condició.

-bool isWait = false: Cert si el block fa esperar.

-JSONObject loopCode: Conté el codi per fer cicles.

-JSONObject loopCondition: Conté el codi de la condició.

-JSONObject sourceCode: Conté el codi original.

-Block nextChildTrueBlock: El següent Block si la condició és certa o no té cap condició.

-Block nextChildFalseBlock: El següent Block si la condició és falsa.

-Block next: El següent Block.

-Block parent: El Block del pare.

-string action: Acció a fer pel jugador.

Mètodes:

+Block(JSONObject source, Block parent, Block next): Inicialitza el Block recursivament.

+void ExtractInfoBlock(JSONObject source): Extreu la informació del Block i la posa als atributs.

+void ExtractBrothers(JSONObject source): Extreu la informació per als Blocks contigus a aquest, els quals no es troben dintre del Block que el crida.

+Block ExtractChildren(JSONObject source): Extreu la informació per als Blocks fills a aquest.

+JSONObject CreateJSONObject(string key, JSONObject itemList): Retorna un JSONObject amb la informació entrada.

+JSONObject TransformToList(JSONObject source): Canvia el JSONObject a tipus array i el retorna.

8.3.2.3.9. CodeBlock

-Team team
-int player
-MiniJugada miniJugada
+bool HasSomething(Block block, string keySearched)
+void Execute(Block block, Team team1, MiniJugada minijugada)
+void ChooseKey(Block block)
+void Passar(Block block)
+void If(Block block)
+void While(Block block)
+bool LookCondition(JSONObject jSONObject)
+void Wait(Block block)
+void Swim(Block block)
+void Xutar(Block block)
+void Sprint(Block block)

Classe que s'encarrega de desxifrar el contingut de cada Block i que el converteix en accions per als diferents Jugadors.

Atributs:

-Team team: Equip de la classe.

-int Player: Posició del jugador.

-MiniJugada miniJugada: Minijugada que s'està referint.

Mètodes:

+bool HasSomething(Block block, string keySearched): Retorna cert si el block té el valor entrat.

+void Execute(Block block, Team team1, MiniJugada minijugada): Executa el Block entrat.

+void ChooseKey(Block block): A partir de l'acció del Block, l'executa.

+void Passar(Block block): Executa l'acció de passar una pilota al jugador que trobem al Block

+void If(Block block): Executa una condició if i segons el resultat tria el Block per condició certa o falsa.

+void While(Block block): Executa una condició while que segons el resultat tria el Block de dins el while o el següent.

+bool LookCondition(JSONObject jSONObject): Retorna cert si la condició del Block és certa, altrament fals.

+void Wait(Block block): Executa l'acció d'esperar per el jugador de la posició en el Block entrat.

+void Swim(Block block): Executa l'acció de nedar per el jugador de la posició en el Block entrat.

+void Xutar(Block block): Executa l'acció de xutar per el jugador de la posició en el Block entrat.

+void Sprint(Block block): Executa l'acció de sprintar per el jugador de la posició en el Block entrat.

8.3.2.3.10. PlayerBase

-PassTheBall passTheBall
-bool isLocal
-NavAgent agent
+void IsLocal(bool value)
+bool PassBall(GameObject player)
+void LookAt(Vector3 target)
+void MoveTo(Vector3 position)
+void MoveToPlayer(GameObject player)
+void SpringTo(Vector3 position)
+void SpringToPlayer(GameObject player)
+bool Xutar(Vector3 position)
+void Press(GameObject Player)
+void Jump()

Classe abstracta bàsica que utilitzen tots els jugadors. Aquesta classe permet passar la pilota, xutar, saltar, moure's i sprintar.

Atributs:

-PassTheBall passTheBall: Classe per poder fer passes i xuts.

-bool isLocal: Cert si forma part de l'equip local.

-NavAgent agent: Classe per moure's per la piscina.

Mètodes:

+void IsLocal(bool value): Modifica l'atribut isLocal.

+bool PassBall(GameObject player): Passa la pilota al jugador entrat.

+void LookAt(Vector3 target): Mira a la posició entrada.

+void MoveTo(Vector3 position): Es mou a la posició entrada.

+void MoveToPlayer(GameObject player): Es mou a la posició del jugador entrat.

+void SpringTo(Vector3 position): Sprinta a la posició entrada.

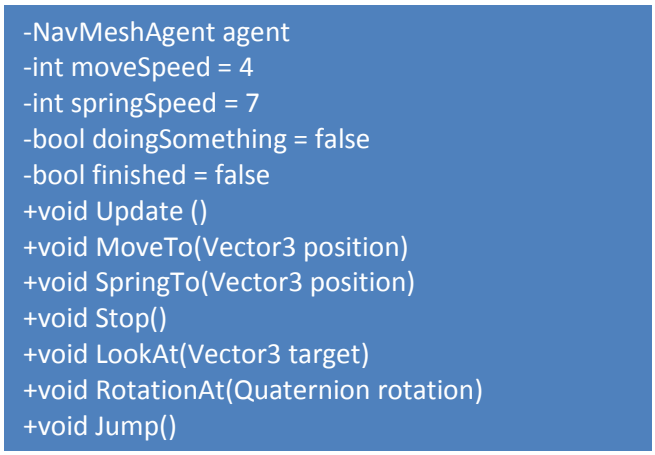
+void SpringToPlayer(GameObject player): Sprinta a la posició del jugador entrat.

+bool Xutar(Vector3 position): Xuta a la posició entrada.

+void Press(GameObject Player): Pressiona al jugador entrat.

+void Jump(): Fa un salt al camp.

8.3.2.3.11. NavAgent



Classe que permet moure un objecte per la part de l'escenari que es permet ser recorregut per un NavMeshAgent. Aquesta classe permet moure, parar, rotar/mirar i saltar.

Atributs:

-NavMeshAgent agent: Classe de l'API que controla el moviment d'un objecte.

-int moveSpeed = 4: Velocitat de nedar.

-int springSpeed = 7: Velocitat de sprint.

-bool doingSomething = false: Cert si està fent alguna cosa

-bool finished = false: Cert si ha arribat a la destinació.

Mètodes:

+void Update (): Controla si ha arribat d'actualitzar els atributs.

+void MoveTo(Vector3 position): Mou l'objecte a la posició entrada.

+void SpringTo(Vector3 position): Mou l'objecte amb la velocitat springSpeed a la posició entrada.

+void Stop(): Para el moviment de l'objecte.

+void LookAt(Vector3 target): Rota l'objecte a la posició entrada.

+void RotationAt(Quaternion rotation): Rota l'objecte amb la rotació entrada.

+void Jump(): Salta al camp.

8.3.2.3.12. PlayerControlled

+void Update()
+void NotMoving()
+void LookAtMouse()
+void ChangeMoving()
+void RequestBall()
+void StealBall(GameObject player)

Classe que llegeix els inputs del jugador i hereta de la classe PlayerBase. Aquesta classe permet moure el jugador, robar una pilota i demanar la pilota.

Mètodes:

+void Update(): Captura si s'ha premut alguna tecla que faci interaccionar el jugador.

+void NotMoving(): Para el jugador de moure's.

+void LookAtMouse(): Rota el jugador a la posició del ratolí.

+void ChangeMoving(): Mou el jugador a la posició entrada per teclat.

+void RequestBall(): Demana la pilota.

+void StealBall(GameObject player): Roba la pilota

8.3.2.3.13. FieldPlayer

-bool launchAlert = false
+void Update()
+event Initializer.EndedActionVisitors
OnEndedMovementActionVisitors
+event Initializer.EndedActionLocals
OnEndedMovementActionLocals
+void MoveTo(Vector3 position)
+void MoveToPlayer(GameObject player)
+void SpringTo(Vector3 position)
+void SpringToPlayer(GameObject player)

Classe que controla els jugadors de camp i que hereta de la classe PlayerBase. Aquesta classe permet moure i sprintar. També llança events quan arriba a la seva destinació segons la Jugada.

