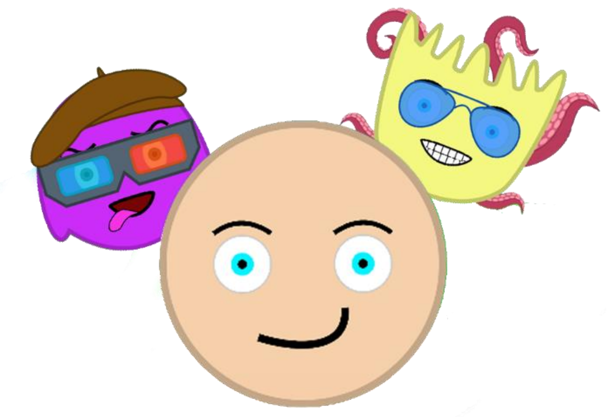
TREBALL FINAL DE GRAU

DESENVOLUPAMENT D'UN VIDEOJOC DE TIPUS "MASCOTA VIRTUAL"

Santi van Gelderen



Document: Memòria

Estudi: Grau en Disseny i Desenvolupament de Videojocs

Tutor: Gustavo Patow

Departament: Informàtica, Matemàtica Aplicada i Estadística

Àrea: Llenguatges i Sistemes Informàtics

Convocatòria: Juny 2022

Índex

1. Introducció i objectius 6
   1.1. Introducció 6
   1.2. Motivacions 6
   1.3. Motivacions personals 7
   1.4. Distribució de tasques 7
2. Estudi de viabilitat 9
   2.1. Recursos necessaris i viabilitat 9
      2.1.1. Recursos tècnics 9
      2.1.2. Recursos humans 9
      2.1.3. Recursos econòmics 10
   2.2. Estudi de mercat 11
      2.2.1. Estat de l'art 11
      2.2.2. Model de negoci 13
   2.3. Públic objectiu 14
3. Planificació 15
   3.1. Metodologia de treball 15
   3.2. Eines de treball i programari 16
      3.2.1. Motor de joc 16
      3.2.2. Editor de codi 17
      3.2.3. Programari artístic 17
      3.2.4. Altres 18
4. Marc de treball i conceptes previs 19
   4.1. Característiques principals dels videojocs per infants 19
   4.2. Conceptes previs del motor de joc 20
      4.2.1. Gestió d'elements i objectes a Unity 20
      4.2.2. Gràfics i físiques 2D 21
      4.2.3. Scripting 22
      4.2.4. Llibreries utilitzades 23
      4.2.5. Altres terminologies utilitzades 23
5. Disseny del videojoc 24
   5.1. Disseny de la mascota 24
   5.2. Disseny d'espais 26
   5.3. Disseny de minijocs 27
      5.3.1. Tir al Plat 28
      5.3.2. Plataformes 28

5.3.3. Escalada 29
5.3.4. Memoritzar 29
5.4. Economia del joc 30
5.5. Mecàniques 31
5.6. Interfícies 32
5.6.1. HUD 32
5.6.2. Interfícies específiques per a cada espai 33
5.7. Marca identificadora 34
5.7.1. Nom 34
5.7.2. Tipografia 34
5.7.3. Logotip 35
5.8. Estil artístic 36
6. Implementació 36
6.1. Mascota 37
6.1.1. Estructura 37
6.1.1.1. Atributs 38
6.1.1.2. Estat 38
6.1.1.3. Visual 39
6.1.2. Animacions 40
6.1.3. Modificacions gràfiques puntuals 43
6.1.4. Prefab Final 45
6.2. Espais 46
6.2.1. Exterior 46
6.2.2. Interior 47
6.2.3. Laboratori 49
6.2.3.1. Objectes de l'espai 49
6.2.3.2. Funcionalitats 50
6.2.3.2.1. Canvi de forma 51
6.2.3.2.2. Canvi de mida 53
6.2.3.2.3. Canvi de posició 54
6.2.3.2.4. Mostrar i amagar complements 57
6.2.3.3. Distribució de les parts 59
6.2.3.3.1. Modificació del cos 60
6.2.3.3.2. Modificació de les celles 60
6.2.3.3.3. Modificació dels ulls 61
6.2.3.3.4. Modificació de la boca 62

6.2.3.4. Animacions de panells i menús 63
6.2.4. Cuina 65
6.2.4.1. Objectes de l'espai 65
6.2.4.2. Funcionalitats 66
6.2.4.2.1. Comprar menjar 66
6.2.4.2.2. Alimentar a la mascota 72
6.2.4.3. Distribució de les funcionalitats 75
6.2.4.3.1. Panell d'alimentar a la mascota 76
6.2.4.4. Animacions de panells i menús 77
6.2.5. Lavabo 78
6.2.5.1. Objectes de l'espai 78
6.2.5.2. Funcionalitats 79
6.2.6. Habitació 81
6.2.6.1. Objectes de l'espai 81
6.2.6.2. Funcionalitats 82
6.2.6.2.1. Comprar complements 82
6.2.6.2.2. Vestir a la mascota 90
6.2.6.2.3. Posar a dormir a la mascota 91
6.2.6.3. Distribució de les funcionalitats 93
6.2.6.3.1. Panell de comprar complements 93
6.2.6.3.2. Panell de vestir a la mascota 94
6.2.6.4. Animacions de panells i menús 95
6.2.7. Espai de minijocs 96
6.3. Minijocs 98
6.3.1. Tir al plat 98
6.3.2. Escalada 101
6.3.3. Memoritzar 104
6.3.4. Plataformes 109
6.4. Canvi d'idioma 112
6.5. Menús 116
6.6. Interfícies 120
6.6.1. HUD 120
6.6.2. Interfícies dels minijocs 123
6.6.2.1. Interfícies de Tir al Plat 125
6.6.2.2. Interfícies d'Escalada 126
6.6.2.3. Interfícies de Memoritzar 127

6.6.2.4. Interfícies de Plataformes 127
6.6.3. Espais amb interfícies 127
6.7. Inici del videojoc 131
6.8. Valors dels elements comprables 132
7. Proves i resultats 133
8. Conclusions 136
9. Treball futur 138
10. Bibliografia 139
11. Annexos 141
11.1. Taula Aliments 141
11.2. Taula complements 145
11.3. Vídeo 146

1. Introducció i objectius

1.1. Introducció

Vivim un moment a la societat on els infants han nascut i crescut amb un dispositiu mòbil sota el braç, el qual ofereix un seguit de funcions en diferents àmbits com l'educació, la informació, la interacció amb altres persones, etc. Tot i això, quan aquests nens i nenes estan a casa, fan servir aquests dispositius en la gran majoria de temps per jugar i entretenir-se, a causa de que és la eina que més acostumats estan a fer servir i perquè ofereix un ampli ventall d'opcions, tenint milers i milers de jocs disponibles per qualsevol tipus de dispositiu, per totes les edats i de qualsevol temàtica en concret. Un d'elles és la de videojocs de mascotes virtuals, on generalment l'usuari té una mascota pròpia la qual pot personalitzar, cuidar, vestir, jugar amb ella, etc. Aquests tipus de videojocs van anar creixent exponencialment fins l'arribada de *Pou*, el joc que va dominar el mercat de videojocs d'aquest mateix sector i referència número 1 en jocs per infants. A partir d'aquí, els jocs que han anat sortint de mascotes virtuals han seguit sempre els mateixos patrons, amb clares similituds a *Pou* i sense destacar en res que no tingués aquest videojoc.

És per això que considero que desenvolupar un videojoc d'aquest tipus em pot aportar l'experiència de desenvolupar un joc des de zero en solitari, i alhora suposar-me un repte en el qual hauré de trobar un aspecte on el meu producte pugui marcar la diferència en comparació a un gegant com pot ser *Pou.*

1.2. Motivacions

L'objectiu principal d'aquest projecte és desenvolupar un videojoc sencer, amb una suficient qualitat com per poder-lo publicar a qualsevol pàgina o aplicació i endinsar-lo dins del mercat actual. Es buscarà també potenciar al màxim una de les mecàniques en concret, fent que aquesta sigui la referència principal del videojoc i encarregada de fer destacar el producte a l'hora de comparar-lo amb altres del mateix sector.

Concretament, els objectius són els següents:

- Aprendre a dominar *Unity Engine*.
- Aprendre a dominar *Inkscape* i *Adobe Photoshop* per crear elements 2D pel videojoc.
- Estudiar, identificar i entendre les mecàniques principals que utilitzen els videojocs de mascotes virtuals.
- Treballar el balanç del joc.
- Dissenyar un sistema que permeti crear un videojoc que es pugui ampliar de forma senzilla a través d'una bona estructura i una bona organització dels recursos.

1.3. Motivacions personals

En el moment en que vaig començar aquest projecte treballava com a professor i coordinador d'una acadèmia on s'ensenya programació a infants. És per això que vaig decidir que nens i nenes d'edats similars als meus alumnes fossin el públic objectiu del videojoc, per poder compartir amb ells i elles el seguiment del projecte i poder rebre crítiques i suggeriments en tot moment per part seva. També tinc altres motivacions o objectius personals com poden ser el següents:

- Poder experimentar tot el procés d'inici a fi del desenvolupament d'un videojoc en solitari.
- Descobrir on són els meus límits actuals com a programador i veure en quins aspectes puc millorar.
- Posar en pràctica diferents tècniques i consells adquirits durant el grau per desenvolupar un videojoc.
- Finalitzar el grau amb un projecte d'alta qualitat i poder-lo comparar amb videojocs del mateix sector que estiguin al mercat actual.