Atributs:

-bool launchAlert = false: Cert si ha de comprovar que ha arribat a la destinació del moviment.

Mètodes:

+void Update(): Controla si ha arribat a destinació d'aixecar la bandera dels events.

+event Initializer.EndedActionVisitors OnEndedMovementActionVisitors: Event per quan un jugador visitant arriba a destinació.

+event Initializer.EndedActionLocals OnEndedMovementActionLocals: Event per quan un jugador local arriba a destinació.

+void MoveTo(Vector3 position): Mou el jugador a la posició entrada.

+void MoveToPlayer(GameObject player): Mou el jugador a la posició del jugador entrat.

+void SpringTo(Vector3 position): Sprinta el jugador a la posició entrada.

+void SpringToPlayer(GameObject player): Sprinta el jugador a la posició del jugador entrat.

8.3.2.3.14. FieldPlayerState

Enumeració dels estats d'un jugador de camp. Aquests poden ser els següents: *Descansant, Saltant, Nedant, Sprintant, Presionant, FentFalta, FentExpulsio, Passant, Xutant, XutarPenal, EsperarXutPenal, HomeDeMes* i *HomeDeMenys.*

8.3.2.3.15. GoalKeeper

-Vector3 initPosition
-float metersAdvanced = 5
+void DefendGoal(Vector3 centerGoal)
+void ResetPosition()
+void MoveTo(Vector3 position)
+void MoveToPlayer(GameObject player)
+void SpringTo(Vector3 position)
+void SpringToPlayer(GameObject player)

Classe que controla els porters i que hereta de la classe PlayerBase. Permet defensar la porteria, moure i sprintar.

Atributs:

-Vector3 initPosition: Posició inicial del porter.

-float metersAdvanced = 5: Quantitat d'unitats que s'avança.

Mètodes:

+void DefendGoal(Vector3 centerGoal): Fa que sempre el porter estigui entre el centre de la porteria i el lloc on estigui la pilota.

+void ResetPosition(): Retorna el porter a la posició inicial.

+void MoveTo(Vector3 position): Mou el porter a la posició entrada.

+void MoveToPlayer(GameObject player): Mou el porter a la posició del jugador entrat.

+void SpringTo(Vector3 position): Sprinta el porter a la posició entrada.

+void SpringToPlayer(GameObject player): Sprinta el porter a la posició del jugador entrat.

8.3.2.3.16. GoalKeeperState

Enumeració dels estats d'un porter. Aquests poden ser els següents: *Descansant, Saltant, Nedant, Sprintant, Presionant, FentFalta, FentExpulsio, Passant, Xutant* i *Parant.*

8.3.2.3.17. PassTheBall

-bool launchAlert = false
-Vector3 currentBallPosition
-Vector3 startBallPosition
-Vector3 destination
-Vector3 gravity
-float currentTime
-float startTime
-bool ballMoving = false
-Vector3 displacement
-double timeToReach
-GameObject player
-GameObject oldPlayer
+event Initializer.EndedBallPass OnEndedBallPass
+void Update ()
+bool Passar(GameObject playerToPass)
+bool Xutar(Vector3 target)
+void UpdatePosition()
+void OnCollisionEnter(Collision other)
+bool CanGiveAPass()
+GameObject WhoHasTheBall()
+Vector3 CurrentBallPosition()

Classe que controla la pilota. Permet fer passes entre jugadors, a l'espai i també poder fer xuts.

Atributs:

-bool launchAlert = false: Cert si s'ha d'enviar un event per quan hagi arribat a destinació.

-Vector3 currentBallPosition: Posició actual de la pilota.

-Vector3 startBallPosition: Posició inicial de la pilota.

-Vector3 destination: Posició final de la pilota.

-Vector3 gravity: Gravetat que té la pilota.

-float currentTime: Temps actual.

-float startTime: Temps inicial que la pilota ha començat el moviment.

-bool ballMoving = false: Cert si la pilota està en moviment.

-Vector3 displacement: Direcció de desplaçament de la pilota.

-double timeToReach: Temps que tardarà a arribar a destí.

-GameObject Player: Jugador que té la pilota.

-GameObject oldPlayer: Antic Jugador que tenia la pilota.

Mètodes:

+event Initializer.EndedBallPass OnEndedBallPass: Event que es crida quan la pilota ha arribat a destí.

+void Update (): Actualitza la posició de la pilota i del temps.

+bool Passar(GameObject playerToPass): Fa un passe al objecte entrat per parametre.

+bool Xutar(Vector3 target): Fa un xut a la posició entrada.

+void UpdatePosition(): Actualitza la posició de la pilota.

+void OnCollisionEnter(Collision other): Depenent de la col·lisió, la pilota es deté.

+bool CanGiveAPass(): Cert si es pot fer un passe.

+GameObject WhoHasTheBall(): Retorna qui té la pilota.

+Vector3 CurrentBallPosition(): Retorna la posició actual de la pilota.

8.3.2.3.18. PositionsConversion

```
-float distanceBetweenLetters = 2
-float distanceBetweenNumbers = (float)3.5
-int letterStart = -20
-int numberStart = -40
+Vector3 Convert(string position, bool localTeam)
```

Classe que divideix el camp de joc en una malla format per una lletra i un número. A partir d'aquesta codificació ens dóna la posició en el món real del videojoc.

Atributs:

-float distanceBetweenLetters = 2: Distància entre les diferents lletres de la malla.

-float distanceBetweenNumbers = (float)3.5: Distància entre els diferents números de la malla.

-int letterStart = -20: Començament en el món de la part de les lletres.

-int numberStart = -40: Començament en el món de la part dels números.

Mètodes:

+Vector3 Convert(string position, bool localTeam): Converteix la posició codificada i si l'equip és local o no a una posició en el món.

Capítol 9. Implementació i proves

En aquest capítol es detallaran les parts més complexes del joc, explicant els diferents problemes trobats i les seves solucions.

9.1. Aigua

En aquest apartat s'explica com s'ha fet per crear l'aigua de la piscina.

Primerament es va fer un codi per tal de fer una malla per aplicar-hi les modificacions necessàries que simulessin l'aigua. Per tal de poder fer aquesta malla es va crear dos classes, la classe MeshBuilder i ShallowWater.

La classe MeshBuilder és l'encarregada de guardar els diferents punts a l'espai, poder fer les unions entre els punts formant triangles i finalment construir la malla.

En el següent codi es pot veure com es crea una malla a partir dels punts, normals i UV que tingui com a atributs de la classe. Es fa servir l'API d'unity per simplificar el codi.

```
public Mesh CreateMesh()
{
    Mesh mesh = new Mesh();

    mesh.vertices = punts.ToArray();
    mesh.triangles = indexs.ToArray();

    if (normals.Count == punts.Count)
        mesh.normals = normals.ToArray();

    if (m_UVs.Count == punts.Count)
        mesh.uv = m_UVs.ToArray();

    mesh.RecalculateBounds();

    return mesh;
}
```

La classe ShallowWater és l'encarregada d'introduir tots els vèrtexs, normals i UV a la classe MeshBuilder per tal de crear la malla.

En el següent codi podem veure com al mètode createPlane crea un MeshBuilder, per cada segment a tenir, afegeix els vèrtex, normal i UV mitjançant el mètode BuildQuadForGrid i llavors crea la malla.

```
void createPlane (){
    MeshBuilder meshBuilder = new MeshBuilder();

    for (int i = 0; i <= totalSegments; i++){
        float z = lengthMesh * i;
        float v = (1.0f / totalSegments) * i;
        for (int j = 0; j <= totalSegments; j++){
            float x = widthMesh * j;
            float u = (1.0f / totalSegments) * j;
            Vector3 offset = new Vector3(x, Random.Range(-
maxHeightMesh, maxHeightMesh), z);
            Vector2 uv = new Vector2(u, v);
            bool buildTriangles = i > 0 && j > 0;
            BuildQuadForGrid(meshBuilder, offset, uv, buildTriangles,
totalSegments+1);
        }
    }
    MeshFilter filter = GetComponent<MeshFilter>();

    if (filter != null){
        Mesh mesh = meshBuilder.CreateMesh();
        mesh.RecalculateNormals();
        filter.sharedMesh = mesh;
    }
    done = true;
}

void BuildQuadForGrid(MeshBuilder meshBuilder, Vector3 position,
Vector2 uv, bool buildTriangles, int vertsPerRow) {
    meshBuilder.Vertexs.Add(position);
    meshBuilder.UVs.Add(uv);

    if (buildTriangles) {
        int baseIndex = meshBuilder.Vertexs.Count - 1;

        int index0 = baseIndex;
        int index1 = baseIndex - 1;
        int index2 = baseIndex - vertsPerRow;
        int index3 = baseIndex - vertsPerRow - 1;

        meshBuilder.AddTriangle(index0, index2, index1);
        meshBuilder.AddTriangle(index2, index3, index1);
    }
}
```

Ara, un cop obtinguda una malla per treballar, s'ha de fer moure les alçades dels vèrtex de la malla. Per tal de fer moure les alçades d'una manera realista, el que es fa és primerament obtenir les alçades dels vèrtex veïns. Llavors calcular la velocitat que augmentarà o disminuirà l'alçada i calcular la nova alçada. Un cop obtinguda la nova alçada s'aplicaria a la malla i es guardaria la velocitat de cada vèrtex per a la següent passada. El pseudocodi seria el següent, on *f* seria l'acceleració, *c* seria la velocitat de la onada, *u* seria la malla, *h* seria l'alçada i u_{new} la malla amb la nova alçada:

```
forall i,j
   f = c^2*(u[i+1,j]+u[i-1,j]+u[i,j+1]+u[i,j-1] – 4u[i,j])/h^2
   v[i,j] = v[i,j] + f*Δt
   u_new[i,j] = u[i,j] + v[i]*Δt
endfor
forall i,j: u[i,j] = u_new[i,j]
```

Un cop passat de pseudocodi a C# obtenim el següent codi:

```
heightMap =
GameObject.FindGameObjectWithTag("Water").GetComponent<MeshFilter>().m
esh.vertices;
for (int i = 0; i < totalSegments+2; i++) {
    for (int j = 0; j < totalSegments+2; j++){
        bool wall = false;
        float sumHeight = 0f;
        int totalNoWalls = 0;
        if(i>0){
            sumHeight+=getVertex(i-1, j).y;
            totalNoWalls++;
        }
        else{
            wall = true;
        }
        if(i<totalSegments+1){
            sumHeight+=getVertex(i+1, j).y;
            totalNoWalls++;
        }
        else{
            wall = true;
        }
        if(j>0){
            sumHeight+=getVertex(i, j-1).y;
            totalNoWalls++;
        }
        else{
            wall = true;
        }
        if(j<totalSegments+1){
            sumHeight+=getVertex(i, j+1).y;
            totalNoWalls++;
        }
        else{
            wall = true;
        }
        float f = velocityPropagation*(sumHeight-
totalNoWalls*getVertex(i, j).y)/columnWidth;
        float previousVelocityMap =
newHeightMap.GetPixel(i,j).g*maxHeightMesh;
        if(newHeightMap.GetPixel(i,j).a != 1f){
            previousVelocityMap = -previousVelocityMap;
        }
        float velocityMap = previousVelocityMap + f*time;
        Color color = new Color(0f,0f,1f,1f);
        color.g = Mathf.Abs(velocityMap/maxHeightMesh);
        float vertexNextPosition;
        if(velocityMap<0){
            color.a = 0f;
        }
        if(!wall){
```

```
            vertexNextPosition = (getVertex(i, j).y +
velocityMap)*0.95f;
        }
        else{
            vertexNextPosition = (getVertex(i, j).y +
velocityMap)*0.9f;
        }
        color.r = Mathf.Abs(vertexNextPosition/maxHeightMesh);
        if(vertexNextPosition<0){
            color.b = 0f;
        }
        newHeightMap.SetPixel(i,j,color);
    }
}
for (int i = 0; i < totalSegments+2; i++) {
    for (int j = 0; j < totalSegments+2; j++){
        float vertexNextPosition =
newHeightMap.GetPixel(i,j).r*maxHeightMesh;
        if(newHeightMap.GetPixel(i,j).b != 1f){
            vertexNextPosition = -vertexNextPosition;
        }
        setVertex(i, j, vertexNextPosition);
    }
}
GameObject.FindGameObjectWithTag("Water").GetComponent<MeshFilter>().m
esh.vertices = heightMap;
```

Amb aquest codi ens trobem un problema, el qual és que el codi s'executa en la CPU. Amb això trobem que per moure una malla molt grossa és molt costós en recusos per la CPU. Amb les proves que he fet al meu ordinador, havia trobat que amb una malla de 30 per 30 ja hi havia una baixada important per segon.

Per tal de millorar l'eficiència del codi, la solució va ser que tota aquesta feina la fes la GPU i així alliberar recursos per la CPU per a la intel·ligència artificial del joc. El codi final ha estat el següent, on s'ha dividit el codi anterior en diferents shaders. Els dos més importants són el WaterShader i el WaterShaderMaterial.

El shader WaterShader s'encarrega de fer els càlculs de les noves altures de la malla i els retorna com a una textura. Com a dificultat afegeida, les textures només permeten com a valors dels colors entre 0 i 1. Per tant s'ha hagut de normalitzar els valors de les altures i de la velocitat.

```
float4 frag (fragmentInput input):COLOR{

        bool wall = false;
        float sumHeight = 0;
        int totalNoWalls = 0;
        int i = input.pos.x;
        int j = input.pos.y;
        if(i>0){
            sumHeight += tex2D(_MainTex, float2(i-1,j)).x * 2 * _maxHeightMesh -
_maxHeightMesh;
            totalNoWalls++;
        }
        else{
```

```
                wall = true;
        }
        if(i<_totalSegments){
                sumHeight += tex2D(_MainTex, float2(i+1,j)).x * 2 * _maxHeightMesh -
_maxHeightMesh;
                totalNoWalls++;
        }
        else{
                wall = true;
        }
        if(j>0){
                sumHeight += tex2D(_MainTex, float2(i,j-1)).x * 2 * _maxHeightMesh -
_maxHeightMesh;
                totalNoWalls++;
        }
        else{
                wall = true;
        }
        if(j<_totalSegments){
                sumHeight += tex2D(_MainTex, float2(i,j+1)).x * 2 * _maxHeightMesh -
_maxHeightMesh;
                totalNoWalls++;
        }
        else{
                wall = true;
        }
        float mainPixel = tex2D(_newHeightMap, input.tex).x * 2 * _maxHeightMesh -
_maxHeightMesh;
        float f = _velocityPropagation*(sumHeight-totalNoWalls*mainPixel)/_columnWidth;
        float previousVMap = tex2D(_newHeightMap, input.tex).y * 2 * _maxHeightMesh -
_maxHeightMesh;
        float velocityMap = previousVMap + f*_time;
        float vertexNextPosition = mainPixel + velocityMap;

        if(vertexNextPosition < -_maxHeightMesh){
                vertexNextPosition = - _maxHeightMesh;
        }
        if(vertexNextPosition > _maxHeightMesh){
                vertexNextPosition = _maxHeightMesh;
        }
        if(wall){
                vertexNextPosition = vertexNextPosition*0.97;
        }
        else{
                vertexNextPosition = vertexNextPosition*0.97;
        }
        float4 next;
        next.r = (vertexNextPosition +_maxHeightMesh)/(2*_maxHeightMesh);
        next.g = (velocityMap+_maxHeightMesh)/(2*_maxHeightMesh);
        next.b = 0;
        next.a = 1;
        return next;
}
```

En el shader WaterShaderMaterial el que es fa és modificar les altures dels vèrtexs de la malla a partir d'una textura entrada:

```
fragmentInput vert( vertexInput v )
{
	float3 v0 = mul( _Object2World, v.pos ).xyz;
	float4 tex = tex2Dlod(_newHeightMap, float4(v.texc.xy,0,0));
	v0.y = (tex.r*2*_maxHeightMesh-_maxHeightMesh)*_rescale;
	v.pos.xyz = mul( _World2Object, v0 );
	fragmentInput o;
	o.pos = mul(UNITY_MATRIX_MVP, v.pos);
	o.tex = v.texc;
	return o;
}
```

En les següents figures es pot veure com, a partir de donar l'aigua valors aleatoris d'altura, queda una aigua molt escarpada i al cap d'una estona es calma.