1.4. Distribució de tasques

Normalment un videojoc de les magnituds que m'agradaria fer a mi sol ser desenvolupat per un equip amb diverses persones per a cada sector, com poden ser el disseny, la creació d'elements artístics, la programació, etc. En aquest cas però, aquest projecte serà desenvolupat només per una sola persona, qui haurà de treballar totes les diferents tasques necessàries per crear un videojoc des de zero. A continuació es mostra un quadre (Taula 1) amb la distribució percentual de quant valor i temps es dedicarà a cada un dels elements nombrats i, tot seguit, s'explica el motiu d'aquests percentatges:

Narrativa	0%
Estètica	30%
Mecàniques	30%
Tecnologia	40%

Taula 1: Quadre de distribució de tasques

Primerament, es tractarà d'un videojoc on cada usuari tindrà la seva pròpia mascota, la qual podrà personalitzar de la manera que desitgi i poder fer amb ella un seguit d'activitats. Per tant estem parlant d'un videojoc on no cal que es dediqui temps ni esforç en crear una història o un context previ, a causa de que no és necessari.

Després, pel que fa a l'estètica, a priori pot semblar un dels punts més importants d'un videojoc d'aquest tipus, però personalment no domino tant la part de creació de continguts visuals com per basar el projecte en fer-lo el màxim artístic possible, tot i que li hauré de dedicar temps de totes maneres perquè el resultat final sigui el millor possible.

Seguidament, les mecàniques del videojoc són un altre aspecte molt important d'aquest videojoc, ja que no es pot fugir de les més comunes en l'àmbit dels jocs de mascotes virtuals, però també s'ha de ser original per destacar en algun aspecte. Aquí entren tan la part de disseny com d'implementació, i s'haurà de tenir en compte les mecàniques dels minijocs, les de creació i modificació de la mascota i les del benestar d'aquesta.

Finalment, la part de tecnologia és l'aspecte que pren més protagonisme a causa que la programació és la part que més m'agrada del desenvolupament i en la que hi tinc més facilitats per aconseguir els objectius que em puc arribar a posar. El meu repte personal en aquesta part és que la programació del videojoc sigui el més versàtil possible, és a dir, que no siguin un seguit de mal de caps el fet de voler afegir funcionalitats o mecàniques, i que el joc es pugui ampliar de manera senzilla un cop la base d'aquest estigui desenvolupada. D'altra banda, la part de xarxes és inexistent pel fet de que no desenvoluparé un joc en línia, però la meva intenció és que es tracti d'un joc per jugar amb dispositius mòbils com *smartphones* o tauletes, així que el tipus de desenvolupament pot variar una mica en funció d'aquest aspecte.

2. Estudi de viabilitat

Abans d'iniciar el desenvolupament del projecte, cal veure si aquest és viable a tots els nivells. A continuació es calcularan i comentaran els recursos necessaris per desenvolupar el videojoc, es farà un estudi de mercat per analitzar la cabuda que pot tenir el joc dins del mercat actual, i s'estudiarà el model de negoci i s'analitzarà el públic objectiu.

2.1. Recursos necessaris i viabilitat

A continuació s'exposen els diferents recursos i eines utilitzades pel desenvolupament d'aquest videojoc, incloent parts tècniques, econòmiques i personals.

2.1.1. Recursos tècnics

Pel desenvolupament del projecte no es requereixen grans recursos tècnics, sinó que es pot fer amb un ordinador de gama mitja-alta i un seguit de programes, la gran majoria d'ells gratuïts. D'aquesta manera, per desenvolupar aquest videojoc s'han fet servir els següents recursos:

- **Portàtil ASUS Rog Strix G731GT**: Utilitzat com a eina principal per a desenvolupar el videojoc i per realitzar proves. Característiques principals:
    - **Processador:** Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
    - **Memòria RAM:** 16,0 GB
    - **Gràfica:** NVIDIA GeForce GTX 1650 GDDR 5 @ 4GB (128 bits)

- **Unity v. 2020.3.22f1:** Motor de joc utilitzat per desenvolupar el joc.
- **InkScape:** Programa gratuït de dibuix vectorial, utilitzat per realitzar la gran majoria d'elements visuals del joc.
- **Visual Studio 2019:** Editor de codi gratuït, utilitzat pel desenvolupament del joc.
- **Adobe Photoshop:** Editor d'imatges de pagament, utilitzat per complementar a InkScape podent retocar alguns elements de manera més senzilla i obtenint grans resultats a nivell de definició i qualitat.

2.1.2. Recursos humans

Normalment quan es vol desenvolupar un projecte d'aquestes magnituds i característiques, hi solen treballar persones expertes en diferents sectors, cobrint d'aquesta manera totes les fases del desenvolupament del videojoc. Aquests sectors i tasques són els següents:

- **Disseny:** Encarregats d'idear com serà el joc, definint objectius, narrativa, mecàniques i qualsevol element que ha d'aparèixer al projecte.
- **Programació:** La seva feina és implementar els aspectes definits pels dissenyadors a través del desenvolupament de codi.

- **Art:** Encarregats de l'estètica del joc, la seva feina és dissenyar i crear tots els elements visuals que poden aparèixer al videojoc.
- **So:** Creen tot l'apartat auditiu, és a dir, implementen sons i música del joc.

Tal i com s'ha comentat prèviament, aquest projectes es desenvoluparà només per una persona, tot i que, si és necessari, s'optarà per demanar ajuda a tercers en alguns dels aspectes que s'han esmentat. També per fer possible aquest desenvolupament es podrà recórrer en algun moment a aspectes implementats en altres projectes que puguin ser útils en aquest videojoc.

2.1.3. Recursos econòmics

A l'Apartat 2.1.1, on es comentaven els recursos tècnics necessaris per desenvolupar aquest projecte, s'ha pogut veure que la majoria de programes que s'utilitzen són gratuïts, i en el meu cas els programes que són de pagament o la maquinària que s'utilitza i té un cost econòmic ja els tinc, així que per aquest projecte no hauré de realitzar cap pagament addicional per obtenir algun servei necessari. De totes maneres, a continuació es mostra un llistat (Taula 2) amb els preus aproximats dels recursos tècnics necessaris per desenvolupar el projecte, i d'aquesta manera contextualitzar una hipotètica inversió que s'hauria de fer si no es disposés de cap recurs dels que hem esmentat prèviament.

RECURS	COST
Portàtil ASUS Rog Strix G731GT	~1200
Unity v. 2020.3.22f1	0
InkScape	0
Visual Studio 2019	0
Adobe Photoshop	12.09/mes

Taula 2: Costos dels recursos necessaris

D'altra banda, a l'Apartat 2.1.2 es mostraven els recursos humans necessaris per un projecte d'aquestes característiques, i encara que en aquest cas el projecte es vagi a desenvolupar per una sola persona, farem com amb els recursos tècnics i es mostrarà a continuació un llistat (Taula 3) amb els costos per hora mitjans de professionals de cada sector (dades extretes de *www.payscale.com*).

SECTOR	COST
Disseny	~16€/hora
Programació	~20€/hora
Art	~17€/hora
So	~12€/hora

Taula 3: Sous mitjans de professionals en cada sector

Tenint la planificació del projecte enllestida, podríem calcular el cost del desenvolupament multiplicant les hores de feina de cada professional pel seu salari mencionat a la taula anterior. Tot i això, tal i com s'ha comentat prèviament, el cost total d'aquest projecte en concret serà de 0€, ja que el realitzaré jo sol en tots els aspectes.

2.2. Estudi de mercat

Un cop hem analitzat que el projecte és viable a nivell econòmic, el següent pas és saber si aquest tindria cabuda dins del mercat actual i estudiar els jocs similars per veure si podem destacar en algun aspecte en concret.

2.2.1. Estat de l'art

S'utilitzarà el motor de recerca *Google* per buscar llistats de videojocs similars a la nostra proposta. És important que s'estudiï de forma exhaustiva els millors exponents en l'àmbit de videojocs de mascota virtual, per saber quines són les principals mecàniques que aquests utilitzen, quins aspectes criden més l'atenció, quins altres es poden potenciar més, etc. A continuació es comentaran breument els jocs analitzats i els aspectes més rellevants de cada un, igual que les coses que es podrien millorar en cada cas.

Pou

Tal i com s'ha comentat a la introducció de la memòria, *Pou* és el màxim referent i el joc per excel·lència del sector. Després d'estudiar-lo a fons es poden extreure moltes coses positives d'aquest videojoc, motiu pel qual ha tingut tan èxit, però també hi ha algun aspecte on pot millorar moltíssim.

D'aspectes positius es poden destacar el fet de que és molt intuïtiu, gràcies a que les interfícies són molt clares i pensant en que això és un punt clau a causa de que el públic objectiu són infants, a qui se'ls hi pot fer molt avorrit el joc si no entenen el funcionament bàsic. També cal destacar que té molts minijocs, que la mascota canvia d'aspecte en funció del seu estat i que els espai no estan molt carregats, fent que el joc sigui molt agradable a la vista de l'usuari (Veure Figures 1 i 2).

D'altra banda, personalment no diria que el joc és perfecte perquè en el temps dedicat a provar el joc i estudiar-lo a fons he pogut identificar alguna cosa que no m'ha acabat de convèncer, com per exemple el fet de que no se li pugui posar nom a la mascota. També té alguns canvis d'estètica, sobretot en la interfície, que li fan perdre coherència al joc, i finalment també m'agradaria afegir que quan es puja de nivell no hi ha cap missatge a l'usuari ni cap recompensa.