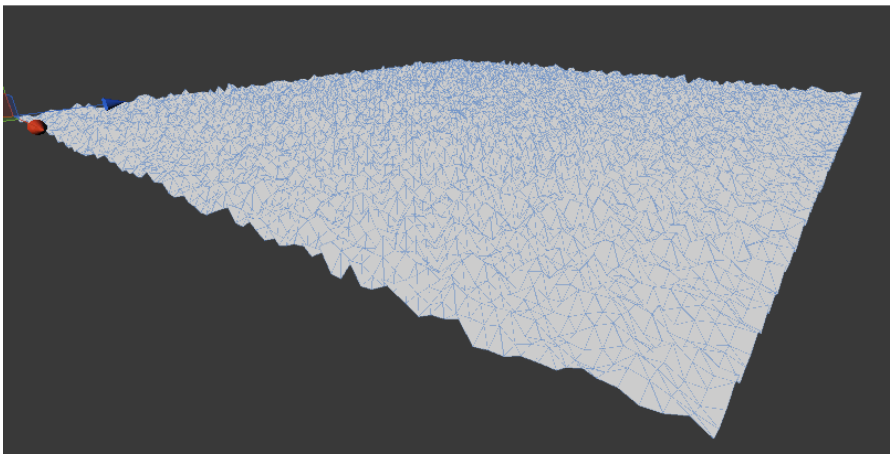


Figura 9.1: Aigua al principi de tenir valors d'altura aleatories.

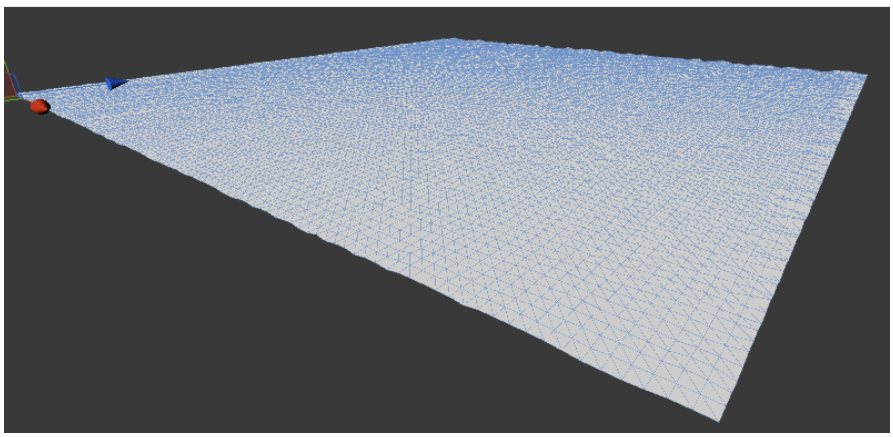


Figura 9.2: Aigua al cap d'una estona.

9.2. Passar la pilota

Per tal de passar la pilota d'un jugador a un altre es crea un codi que a partir de l'acceleració, velocitat i la gravetat fa que la pilota segueixi un moviment parabòlic. Aquest moviment seria el següent en pseudocodi:

```
currentBallPosition = startBallPosition + displacement*time + gravity*time²
```

Aquesta formula calcula la posició actual de la pilota com la suma de: posició de la pilota al principi del moviment, vector de desplaçament multiplicat pel temps i la gravetat multiplicat pel temps al quadrat.

Per tant el codi final que tenim és el següent, on està dividit en dos mètodes. El mètode passar inicialitza les variables necessàries, com per exemple el temps inicial, la posició inicial, la destinació. També es calcula el temps necessari per arribar a cobrir la distancia a un temps real (uns 5 metres s'haurien de fer en 2 segons). I finalment es calcula el vector de desplaçament que està format per la direcció calculada com la diferencia entre destinació i lloc d'inici amb el component y com la gravetat multiplicada per la meitat de temps per arribar al destí.

```
public bool Passar(GameObject playerToPass){
    if(player!=null && player != playerToPass){
        Debug.Log("passar");
        oldPlayer = player;
        player = null;
        ballMoving = true;
        startTime = currentTime;
        startBallPosition = this.transform.position;
        this.destination = playerToPass.transform.position;
        Vector3 direction = (destination - startBallPosition);
        double distance = Math.Sqrt(Math.Pow(direction.x,2) +
Math.Pow(direction.z,2));
        timeToReach = CalculateTime(distance);
        float vx = (float)(direction.x/timeToReach);
        float vy = (float)(-Physics.gravity.y*timeToReach/2);
        float vz = (float)(direction.z/timeToReach);
        displacement = new Vector3(vx,vy,vz);
        return true;
    }
    else
    {
        return false;
    }
}
```

En el mètode UpdatePosition serveix per actualitzar la posició actual de la pilota a partir de la fòrmula descrita anteriorment.

```
private void UpdatePosition(){
    if(ballMoving){
        float time = currentTime - startTime;
        currentBallPosition = startBallPosition + displacement*time +
Physics.gravity *(float)(Math.Pow(time,2)/2);
        this.transform.position = new Vector3(currentBallPosition.x,
currentBallPosition.y, currentBallPosition.z);
    }
}
```

9.3. Lector de jugades

Per tal de fer més fàcil per al desenvolupador i al jugador la personalització del joc i afegir tàctiques pròpies, es va decidir de crear un lector de jugades, on a partir d'un fitxer JSON es pugui parsejar fàcilment i que la intel·ligència artificial segueixi aquestes tàctiques per tal de reflectir l'estil del jugador.

Per tal de facilitar les coses al desenvolupador, es va decidir de fer ús de fitxers JSON ja que a partir d'eines es pot parsejar fàcilment el codi ja que JSON està format bàsicament per una estructura de tag i valor.

Els documents JSON, per tal que siguin capturats correctament pel lector creat, han de tenir la següent estructura. Ha de tenir obligatòriament els camps *teamState* i el *code* per tal que la Jugada funcioni correctament. També té els camps de *initialPositions, numberPlayerInvolved, whoHasBall* i *cut.*

```
{
	"teamState":"Atacant",
	"initialPositions":"-",
	"numberPlayerInvolved":6,
	"whoHasBall":"-",
	"cut":"-",
	"code":[
		{
			"swim":{
				"positionPlayer":6,
				"position":"J23"
			},
			"swim":{
				"positionPlayer":5,
				"position":"A23"
			},
			"swim":{
				"positionPlayer":4,
				"position":"E20"
			},
			"swim":{
				"positionPlayer":3,
				"position":"J19"
			},
			"swim":{
				"positionPlayer":2,
				"position":"P20"
			},
			"swim":{
				"positionPlayer":1,
				"position":"S23"
			}
		},
		{
			"wait":{
				"positionPlayer":[1,2,3,4,5,6]
			}
		},
		{
			"while":{
				"condition":"True",
				"conditionTrue":{
					"selectPlayerWithBall":"-",
					"passar":{
						"positionPlayer":2
					}
				}
			}

		}
	]
}
```

Un cop obtinguts els valors, el que es va decidir és crear una estructura interna pel contingut de *code* per tal de que fos fàcil poder fer les diferents accions. Es va decidir crear dues classes: Block i CodeBlock. La classe Block és l'encargada de guardar la informació de code i la classe CodeBlock és l'encargada de passar per cada Block i anar executant el contingut.

La classe Block contindria la informació d'una acció. Aquesta acció podria ser de qualsevol mena, com per exemple xutar, passar, nedar, while, if, etc. Per poder manejar fàcilment la informació es va decidir de tenir com a atributs unes relacions amb altres Blocks. Aquestes relacions serien *nextChildTrueBlock, nextChildTrueBlock, next* i *parent*. El perquè de tenir 4 relacions són els següents:

- *nextChildTrueBlock* i *nextChildTrueBlock* són relacions amb altres Blocks per quan el block actual sigui una condició (*if* o *while*). Per tant depenent de la condició, s'ha de seguir un camí o un altre, en aquest cas un Block o un altre.
- *Next* aquest Block s'utilitzaria per les accions següents que no estigués com a Child.
- *Parent* serviria bàsicament per quan ens trobem amb un *while*, on al final de les accions dintre d'un *while*, hem de tornar a la condició.

Si mostrem el codi podem veure com es crea un block, on es passa la informació com a JSONObject, el pare i el següent.

```
public Block(JSONObject source, Block parent, Block next)
{
    this.parent = parent;
    this.next = next;
    if (source.IsArray)
    {
        ExtractInfoBlock(source);
    }
    else
    {
        if (source.IsObject)
        {
            if (source.list.Count > 0)
            {
                ExtractInfoBlock(source);
            }
        }
    }
}
```

Un cop tractat si es un objecte o una llista entraria al següent mètode, on obtindria l'acció i el contingut. També intentaria obtenir tags com *condition, conditionTrue i conditionFalse* per si l'acció és una condició, si no ho fos, quedaria com a null. Si hi fos, llavors faria crides al mètode ExtractChildren ja que significaria que té codi dintre. Aquests blocs els guardaria com a nextChildBlock. Finalment extrauria la informació dels blocks següents.

```
private void ExtractInfoBlock(JSONObject source)
{
    Action = source.keys[0];
    sourceCode = source.list[0];
    if (sourceCode["condition"])
    {
        hasCondition = true;
        loopCondition = sourceCode["condition"];
        if (action == "while")
        {
            isLoop = true;
        }
    }
    if (sourceCode["conditionTrue"])
    {
        nextChildTrueBlock =
ExtractChildren(sourceCode["conditionTrue"]);
    }
    if (sourceCode["conditionFalse"])
    {
        nextChildFalseBlock =
ExtractChildren(sourceCode["conditionFalse"]);
    }
    ExtractBrothers(source);
}
```

Mètode que extreu els Blocks següents al cridat.

```
private void ExtractBrothers(JSONObject source)
{
    Block next1 = null;
    if (source.list.Count > 1)
    {
        for (int i = source.list.Count - 1; i >= 0; i--)
        {
            next1 = new Block(CreateJSONObject(source.keys[i],
source.list[i]), this.parent, next1);
        }
        next = next1;
    }
}
```

Mètode que extreu Blocks que estan a un nivell inferior al que ha cridat.

```
private Block ExtractChildren(JSONObject source)
{
    Block next1 = null;
    for (int i = source.list.Count - 1; i >= 0; i--)
    {
        next1 = new Block(CreateJSONObject(source.keys[i],
source.list[i]), this, next1);
    }
    return next1;
}
```

Un cop obtingut una estructura, cal executar la informació. Això se n'encarrega la classe CodeBlock el qual llegeix la informació del Block actual i a partir de la seva acció executa i tria quin és el següent Block.

El mètode principal seria Execute el qual demana un Block, un equip i una MiniJugada. Aquests dos últims paràmetres sòn per fer les crides de les accions.

```
public static void Execute(Block block, Team team1, MiniJugada
minijugada)
{
    team = team1;
    miniJugada = minijugada;
    ChooseKey(block);
}
```

En aquest mètode, a partir de l'acció que té el Block, es tria quin mètode ha de fer servir.

```
private static void ChooseKey(Block block)
{
    switch (block.Action)
    {
        case "swim":
            Swim(block);
            break;
        case "wait":
            Wait(block);
            break;
        case "selectPlayer":
            player = (int) block.SourceCode.n;
            break;
        case "while":
            While(block);
            break;
        case "passar":
            Passar(block);
            break;
        case "xutar":
            Xutar(block);
            break;
        case "if":
            If(block);
            break;
        case "selectPlayerWithBall":
            player = team.GetPositionPlayerWithBall();
            break;
    }
}
```

El mètode Passar el que faria és obtenir la posició on ha de fer el passe del codi font i esperar que la pilota arribi i no aturar tota l'aplicació.

```
private static void Passar(Block block)
{
    Debug.Log("passe");
    if (player != -1)
    {
        int playerToPass = (int)block.SourceCode["positionPlayer"].n;
        if (!team.PassarPilota(player, playerToPass))
        {
            Debug.Log("error in pass ball");
        }
        team.WaitUntilBallReachDestination();
        miniJugada.Waiting = true;
    }
}
```

En el mètode If trobem que és una condició, per tant ha de comprovar si és certa o falsa. A partir d'aquí carrega blocks a la cua que trobem a la MiniJugada i les executa.

```
private static void If(Block block)
{
    bool itsTrue = LookCondition(block.LoopCondition);
    if (itsTrue)
    {
        miniJugada.LoadUntilCondition(block.NextChildTrueBlock);
    }
    else{
        if (block.NextChildFalseBlock != null)
        {
            miniJugada.LoadUntilCondition(block.NextChildFalseBlock);
        }
        else
        {
            if (block.Next != null)
            {
                miniJugada.LoadUntilCondition(block.Next);
            }
            else
            {
                miniJugada.LoadUntilCondition(block.Parent);
            }
        }

    }
}
```

En aquest mètode també trobem una condició que carrega tot el contingut del while si és cert. Si és falsa la condició, el que fa és carregar els blocks següents seus o al pare.

```
private static void While(Block block)
{
    bool itsTrue = LookCondition(block.LoopCondition);
    if (itsTrue)
    {
        miniJugada.LoadUntilCondition(block.NextChildTrueBlock);
    }
    else
    {
        if (block.Next != null)
        {
            miniJugada.LoadUntilCondition(block.Next);
        }
        else
        {
            miniJugada.LoadUntilCondition(block.Parent);
        }
    }
}
```

Aquest mètode conté totes les condicions per ser avaluades. Quan es troba una nova condició s'introdueix en aquest mètode.

```
private static bool LookCondition(JSONObject jSONObject)
{
    bool result = false;
    switch (jSONObject.str)
    {
        case "HasBall":
            result = team.HasPlayerBall(player);
            break;
        case "True":
            result = true;
            break;
    }
    return result;
}
```

Per finalitzar trobem el mètode Wait el qual llegeix de quines posicions de jugador ha d'esperar i no executa cap mena de block. Així surt de l'execució del Codeblock i no bloqueja l'aplicació.

```
private static void Wait(Block block)
{
    JSONObject source = block.SourceCode["positionPlayer"];
    if (source.IsArray)
    {
        for (int i = 0; i < source.list.Count; i++)
        {
            int positionPlayer = (int)source.list[i].n;
            team.SetAlertPlayer(positionPlayer, true);
        }
    }
    else
    {
        int positionPlayer = (int)source.n;
        team.SetAlertPlayer(positionPlayer, true);
    }
}
```

Capítol 10. Implantació i resultats

10.1. Procés de desenvolupament

Un cop es va triar la temàtica del videojoc, es defineixen què es farà i quines seran les fites. Llavors es comença a cercar un motor de videojocs més oportú per al videojoc. Un cop triat Unity, es mira dels llenguatges de programació disponibles, quina és la més fàcil d'adaptar-se. A partir d'aquí es comença a fer tutorials de Unity per aprendre les funcionalitats més bàsiques i l'API.