Figura 1: Logotip de Pou

Figura 2: Diferents espais de Pou

My Boo

My Boo també és un videojoc molt important en el sector de jocs de mascotes virtuals, tenint més d'un milió de descàrregues i molt bones opinions dels usuaris. Després d'estudiar-lo també se n'ha pogut extreure molts aspectes positius, com el fet de que la mascota té un ventall d'opcions molt ampli que fan que l'usuari la pugui personalitzar de moltes maneres diferents. (Veure Figura 3) Pel que fa a l'estètica també és un punt a destacar, ja que segueix la mateixa línia d'estil en tot el joc fent-lo molt coherent. També es pot destacar la cinemàtica que apareix quan es puja de nivell i el fet de que es pugui posar un nom a la mascota. (Veure Figura 4).

Tot i això també hi ha coses que es poden millorar, com el fet que els espais estan molt carregats amb botons i objectes. També cal dir que, per molt que se li pugui posar un nom a la mascota, aquest gairebé mai es veu i la gestió d'espais és molt similar a la de *Pou*, fent pensar que pugui ser una còpia.

Figura 3: Logotip de My Boo

Figura 4: Diferents espais de My Boo

Moy 7

Un altre joc del sector que té moltes descàrregues és *Moy 7*, tot i que aquest és el típic joc que té crítiques molt diverses, agrada molt o no agrada gens. Després d'estudiar-lo i analitzar-lo, he pogut veure el motiu: *Moy 7* té moltes coses bones, com la gran varietat d'espais que té, una estètica molt coherent, el fet de que se li pugui posar nom a la mascota o la cinemàtica de quan es puja de nivell. Però tot i així, té alguns aspectes negatius que són molt rellevants en el joc, motiu el qual rep moltes crítiques dolentes, perquè són aspectes els quals molta gent hi pot arribar a donar molta importància en un joc d'aquest estil. (Veure Figures 5 i 6).

Un dels aspectes a millorar és la càrrega d'espais, que es veuen molt plens com a *My Boo* i pot arribar a ser estressant per l'usuari. El joc en molts aspectes és molt poc intuïtiu, costa molt entendre totes les mecàniques del videojoc i això agafa molta importància quan parlem d'un joc que a priori és per nens i nenes. També caldria millorar moltíssim els minijocs, els quals no tenen tutorials i molts d'ells no tenen res a veure amb la mascota.

Figura 5: Logotip de Moy 7

Figura 6: Diferents espais de Moy 7

Dogotchi

Dogotchi no és dels jocs amb més descàrregues ni més potents del sector, però dins de la tipologia de jocs de mascota virtual destaca en un tipus de jocs que són els anomenats "retro", aquells que són imitacions de jocs d'èpoques anteriors o simplement utilitzen estètiques antigues i poc modernes. En aquest cas, *Dogotchi* és un dels joc que intenten ser una rèplica del mític *Tamagochi*, però portant-lo a l'era digital i adaptant-lo a dispositius mòbils. El que més destaca d'aquest joc és evidentment l'estètica i la coherència que manté, jugant molt amb píxel-art i utilitzant una paleta de colors monocromàtics. Tot i això, és un joc molt avorrit a comparació amb altres com el que hem analitzat anteriorment, ja que té pocs minijocs i mecàniques molt simples. També es tracta d'un joc molt poc intuïtiu per infants a causa de l'estil, i si reunim tots aquests aspectes negatius o que es poden millorar molt podem entendre que a nivell de descàrregues estigui molt per sota dels altres i que obtingui diverses crítiques dolentes. (Veure Figures 7 i 8).

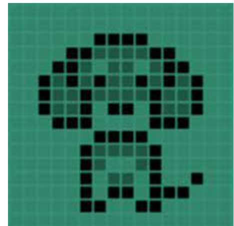


Figura 7: Logotip de Dogotchi

Figura 8: Diferents espais de Dogotchi

2.2.2. Model de negoci

A continuació es definirà el model de negoci del videojoc, així que primerament s'exposaran les diferents opcions que existeixen al mercat actual:

- **De pagament (*Pay-to-Play*):** Aquest model es basa en la idea de fer pagar al jugador per tot el contingut del joc sense cap mena de limitació. És el model tradicional on els jugadors compren una còpia física o digital i el poden usar sense restriccions de contingut.
- **De subscripció:** La mateixa idea que el model de pagament, amb la diferència que el pagament es realitza per un cert període de temps. Quan aquesta subscripció finalitza, el jugador pot escollir seguir pagant o no. En cas de no fer-ho, es limita l'accés al joc.
- **Gratuïts (*Free-to-Play*):** El model consisteix en oferir gratuïtament el contingut del videojoc. El rendiment econòmic prové de la possibilitat de compra de diferents elements del joc o d'anuncis.
- ***Freemium*:** La idea és oferir de manera gratuïta l'ús del videojoc però no la totalitat del contingut. Així doncs, per accedir a aquesta totalitat, el jugador ha de pagar.

Analitzant aquests 4 models i la tipologia de models que utilitzen els jocs del mercat, l'opció que més s'adapta a la meva proposta és el model *Free-to-Play.* En aquest cas, i el que fan la gran majoria de jocs de mascotes virtuals, els jugadors obtenen el joc de forma gratuïta però tenen la possibilitat de pagar per obtenir diferents accessoris o pujar de nivell de forma més ràpida. Aquest model funciona molt bé en aquests tipus de videojocs a causa de que no es limita al jugador en cap moment a res, però d'altra banda s'ofereix la possibilitat d'accedir a més contingut ràpidament a canvi de petits pagaments, els quals molts usuaris també veuen com una bona manera de donar suport als desenvolupadors.

2.3. Públic objectiu

Tal i com s'ha anat afirmant durant la memòria, el públic objectiu d'aquest videojoc són infants, més concretament nens i nenes d'entre 5 i 12 anys, tot i que molts adolescents també tenen jocs de mascota virtual als seus dispositius però, o no solen ser freqüents , o els hi acaben durant molt temps instal·lats als dispositius mòbils.

Sabent que existeix una franja d'edat relativament curta i molt fixada, s'haurà de tenir en compte a l'hora de desenvolupar el disseny del videojoc, com el fet de saber quins i com han de ser els elements que hauran de prendre protagonisme, com es creen les interfícies, fer que el videojoc sigui el màxim intuïtiu possible, etc. Per una banda pot semblar que el fet de tenir aquesta franja tan limitada faci que el videojoc mai pugui arribar a ser un èxit en el mercat actual ni que obtingui un nombre de descàrregues elevat, però d'altra banda tenir molt clar quin és el públic objectiu i no estar treballant pensant en tants panorames diferents pot facilitar la feina de disseny i, en conseqüència, poder fer que aquest públic objectiu estigui al màxim de còmode amb el projecte que es vol arribar a desenvolupar.

3. Planificació

3.1. Metodologia de treball

Des d'inicis de setembre del 2021 es va iniciar la planificació del projecte, un cop decidit el projecte i tenint consciència del temps que es disposa i els recursos, tan tècnics com sobretot humans amb els que es treballarà. Per això es va realitzar una divisió de tasques en 3 grans grups: disseny, desenvolupament i proves. A continuació es comentaran cada un d'aquests grups i el temps que es va dedicar a cada un d'ells:

- **Disseny (Setembre 2021 – Novembre 2021):** Abans d'endinsar-nos en el desenvolupament del projecte, s'havia de deixar clar com seria aquest en tots els aspectes. En aquesta fase entra en joc l'estudi de mercat, l'anàlisi de punts forts i febles de jocs similars al meu, decisions sobre mecàniques i estil, disseny d'espais i minijocs, etc. Aquí es decideix quina forma tindria el joc i es comença a desenvolupar en pseudo-codi algunes de les mecàniques principals.

- **Desenvolupament (Novembre 2021 – Abril 2022):** Les tasques de desenvolupament es poden dividir en dos grups:
  - **Programació**: Es desenvolupen totes les mecàniques del joc de manera òptima, ja que prèviament s'ha deixat clar com havien de ser aquestes pensant en tot el contingut que ha de tenir el joc. També es desenvolupa la gestió d'espais, els diversos minijocs que hi haurà, sons, interfícies i menús.
  - **Art:** Un cop les mecàniques del joc es van desenvolupant i queden definides, es dona vida al joc a través de l'art. En aquest grup entra tota la creació dels recursos 2D que apareixeran al videojoc, seguint sempre una mateixa estètica per no perdre la coherència en el projecte.

- **Proves (Abril 2022 – Maig 2022):** Un cop es tingui una primera versió del joc és hora de posar-lo a prova i trobar tots els errors que aquest pugui tenir per solucionar-los, igual que aspectes que visualment puguin fallar o siguin millorables, per acabar tenint una proposta que sigui sobretot funcional i coherent.

Per la organització d'aquest projecte s'ha utilitzat el servei que ofereix la pàgina web de *Trello.com*, (Veure Figura 9) dividint les tasques i assignant-les a diferents mesos, tot i que aquestes puguin variar segons el ritme que porti el procés i possibles inconvenients que puguin anar apareixent durant el desenvolupament del videojoc.