A continuació es va començar a implementar els diferents moviments pel jugador que es controla. Llavors es programa els passes, comprovant les físiques del programa. A partir d'aquí es canvia de temàtica i es passa a crear l'aigua. On un cop feta la primera versió, ens trobem que la CPU és molt lenta per poder moure la malla de l'aigua i es decideix programar-ho per GPU. Amb això s'aprèn programar shaders i al cap de forces intents s'acaba fent funcionar amb una malla és gran i sense obtenir cap baixada de velocitat.

A partir d'aquí es comença a programar la intel·ligència artificial, on es pren la decisió de fer un sistema de control de jugades per facilitar la creació de noves jugades i les variants amb fitxers de text per les diferents fases d'un partit de waterpolo. Llavors es decideix que cada jugador tingui una petita intel·ligència artificial per tal de poder decidir entre executar una acció que li demana una jugada, intenta moure's o esperar per fer-la.

Un cop acabat, en aquest punt es decideix amb el temps restant fer unes interfícies bàsiques i s'acaba d'arrodonir la memòria.

10.2. Legislació i normativa

Aquest projecte no conté cap problema amb la legislació i la normativa vigent.

Pel que fa la llei orgànica de protecció de dades de caràcter personal (LOPD), aquesta aplicació no la vulnera ja que no es guarda en cap moment dades de l'usuari.

Amb la llei de serveis de la societat de la informació i comerç electrònic (LSSICE), aquesta aplicació tampoc la vulnera ja que no constitueix una activitat econòmica.

10.3. Aplicació resultant

Totes les tasques s'han realitzat correctament, excepte de posar models dels jugadors, animacions, sons i una piscina modelada.

A continuació es mostren diferents captures de pantalla del videojoc, mostrant les diferents parts del joc.

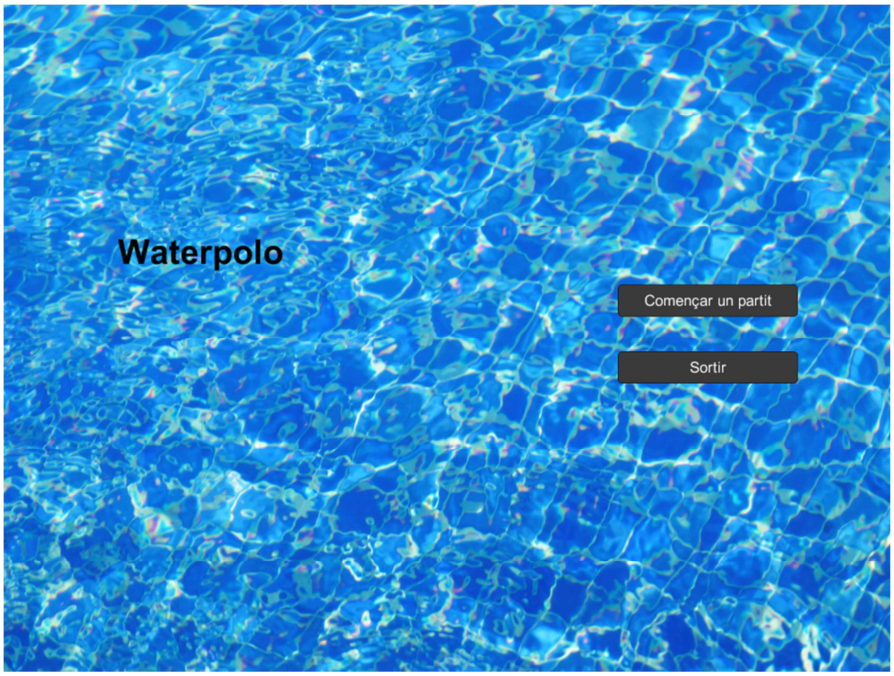


Figura 10.1: Menú principal del joc.

En la figura 10.1 es pot veure el menú principal del joc on, bàsicament, permet sortir de l'aplicació i començar un partit.

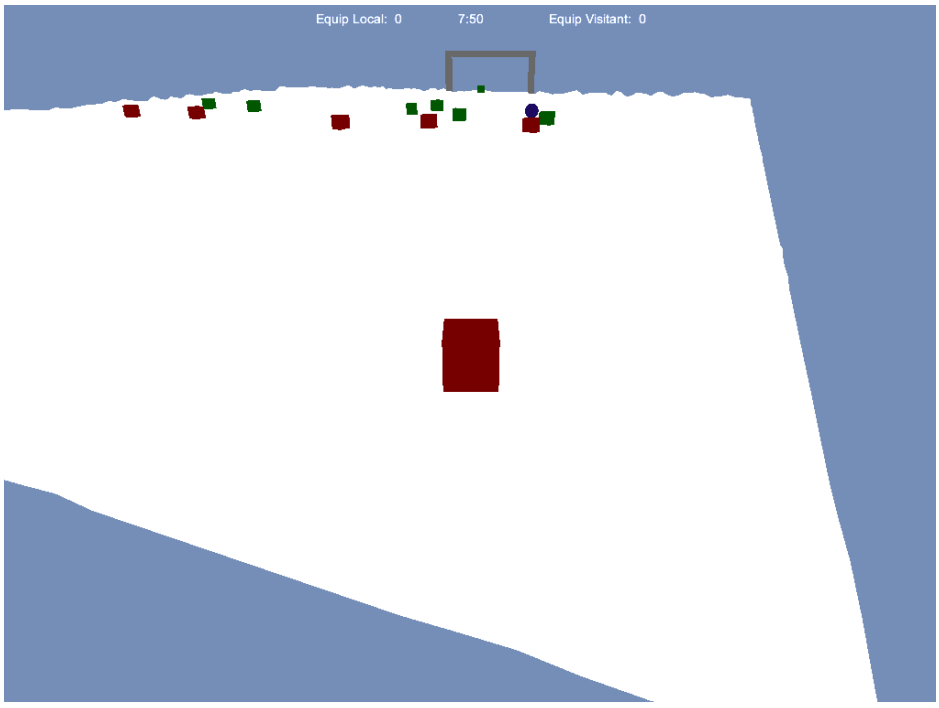


Figura 10.2: Interfície en un partit.

En la Figura 10.2 es pot veure l'inici d'un partit on els dos equips han anat a buscar la pilota al centre de la piscina i en aquest cas l'equip local(vermell) ha agafat la pilota i comença l'atac.

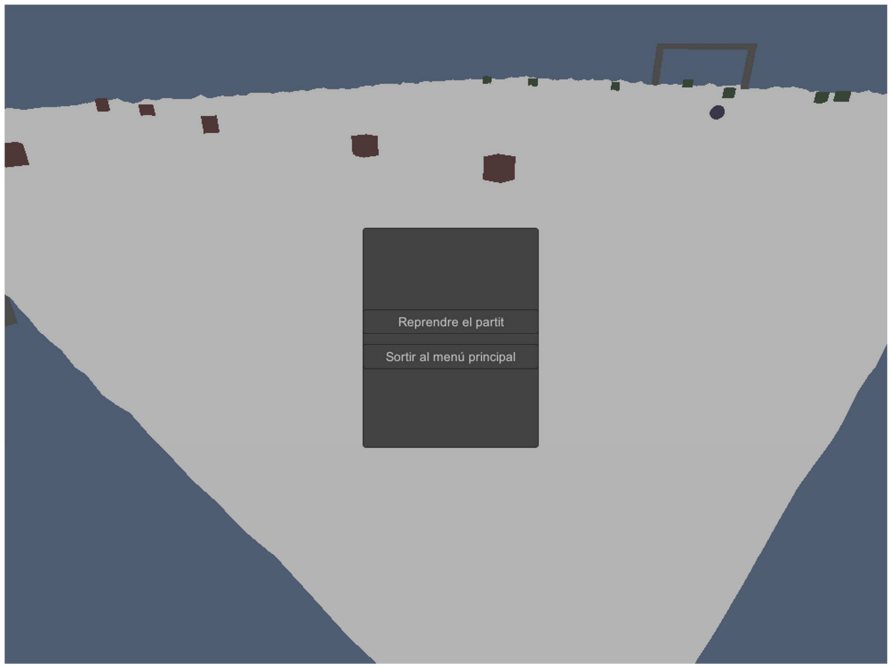


Figura 10.3: Interfície durant una pausa en un partit.