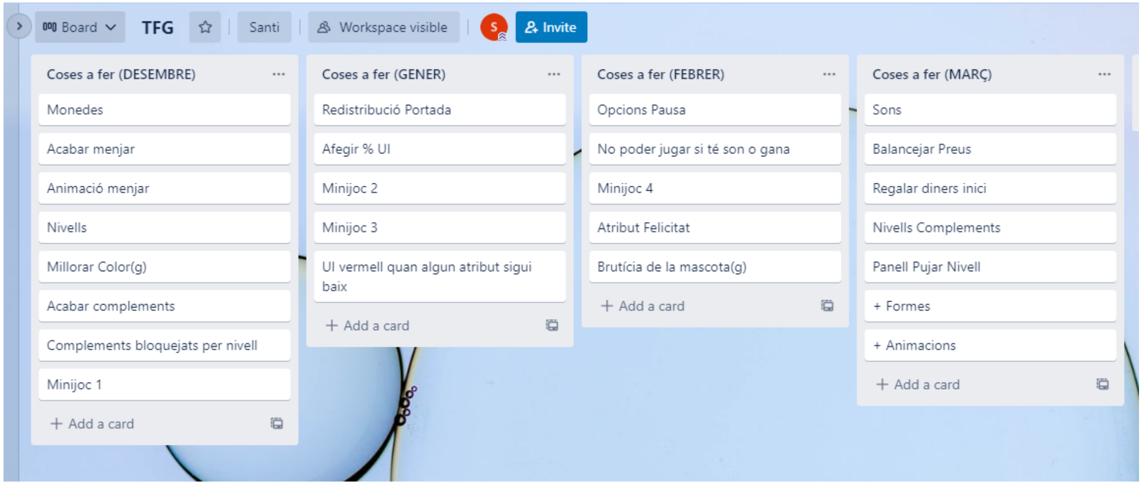


Figura 9: Captura de pantalla de *Trello* per organitzar el projecte

3.2. Eines de treball i programari

En aquest apartat s'analitzen els programes que s'han utilitzat pel desenvolupament del videojoc, comentant els motius de per què s'han utilitzat i les seves principals característiques.

3.2.1. Motor de joc

El motor de joc és un dels elements més importants a l'hora de desenvolupar un videojoc, i l'elecció d'aquest agafa encara més importància quan es vol crear un joc des de zero. En l'actualitat existeixen diversos motor de jocs potents, però n'hi ha dos que destaquen per sobre de la resta: *Unity* i *Unreal Engine*. Ambdós són la base de molts dels videojocs que hi ha al mercat actual, i només els fan servir grans empreses de la indústria per desenvolupar productes, sinó que també solen ser les primeres opcions per desenvolupadors independents a causa de l'automatització que aquests ofereixen i per la gran comunitat que tenen, encarregada de resoldre dubtes, publicar exemples, oferir documentació, etc.

Tot i ser dos grans productes, tenen algunes diferencies que poden fer decantar a l'usuari a triar-ne un o l'altre. Per una banda, *Unreal Engine* destaca en el desenvolupament de jocs en 3D, ja que ofereix un ampli ventall d'opcions a artistes i dissenyadors de jocs d'aquest tipus. D'altra banda, *Unity* consta d'eines per desenvolupar jocs 3D i 2D però destaca en jocs en 2D per la interfície, ja que és tot molt intuïtiu i facilita molt la feina al desenvolupador. De fet, aquest és el gran motiu pel qual s'ha escollit *Unity* com a motor de joc del projecte (Veure Figura 10). A més, *Unity* compta amb la comunitat de desenvolupadors més gran que hi ha en l'actualitat, fent que la resolució de dubtes sigui molt ràpida i eficient, obtenint diverses maneres per resoldre un problema i molts exemples i tutorials.

Figura 10: Logotip de *Unity*

3.2.2. Editor de codi

Per escriure el codi del joc s'ha utilitzat un únic editor, *Visual Studio 2019* (Veure Figura 11). Es tracta d'un editor de codi molt potent i és el predeterminat de *Unity*. Tanmateix no només s'ha escollit per aquests dos aspectes, sinó que la característica principal per la qual *Visual Studio 2019* ha estat l'únic editor que s'ha fet servir durant el projecte és el fet de poder fer proves de codi en temps real, és a dir, en el moment en que es fa un canvi al codi i aquest es desa, canvia automàticament el projecte fet a *Unity*. D'aquesta manera s'evita el fet d'haver d'anar reiniciant el motor de joc cada vegada que es vol comprovar una modificació.

Figura 11: Logotip de *Visual Studio*

3.2.3. Programari artístic

Sent el projecte un videojoc en 2D, s'ha escollit com a programa artístic principal *Inkscape* (Veure Figura 12). Es tracta d'un editor de gràfics vectorial on es poden crear i editar diagrames, línies, gràfics, logotips i tot tipus d'il·lustracions. És una eina de dibuix molt potent i còmode, a part de ser fàcil de fer servir i d'entendre. En aquest projecte s'ha utilitzat per crear tots els objectes visuals com complements, formes de mascotes, interfícies, etc., seguint sempre una mateixa tipologia per no perdre la coherència en el projecte.

Figura 12: Logotip d'*Inkscape*

També s'ha utilitzat *Adobe Photoshop* (Veure Figura 13) per retocar alguns aspectes del videojoc, com la portada i els fons dels minijocs i dels diferents espais. També s'ha fet servir per crear el logotip del videojoc.

Figura 13: Logotip d'*Adobe Photoshop*

3.2.4. Altres

Per escriure la memòria del projecte s'ha utilitzat *Microsoft Word* (Veure Figura 14), un dels programes de processament i tractament de textos més coneguts del mercat.

Figura 14: Logotip de *Microsoft Word*

4. Marc de treball i conceptes previs

Abans d'aprofundir en la part més tècnica i àmplia del projecte, és necessari familiaritzar-se amb alguns conceptes per tal d'entendre millor el desenvolupament. També s'especificaran algunes característiques generals dels videojocs que tenen un públic objectiu similar al que s'està desenvolupant, per comprendre el motiu de l'aparició contínua al projecte d'alguns elements.

4.1. Característiques principals dels videojocs per infants

Des d'un bon principi es volia que aquest projecte fos destinat a nens i nenes, més específicament es volia fer per infants hospitalitzats. Després d'estudiar la situació i estar en contacte amb l'hospital Sant Joan de Déu de Barcelona, un dels centres matern-infantils d'Europa que atén més pacients, es va arribar a la conclusió que hi ha tants infants hospitalitzats per tantes raons diverses, que és gairebé impossible desenvolupar un videojoc o aplicació que puguin fer servir tots, ja que s'han de tenir en compte masses factors tan psicològics com físics per acabar tenint un producte que el puguin utilitzar tots sense cap mena de restricció. Tot i això, fer un videojoc per infants seguia sent viable i hi havia motivacions personals, les esmentades a l'Apartat 1.3, que ajudaven a tenir la voluntat de fer-ho.

D'aquesta manera, abans d'iniciar la part més tècnica, cal que es tingui en compte durant la lectura d'aquesta memòria el fet de que s'està desenvolupant un videojoc per infants, és a dir, hi haurà un seguit d'elements els quals serien diferent si es tractés d'un videojoc pensat per a que el públic objectiu fos més gran. Per exemple, en jocs com aquest és molt important la claredat de les interfícies, ja que es vol que el nen o nena sàpiga en tot moment on està, per què serveixen els elements que està visualitzant i que no es senti confós al veure alguna cosa que no acabi d'entendre el motiu pel qual l'està veient. Això es veurà reflectit en l'abundància de textos fent referència a objectes, funcions o espais, per deixar clar en tot moment el que es vol transmetre a l'usuari amb qualsevol objecte.

També són molt importants els sons, per molt repetitius que puguin arribar a semblar en algun moment, ja que és una altra manera d'identificar objectes que ja s'han vist prèviament i dels quals l'infant ja en té algun tipus de coneixement, facilitant l'enteniment de les funcionalitats.

4.2. Conceptes previs del motor de joc

4.2.1. Gestió d'elements i objectes a Unity

A continuació s'explicaran conceptes del funcionament bàsic de *Unity*, d'aquesta manera es podrà entendre millor el desenvolupament del projecte llegint la memòria.

Primerament cal comentar el flux de treball dels **assets**. Veure Figura 18. Un asset és una representació de qualsevol ítem que pot ser utilitzat al projecte. Un asset pot venir d'un arxiu creat fora de *Unity*, com un dibuix o una imatge en 2D, però també poden ser crear dins del mateix motor de joc, com un *Sprite Renderer*, un controlador d'animacions o un text simple.

Figura 18: Captura d'alguns *assets* del projecte

Pel que fa a les finestres principals n'hi ha 6 (Veure Figura 19):

- **Finestra del projecte**: on es pot accedir i gestionar els assets que pertanyen al projecte.
- **Finestra d'escena:** per seleccionar i posicionar entorns, jugadors, càmera, enemics i qualsevol tipus d'objecte.
- **Finestra de jerarquia:** conté cada un dels objectes de la escena. Alguns d'aquests són instàncies directes d'arxius d'assets i altres són instàncies de *Prefabs*, que són objectes personalitzats i creats amb una sèries de característiques, pensats per reutilitzar-los i afegir-los al projecte d'una manera ràpida i senzilla.
- **Finestra d'Inspector:** l'Inspector s'utilitza per veure i editar propietats dels objectes, igual per les preferències i altres ajustaments dins de *Unity*.
- **Barra d'eines:** consta de 4 controls bàsics, cada un d'ells relacionats amb diferents parts de l'editor: eines de transformacions bàsiques, botons que afecten en la visualització de la finestra d'escena, botons d'inici, pausa i pas del joc i desplegables que controlen els objectes que es mostren a la finestra d'escena.
- **Finestra del joc:** renderitzada per les càmeres del joc, és la representació del projecte finalitzat.

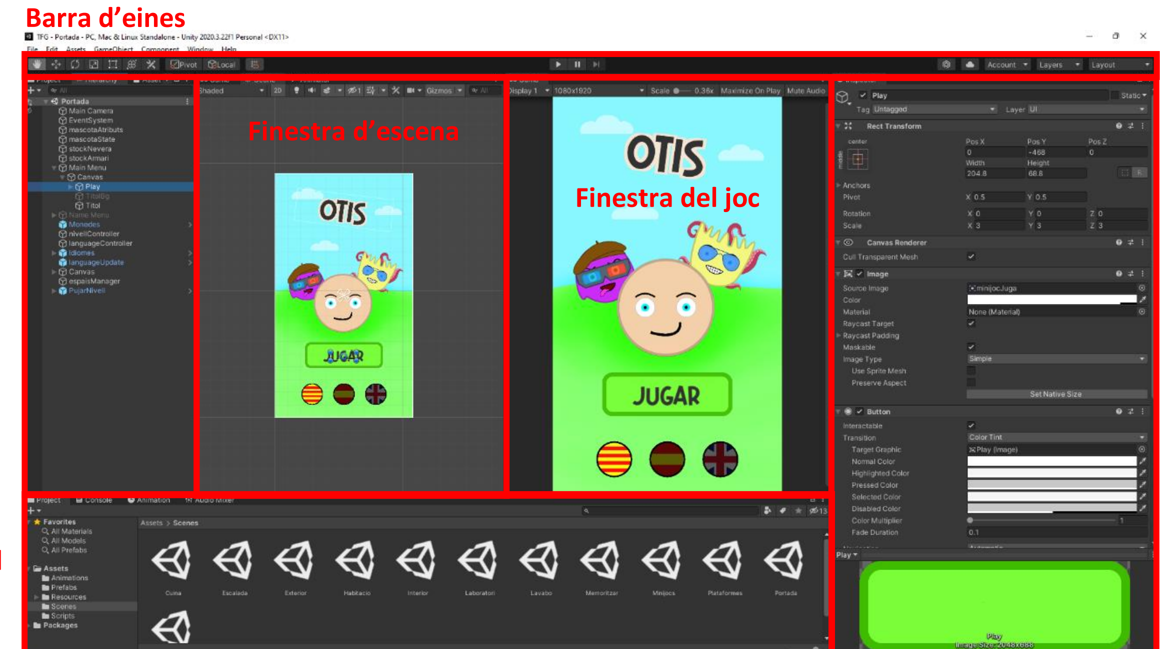


Figura 19: Finestres principals de *Unity*

4.2.2. Gràfics i físiques 2D

Els objectes gràfics en 2D a *Unity* són coneguts com a **sprites**. Els sprites són només textures estàndard, però hi ha diverses tècniques per combinar i manipular aquestes textures per rendiment i conveniència durant el desenvolupament. Unity proporciona un editor integrat que permet extreure gràfiques sprites des d'una imatge més gran. Això permet editar un nombre de components d'imatge dins d'una sola textura l'editor d'imatge. Els sprites són renderitzats amb un components anomenat *Sprite Renderer*, el qual es pot afegir a un objecte mitjançant el menús de components, o directament es pot crear un objecte amb un *Sprite Renderer* que ja estigui adjuntat.

Pel que fa a les físiques, Unity compta amb un motor de física separat per a la manipulació de la física en 2D per tal de fer ús d'optimitzacions disponibles únicament en 2D. Els components corresponen als components de física estàndard 3D com ara *Rigidbody* o *Box Collider*, però amb "2D" agregat al nom. Aleshores, els sprites poden estar equipats amb *Rigidbody 2D* i *Box Collider 2D*. La majoria dels components de física en 2D són simplement versions "aplanades" de l'equivalent en 3D.

Els *colliders* defineixen una forma aproximada d'un objecte que és utilitzat pel motor de física per determinar col·lisions amb altres objectes. Es poden fer servir diferents tipus de *colliders* segons la forma que es vol donar a cada objecte, ja siguin qudrats, cercles o formes més complexes. Tanmateix, afegir un *Rigidbody 2D* permet a un sprite ser mogut d'una manera físicament convincent en aplicar forces des d'un *script*. Quan el component *collider* apropiat està adjuntat a l'objecte sprite, aquest serà afectat per col·lisions amb altres objectes en moviment. Utilitzant física, simplifica moltes mecàniques de joc i permet un comportament real i emergent amb un mínim de codi.

4.2.3. Scripting

La programació de ***scripts*** és un ingredient essencial en tots els jocs. Fins i tot el joc més simple necessitarà *scripts* per respondre a entrades del jugador i assegurar que els esdeveniments del joc s'executin en el moment adequat. A més, els *scripts* poden ser usats per crear efectes gràfics, controlar el comportament físic d'objectes o fins i tot implementar un sistema d'intel·ligència artificial per als personatges del joc.

Abans d'endinsar-nos amb el tema dels *scripts*, hi ha un altre concepte que cal tenir clar, i és el dels ***GameObjects***. Els *GameObjects* són objectes fonamentals a *Unity* que representen qualsevol element que pugui aparèixer al videojoc com personatges o l'escenari. Aquests no aconsegueixen res per si mateixos, però funcionen com a contenidors per a components, que implementen la veritable funcionalitat. Per exemple, un simple cub sòlid (Veure Figura 20) és un *GameObject* que té un component *Mesh Filter* i un *Mesh Renderer* per dibuixar la superfície del cub i un altre component *Box Collider* per representar el volum sòlid de l'objecte en termes de física com els que s'han vist a l'Apartat 4.2.2.

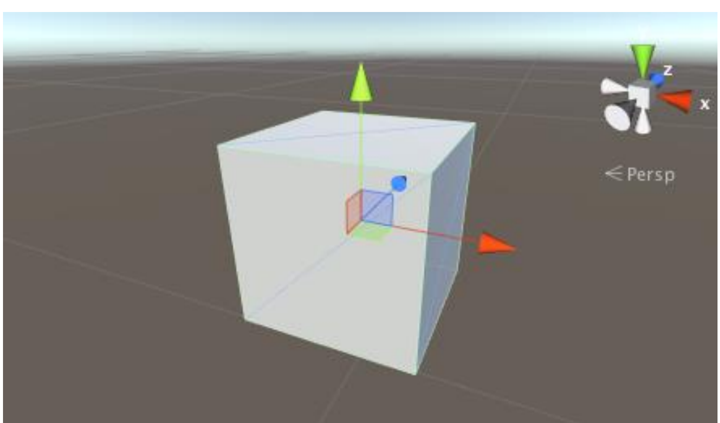


Figura 20: Cub sòlid a l'entorn de *Unity*

D'aquesta manera, i seguint amb el tema de l'*scripting*, el comportament dels *GameObjects* és controlat pels Components que estan adjunts. Però, encara que els components integrats de *Unity* siguin molt versàtils, s'ha d'anar més enllà del que poden proporcionar per implementar les característiques pròpies d'un videojoc, i aquí és on entren en joc els *scripts*. *Unity* permet crear un component nou utilitzant scripts. Aquests permeten *trigger* (activar/desactivar) esdeveniments del joc, modificar propietats del component en el temps i respondre a l'input de l'usuari de la manera que es desitgi.

4.2.4. Llibreries utilitzades

Sobre les llibreries utilitzades a *Unity*, és necessari comentar que no s'ha fet servir cap tipus de llibreria externa a aquest motor de joc, sinó que s'ha pogut realitzar tot el desenvolupament del projecte amb les llibreries internes del propi motor. Les dues que més s'han utilitzat, a part de les més comunes, són les següents: *UnityEngine.SceneManagement*, per la gestió d'escenes a través de codi, i *UnityEngine.UI* per modificar i obtenir informació de qualsevol tipus d'element de la interfície del joc, com botons, textos o imatges.

De la primera llibreria esmentada s'han fet servir les següents funcions:

- **GetActiveScene():** Per obtenir informació de l'escena activa actualment.
- **LoadScene():** Per carregar una escena a partir del nom.

Mentre que de la segona llibreria s'han utilitzat funcions per controlar els següents tipus d'elements: Botons, *Canvas*, Imatges, *Sliders*, *Sprites* i Textos.

4.2.5. Altres terminologies utilitzades

Finalment, és necessari també que s'expliquin una sèrie de conceptes més tècnics sobre el desenvolupament del projecte, i d'aquesta manera assegurar que qualsevol persona que llegeixi aquesta memòria pugui entendre el que s'està fent. Durant la memòria es farà referència diverses vegades als ***Prefabs***. Un *Prefab* és un objecte reutilitzable, i creat amb una sèrie de característiques dins del projecte. En aquest cas, a *Unity* és tan fàcil com crear un objecte i arrossegar-lo a una carpeta per convertir-lo en *Prefab*, el qual es pot instanciar al videojoc cada vegada que sigui necessari (Veure Figures 21 i 22).

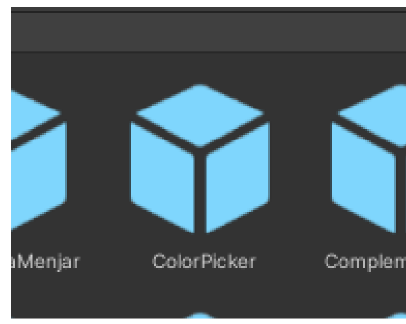


Figura 21: *Prefab ColorPicker* a la finestra de projecte

Figura 22: *Prefab ColorPicker* a la finestra de jerarquia

Un altre concepte que es farà servir freqüentment és el de **punt de *spawn*** o ***spawner***. Es tracta d'un punt de generació d'algun element, que ja pot ser el personatge principal o qualsevol tipus d'objecte. Aquests *spawners* es programen via codi per fer que generin més o menys elements, amb una freqüència més alta o més baixa, en una direcció o en una altra, etc.

5. Disseny del videojoc

En aquesta secció s'explica el disseny d'*Otis*, nom que se li va acabar posant al videojoc, justificant les decisions que es van prendre per cada aspecte del disseny. S'inclouen el disseny de la mascota, de les diferents mecàniques, dels espais, dels minijocs, etc.

5.1. Disseny de la mascota

Després de fer l'estudi de mercat, es va arribar a la conclusió que el videojoc havia de destacar en alguna mecànica en concret, i es va escollir com a mecànica principal la modificació i personalització de la mascota. D'aquesta manera, es vol que la mascota pugui ser de moltes maneres diferents i que l'usuari tingui la sensació que la seva és única, així que el disseny d'aquesta mascota és essencial en el procés de desenvolupament de la mecànica. Quan es va iniciar el disseny es va fer un primer esbós (Veure Figura 23) on es van definir les parts que tindria la mascota, amb els atributs de cada part que podrien arribar a ser modificables per l'usuari.

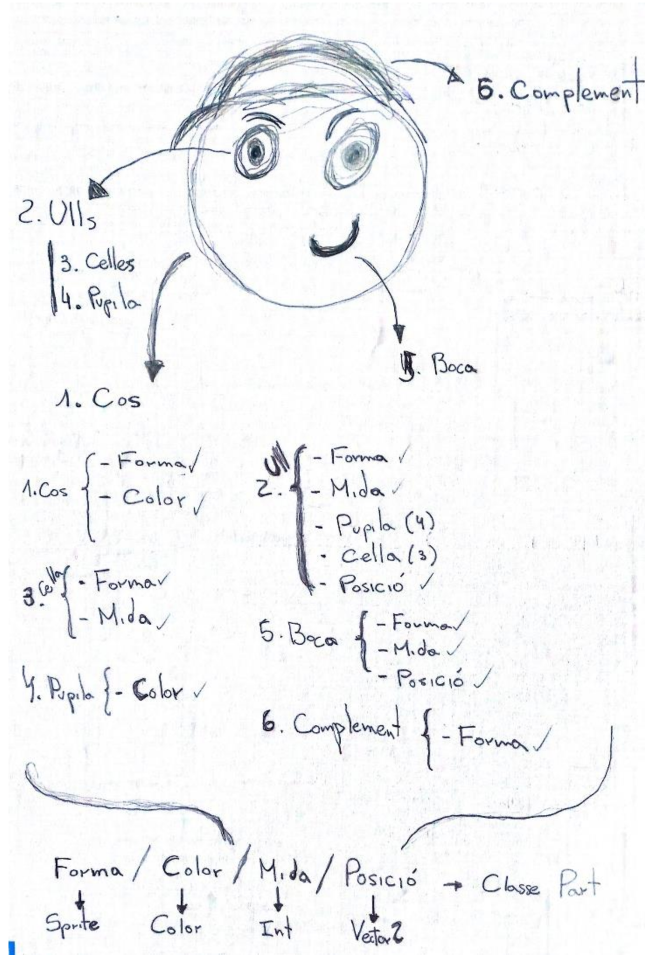


Figura 23: Primer esbós de la mascota

En aquest primer esbós es poden identificar totes les parts de la mascota que es podran modificar amb els atributs corresponents. A continuació s'enumeraran les parts i s'explicaran els diferents atributs.

Parts:

1. **Cos**: Es podrà modificar la forma i el color
2. **Ulls**: Es podrà modificar la forma, la mida, la posició, la cella i la pupil·la
3. **Celles:** Es podrà modificar la forma i la mida
4. **Pupil·la:** Es podrà modificar el color
5. **Boca:** Es podrà modificar la forma, la mida i la posició
6. **Complements:** La mascota en tindrà tres, un a la part superior, un als ulls i un a la part inferior, i de cada un d'ells es podrà modificar la forma

Atributs:

- **Forma:** És la imatge que es mostra de cada part, i a *Unity* es representarà com un *Sprite*.
- **Color:** És ni més ni menys que el color de la part, que també es pot tractar com a color amb els *scripts*.
- **Mida:** Es representarà amb un valor numèric que farà que la part es mostri més gran o més petita.
- **Posició:** Es representarà amb una variable de tipus *Vector2*, que consta de dues variable (*x* i *y*) que seran la posició horitzontal i vertical de la part.

Combinant aquestes parts amb els diferents atributs que es poden modificar, les mascotes seran molt versàtils i cada una serà única per l'usuari, que és l'objectiu principal del projecte. Després de l'esbós vist a la Figura 23 es va fer una sèrie de proves per comprovar aquesta versatilitat que es busca (Veure Figura 24).

Figura 24: Proves de mascotes variant els atributs de les parts

Això és pel que fa al disseny gràfic de la mascota i les diferents parts per a què totes siguin úniques, però també cal comentar el tema del benestar de la mascota i les variables que definiran aquest benestar. L'objectiu del videojoc és que el jugador es faci càrrec d'una mascota virtual, cuidant-la i jugant amb ella, i el que marcarà com de bé se la està cuidant són els següents atributs:

- **Gana**: El jugador haurà de controlar constantment que la seva mascota no passi gana. Aquesta gana sempre augmenta, lentament, durant el joc i es pot disminuir donant-li de menjar a la mascota.
- **Energia**: L'energia disminueix constantment durant el joc, i disminueix de manera més ràpida si la mascota està jugant. Per augmentar aquesta energia s'ha de posar a dormir a la mascota.
- **Felicitat:** Aquest atribut representa com d'avorrida o feliç està la mascota. Si no està jugant aquest atribut disminueix i augmenta quan el jugador està jugant amb qualsevol minijoc.
- **Brutícia:** La brutícia de la mascota augmenta sempre durant el joc, i per netejar-la cal que el jugador la dutxi al lavabo.

5.2. Disseny d'espais

La distribució d'espais en aquests tipus de videojocs va en funció de les mecàniques que tenen, però tot i això hi ha un patró que s'utilitza a tots els jocs i partir d'aquí es poden adaptar aquests espais a cada videojoc. A continuació es mostrarà un esquema de la distribució d'aquests espais a *Otis* (Veure Figura 25) i s'explicarà què es podrà fer a cada un d'aquests espais.

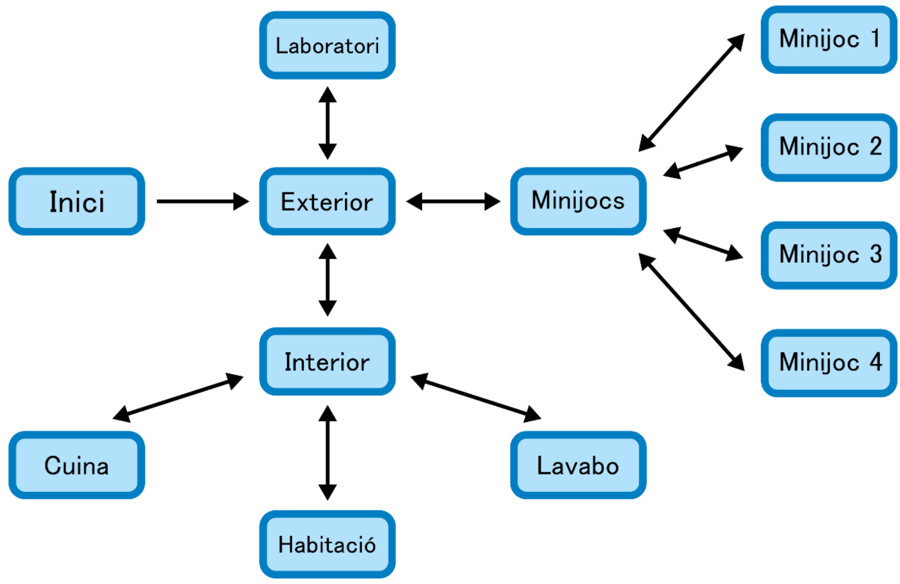


Figura 25: Esquema de la distribució d'espais

Exterior: L'espai de l'exterior és el primer que es troba l'usuari, i té la funció principal de connectar 3 espais diferents, com són l'interior de la casa de la mascota, el laboratori i l'espai de minijocs.

Interior: L'interior de la casa té com a objectiu connectar els 3 diferents espais que hi ha a la casa de l'usuari, que són la cuina, l'habitació i el lavabo.

Laboratori: El laboratori és l'espai on el jugador podrà modificar les parts de la mascota. Aquest procés el podrà fer tantes vegades com vulgui, sense cap mena de limitació per nivell i podent ajustar lliurament els atributs de cada part al seu gust.

Cuina: A la cuina es podrà comprar els diferents tipus de menjar que hi ha per a la mascota i alimentar-la.

Habitació: A l'habitació es podran comprar els complements per a la mascota, vestir-la amb els diferents complements, i posar-la a dormir per recuperar energia.

Lavabo: Al lavabo l'únic que es podrà fer és dutxar a la mascota per disminuir la brutícia d'aquesta.

Minijocs: Aquest espai és on el jugador podrà seleccionar qualsevol dels minijocs que hi ha disponibles per jugar amb la mascota i aconseguir monedes.

5.3. Disseny de minijocs

Els minijocs a *Otis* són molt importants per 3 raons: és la millor manera d'obtenir monedes, una forma d'obtenir punts per pujar de nivell i l'única via per augmentar l'atribut de felicitat de la mascota.