Sempre es pot accedir a la interfície de pausa mentre es juga el partit. Es mostrarà una interfície com el de la figura 10.3. En aquest menú es pot reprendre el partit o es pot sortir del partit i anar al menú principal.

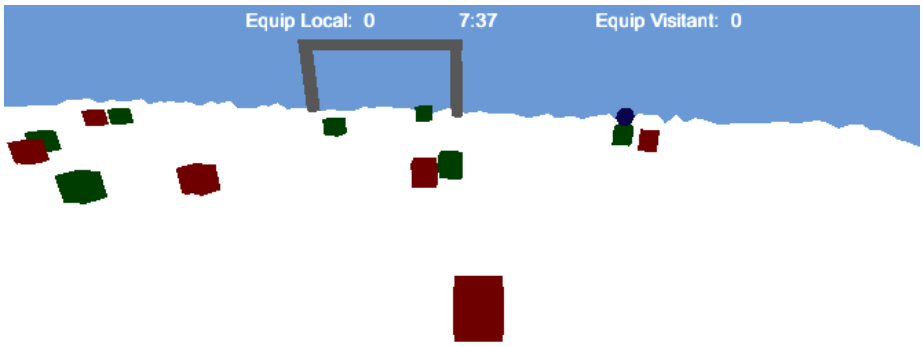


Figura 10.4: Jugador roba la pilota.

En la figura 10.4 es pot veure com l'equip visitant(verd) ha robat la pilota del equip local. A partir d'aquí s'inicia el contraatac del equip visitant cap a la porteria contrària.

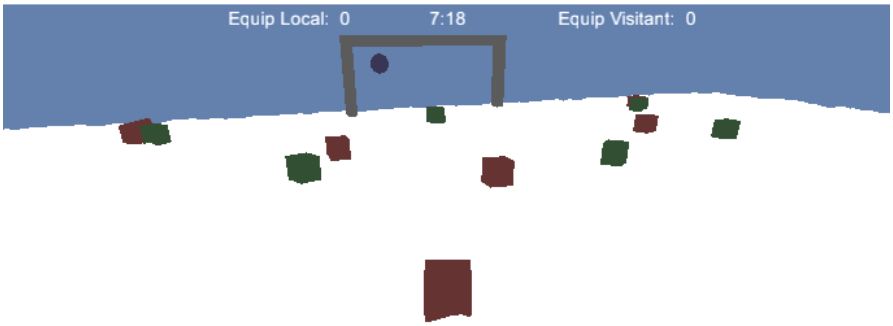


Figura 10.5: Passe a un company del mateix equip.

En la figura 10.5 es pot veure com es fa un passe entre els dos companys del equip visitant.

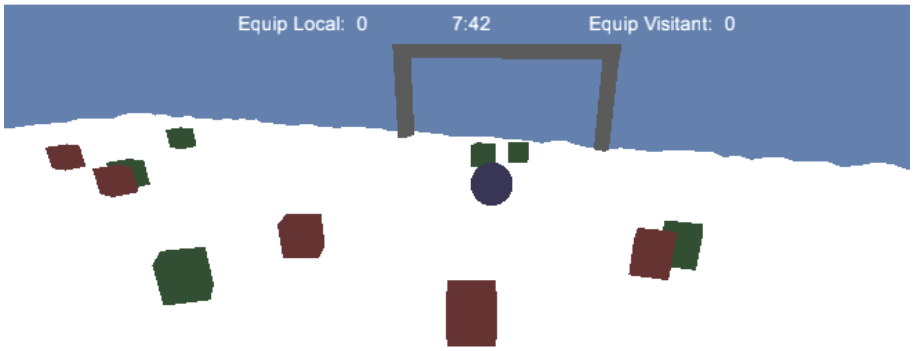


Figura 10.6: Xut a porteria.

En la figura 10.6 es mostra un xut a porteria que al final el porter el pararà.

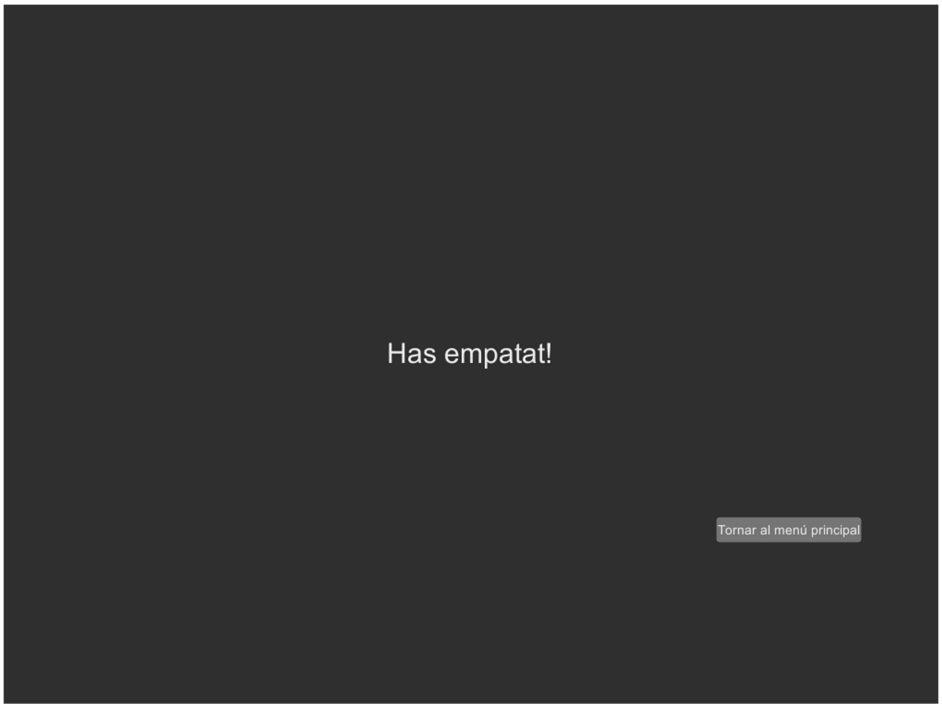


Figura 10.7: Interfície un cop hem acabat el partit i s’ha empatat.

En la figura 10.7 es pot veure la interfície final quan s’ha acabat el partit. Depenent del resultat et pot dir que has guanyat, perdut o empatat. També trobem un botó per tornar al menú principal.

Capítol 11. Conclusions

11.1. Planificació real

Com es pot observar en la figura següent, la planificació final no ha estat l'esperada. Hi ha hagut una desviació important respecte la part del moviment del jugador, ja que s'hi ha inclòs dintre aquest apartat l'animació de l'aigua de la piscina.