La idea pel disseny dels 4 minijocs que s'implementaran en aquest projecte és la de fer mecàniques molt simples pels infants, que es tractin de jocs els quals no siguin difícils d'entendre. Aquests minijocs també seran infinits, és a dir, s'anirà augmentant la dificultat fins arribar al punt on el jugador falli i s'acabi el joc, d'aquesta manera no s'hauran de dissenyar els diferents nivells ni s'hauran de fer coses complexes tenint en compte que aquests minijocs són una part del projecte, i no la base d'aquest. A continuació s'explicarà la idea de cada un dels minijocs que posteriorment s'implementaran (Veure Figura 26).

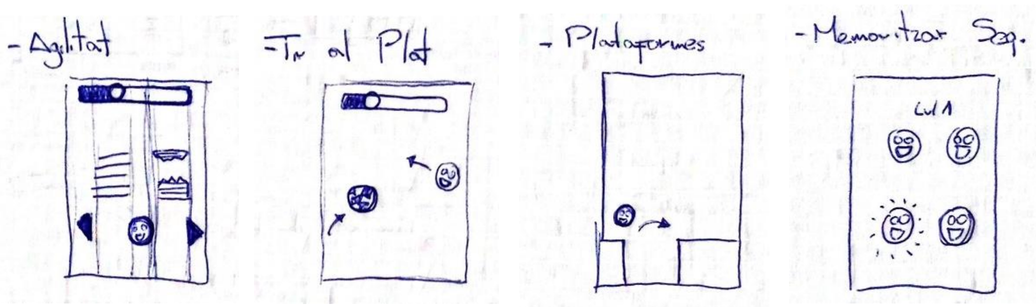


Figura 26: Primers esbossos dels minijocs

5.3.1. Tir al Plat

El Tir al Plat és un minijoc on al jugador li sortiran mascotes per una de les dues bandes de la pantalla (esquerra o dreta) i les haurà de prémer. Hi haurà un temps restant que anirà disminuint i quan aquest estigui a zero, s'haurà acabat el minijoc. Cada vegada que el jugador premi una mascota, aquest temps augmentarà, i cada vegada que una mascota desaparegui de la pantalla i el jugador no la hagi premut, el temps disminuirà encara més. D'aquesta manera es penalitza el fet de saltar-se una mascota i no estar a temps de prémer-la. Tan la quantitat de mascotes que vagin sortint com la velocitat d'aquestes anirà augmentant a mida que vagi passant el temps (Veure Figura 27).

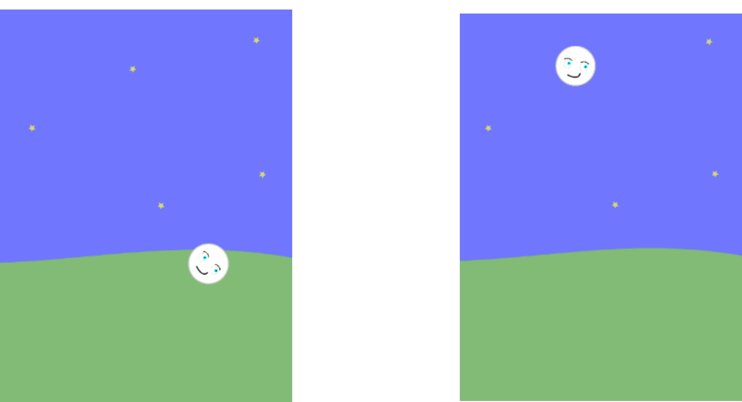


Figura 27: Minijoc Tir al Plat

5.3.2. Plataformes

El segon minijoc és el típic joc de plataformes 2D, on el jugador haurà de prémer la pantalla perquè la mascota salti d'una plataforma a una altra i no caigui. Si la mascota cau, el minijoc s'haurà acabat. Si el jugador manté la pantalla premuda quan la mascota fa un salt, aquesta es mantindrà més estona a l'aire. Les plataformes aniran apareixent cada vegada de manera més ràpida, i hi haurà diferents longituds de plataformes per augmentar la dificultat del minijoc i per fer que el temps de reacció del jugador hagi de ser més baix (Veure Figura 28).

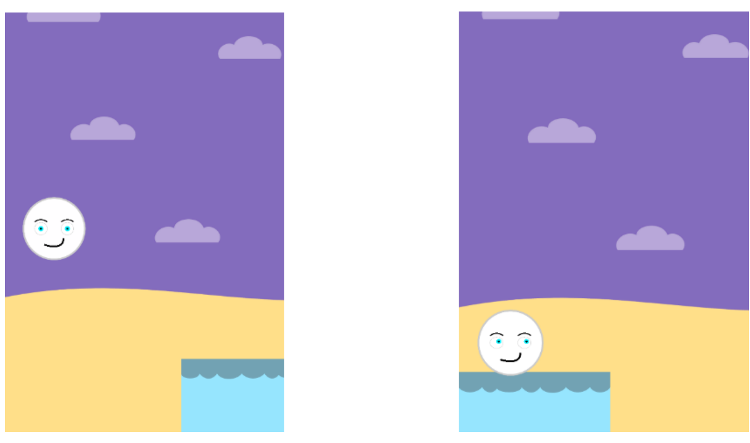


Figura 28: Minijoc Plataformes

5.3.3. Escalada

Al minijoc de l'escalada, la mascota del jugador haurà d'anar escalant per dos arbres els quals un tindrà una fusta trencada i l'altre estarà en bon estat. D'aquesta manera, per escalar, el jugador haurà de prémer l'arbre que té la fusta en bon estat en cada cas. Aquestes fustes s'aniran generant a cada arbre de manera aleatòria. Hi haurà un temps restant que anirà disminuint constantment, però aquest augmentarà quan el jugador premi l'arbre amb la fusta en bon estat. El minijoc finalitzarà si aquest temps s'acaba o si la mascota escala per un arbre que té la fusta trencada (Veure Figura 29).

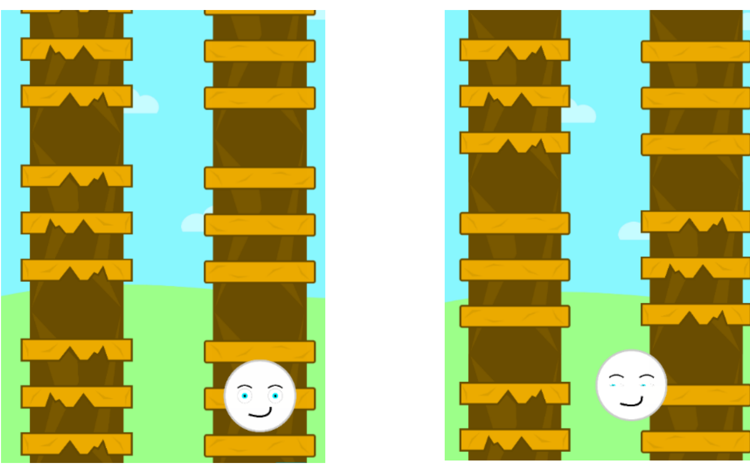


Figura 29: Minijoc Escalada

5.3.4. Memoritzar

Finalment l'últim minijoc és de memoritzar seqüències. Al jugador se li mostrarà una seqüència representada per diferents mascotes i després de memoritzar-la, la haurà de repetir. El nombre de mascotes que es mostrarà anirà variant, igual que la longitud de les seqüències, que anirà augmentant per afegir cada vegada més dificultat al minijoc. El minijoc finalitzarà en el moment en què el jugador s'equivoqui repetint una seqüència (Veure Figura 30).

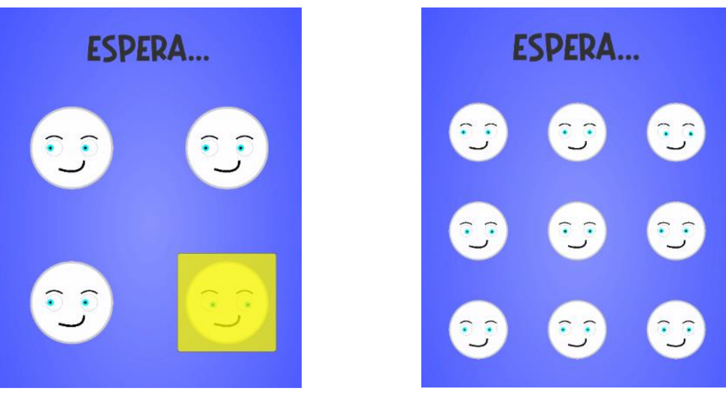


Figura 30: Minijoc Memoritzar

5.4. Economia del joc

L’economia d’*Otis* es basa en tres aspectes del videojoc, les monedes que té el jugador, l’estat de la mascota i el nivell. D’aquesta manera, l’objectiu del joc és mantenir el benestar de la mascota, el qual s’aconsegueix amb una sèrie de mecanismes diferents, i d’aquests n’hi ha molts que és necessari tenir monedes per fer-los. Per exemple, per poder augmentar la felicitat de la mascota, només s’ha de jugar a qualsevol minijoc i no cal gastar monedes, però per disminuir la gana se li ha de donar de menjar, i aquest menjar només es pot comprar amb monedes. D’altra banda, el nivell que tindrà cada jugador li servirà per optar a més o menys complements, és a dir, cada complement tindrà com a requisits per comprar-lo un nombre de monedes i un nivell mínim.

Així doncs, per optar al benestar de la mascota ja s’ha comentat a l’apartat 5.1. quins atributs havia de mantenir el jugador en bon estat i com ho havia de fer.

Ara s’ha de comentar com el jugador pot obtenir les monedes i com pot pujar de nivell. Per aconseguir monedes hi ha diferents vies o maneres:

- Jugant a qualsevol dels 4 minijocs disponibles
- Girant una roda la primera vegada que el jugador entri cada dia al joc, donant-li un número de monedes en funció del que marca aleatòriament la ruleta (Veure Figura 31)
- Pujant de nivell

Per pujar de nivell, el jugador va aconseguint punts realitzant les diferents accions:

- Jugant a qualsevol dels 4 minijocs disponibles
- Donant-li de menjar a la mascota
- Dutxant a la mascota
- Posant a dormir a la mascota
- Comprant complements

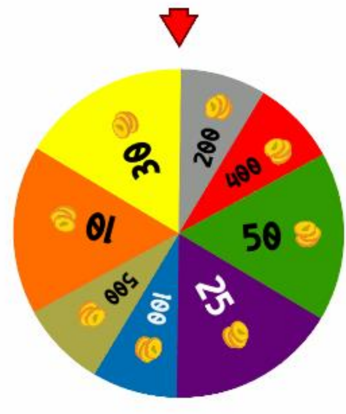


Figura 31: Roda que podrà girar el jugador una vegada al dia

5.5. Mecàniques

Fins ara s'ha comentat com serà la mascota, la distribució dels espais, els diferents minijocs i com funciona l'economia. Així doncs, a continuació s'explicarà tot el que es podrà fer al videojoc, és a dir, les diferents mecàniques i on es podran realitzar aquestes (Veure Taula 4).

Mecànica	Espai/s
Modificar la mascota: - Canviar la forma del cos - Canviar el color del cos - Canviar la forma dels ulls - Moure amunt, avall, esquerra i dreta els ulls - Separar i ajuntar els ulls - Fer els ulls més grans i més petits - Canviar el color dels ulls - Canviar la forma de les celles - Moure amunt, avall, esquerra i dreta les celles - Separar i ajuntar les celles - Fer les celles més grans i més petites - Canviar la forma de la boca - Moure amunt, avall, esquerra i dreta la boca - Fer la boca més gran i més petita	Laboratori
Comprar menjar	Cuina
Alimentar la mascota	Cuina
Comprar complements	Habitació
Vestir la mascota	Habitació
Posar a descansar a la mascota	Habitació
Dutxar a la mascota	Lavabo
Saltar plataformes	Minijoc Plataformes
Repetir seqüència	Minijoc Memoritzar
Prémer plat	Minijoc Tir al Plat
Escalar un arbre	Minijoc Escalada
Girar roda regal diari	Exterior
Canviar d'idioma	*Tots els espais*
Pujar o baixar el volum del videojoc	*Tots els espais*
Pausar el videojoc	*Tots els espais*
Posar nom a la mascota	Inici

Taula 4: Mecàniques d'*Otis*

5.6. Interfícies

Les interfícies d'un videojoc tenen molta importància, i encara més quan el públic objectiu són nens i nenes, ja que aquestes interfícies han de ser molt clares i no provocar cap mena de confusió a l'infant. Aquestes interfícies han de donar tota la informació necessària al jugador en tot moment, així que a continuació s'explicarà els diferents tipus d'interfícies que hi haurà al videojoc i què es mostraran als diferents espais.

5.6.1. HUD

Als videojocs s'anomena HUD (*Head-Up Display*) a la informació que es mostra per pantalla en tot moment, generalment en forma d'icones i números. Aquest HUD ha ser dissenyat per no molestar al jugador i donar-li la informació necessària. En aquest cas, la informació que s'ha decidit donar és: el nombre de monedes que té el jugador, el nivell, el nom de la mascota, l'estat de la mascota i un botó per sortir de l'espai en el que es troba (Veure Figura 32).

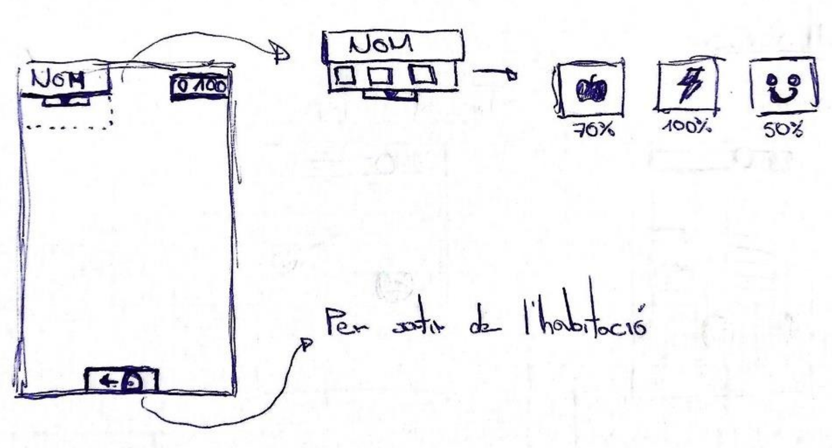


Figura 32: Primer esbós del HUD

5.6.2. Interfícies específiques per a cada espai

El HUD vist a l'apartat anterior es mostrarà a tots els espais, excepte als minijocs perquè pot molestar al jugador, però a cada espai a més s'han de mostrar algunes funcionalitats específiques, les quals es comentaran a continuació:

- **Exterior:** Botons per anar al laboratori, a l'interior de la casa i a l'espai dels minijocs
- **Interior:** Botons per anar a la cuina, al lavabo i a l'habitació
- **Cuina:** Botons per comprar menjar i per accedir al menjar que té el jugador per donar-li a la mascota
- **Habitació:** Botons per engegar i apagar els llums (per posar a descansar a la mascota), per comprar complements i per accedir als complements que té el jugador per vestir la mascota
- **Lavabo:** Botó per dutxar a la mascota
- **Minijocs:** Botons per accedir als 4 minijocs disponibles
- **Minijoc Escalada:** *Slider* que representa el temps restant, punts actuals i punts màxims fets pel jugador
- **Minijoc Tir al Plat:** *Slider* que representa el temps restant, punts actuals i punts màxims fets pel jugador
- **Minijoc Memoritzar:** Punts actuals i punts màxims fets pel jugador
- **Minijoc Plataformes:** Punts actuals i punts màxims fets pel jugador

5.7. Marca identificadora

5.7.1. Nom

Tots els videojocs de mascotes virtuals solen tenir com a nom del joc un nom propi, fent referència a com s'anomena la mascota, però que pot modificar el mateix jugador. Aquests noms propis que es posen als títols dels jocs solen ser noms curts i fàcils de pronunciar, pensant en què el públic objectiu són infants i posar noms complexos pot generar-los refús. És per això que a aquest projecte se li va decidir posar el nom d'*Otis*.

5.7.2. Tipografia

En tot el joc es farà servir la mateixa tipografia, no només per no perdre coherència, sinó també per ser una manera de generar una identitat pròpia al joc. A l'hora d'escollir una tipografia se'n buscava una que fos animada però simple a la vegada, de nou pensant en el públic objectiu del videojoc. Finalment es va escollir la font "Pamit" com a principal i única tipografia del videojoc (Veure Figura 33). També es va optar per utilitzar només lletres en majúscula, perquè són molt més clares i donen menys confusió.

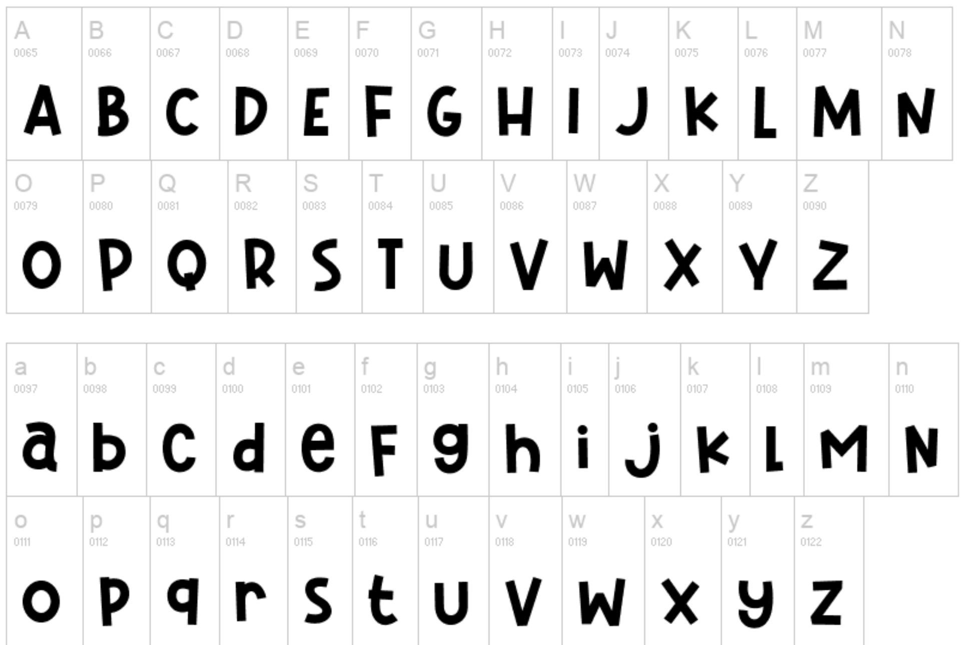


Figura 33: Mostra de la font "Pamit"

5.7.3. Logotip

Per fer el logotip del videojoc s'ha utilitzat, evidentment, la tipografia que s'ha comentat a l'apartat anterior i 3 combinacions de la mascota, fent-les encaixar amb els colors que es desitjava pel logotip i mostrant les diferents possibilitats que tenen aquestes mascotes