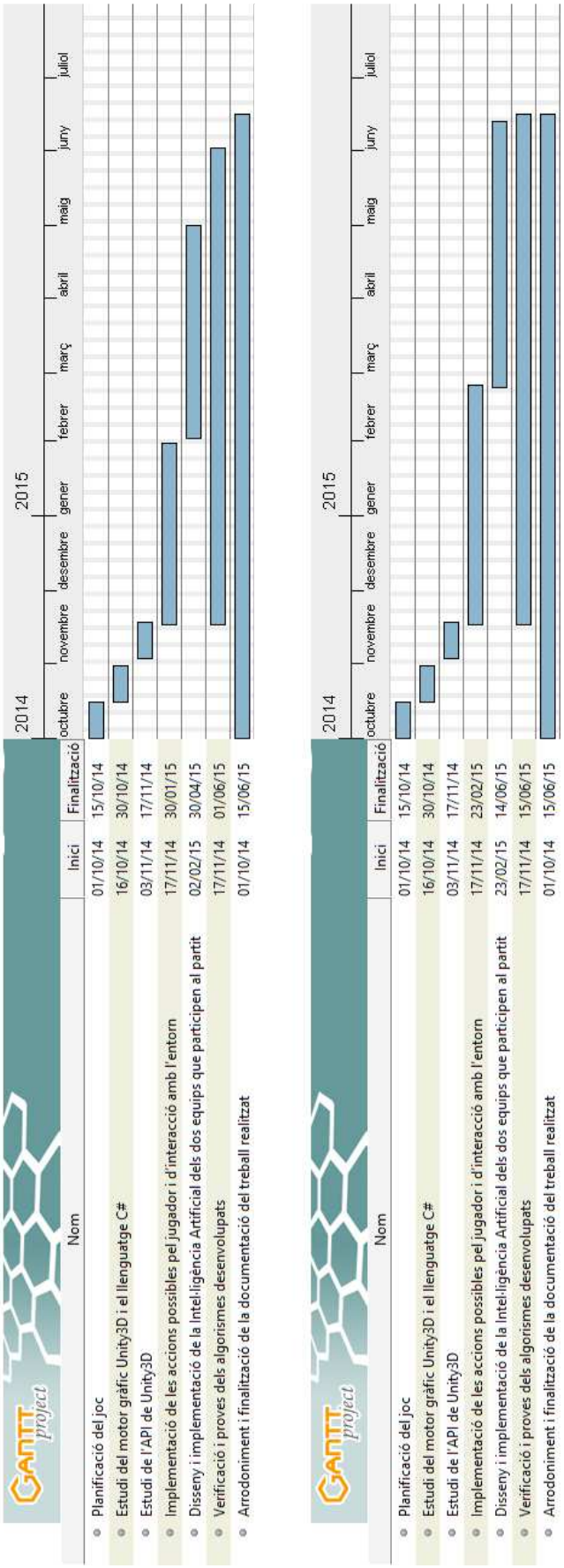


Figura 11.1: Planificació estimada i final.

11.2. Conclusions

Les conclusions a les que s'han arribat durant l'elaboració del projecte de final de grau són les següents:

- No s'ha pogut finalitzar correctament el projecte en la part que afecta el modelatge, animacions i so. S'ha dedicat molt de temps en la programació de la intel·ligència artificial i del moviment de l'aigua. Al final s'ha deixat la part més artística per a un treball futur.
- És important dedicar molt de temps en saber quines eines són les millors per cada tipus de projecte que es vol fer. Si no s'hi dedica el temps suficient per esbrinar quina eina és la millor, al final es gastarà el doble de temps intentant suplir les deficiències de la eina triada.
- El llenguatge C# és molt fàcil d'aprendre si es tenen coneixements de Java i C++. Caldria esbrinar quin llenguatge és el més eficient per programar en Unity.
- Fer una intel·ligència artificial distribuïda amb els jugadors i amb l'equip ha estat una bona decisió. Permet desacoblar les decisions d'equip, de quina jugada es vol fer, amb les petites decisions que puguin fer cada jugador. D'aquesta manera reflecteix la realitat del Waterpolo.
- Aquest és un projecte per desenvolupar amb un equip de programació. Si es fa amb una persona, el desenvolupament és molt lent, comparat amb un equip.

Capítol 12. Treball futur

Les possibles ampliacions i millores d'aquest projecte es poden dividir en dos aspectes: artístic i programàtic.

Per la part artística es poden fer les següents millores:

- Fer el modelatge i animació dels jugadors.
- Fer el modelatge i animació dels àrbitres.
- Fer el modelatge i animació del públic.
- Modelar diferents piscines.
- Crear un conjunt d'equips amb el corresponent escut o emblema del seu equip.
- Afegir banda sonora i efectes de so.

Per la part de programació es poden fer les següents millores:

- Poder fer partides online.
- Implementar les faltes i expulsions.
- Incorporar decisions dels àrbitres.
- Posar joc brut entre els jugadors.

Capítol 13. Bibliografia

En aquest capítol es mostren les fons consultades per el desenvolupament del projecte.

13.1. Pàgines web

Photoshop [Internet]. San Jose: Adobe Systems Incorporated; [citada 20 novembre 2014]. Disponible a: http://www.photoshop.com/products/photoshop

Maya [Internet]. San Rafael: Autodesk, Inc.; [citada 20 novembre 2014]. Disponible a: http://www.autodesk.com/products/maya/overview

Isfe.eu [Internet]. Brussel·les: Interactive Software Federation of Europe; [citada 15 maig 2015]. Disponible a: http://www.isfe.eu/industry-facts/statistics

Forbes [Internet]. Nova York: Forbes publishing; [citada 15 maig 2015]. Disponible a: http://www.forbes.com/sites/johngaudiosi/2012/07/18/new-reports-forecasts-global-video-game-industry-will-reach-82-billion-by-2017/

JSONObject [Internet]. Defective Studios; [citada 19 gener 2015]. Disponible a: http://wiki.unity3d.com/index.php?title=JSONObject

Wikipedia [Internet]. Jimmy Wales; [citada 20 novembre 2014]. Disponible a: https://ca.wikipedia.org/wiki/Portada

Unity [Internet]. San Francisco: Unity Technologies; [citada 20 novembre 2014]. Disponible a: http://unity3d.com/

Unity – Scripting API[Internet]. San Francisco: Unity Technologies; [citada 20 novembre 2014]. Disponible a: http://docs.unity3d.com/ScriptReference/

Unity – Manual [Internet]. San Francisco: Unity Technologies; [citada 20 novembre 2014]. Disponible a: http://docs.unity3d.com/Manual/index.html

Stack Overflow [Internet]. Stack Exchange, Inc.; [citada 20 novembre 2014]. Disponible a: https://stackoverflow.com/

C# Library [Internet]. Redmond: Microsoft Corporation; [citada 20 novembre 2014]. Disponible a: https://msdn.microsoft.com/en-us/library/

13.2. Llibres i articles

Buckland M. Programming Game AI by Example. 1a ed. Plano: Wordware Publishing, Inc; 2005.

Millington I, Funge J. Artificial intelligence for games. 2a ed. Burlington: Morgan Kaufmann; 2009.

MacCabe A, Trevathan J. Artificial Intelligence in Sports Prediction. Townsville: James Cook University, Belfast: MAIT Technologies; 2008[citada 22/11/2014]. Disponible a: http://www.researchgate.net/profile/Jarrod_Trevathan/publication/220841301_Artificial_Intelligence_in_Sports_Prediction/links/00b7d5154fe649278f000000.pdf

Mozgovoy M, Umarov I. Believable Team Behavior: Towards Behavior Capture AI for the Game of Soccer. Aizu-Wakamatsu: University of Aizu, St Petersburg: TruSoft Int'l Inc; 2011 [citada 22/11/2014]. Disponible a: http://web-ext.u-aizu.ac.jp/~mozgovoy/homepage/papers/mu11a.pdf

Capítol 14. Manual d'usuari i/o instal·lació

14.1. Instal·lació

El videojoc és un executable per Windows, el qual no requereix cap mena d'instal·lació de software addicional ni de la pròpia aplicació. Al executar el programa i després d'entrar les opcions gràfiques que volem pel joc, ja estarem directament al menú principal.

14.2. Controls

Els controls del videojoc són els següents:

- Moure's pel camp de joc:

- Saltar:

- Demanar pilota:

- Sprintar:

- Robar pilota:

- Pausar partida:

- Passar:

- Xutar: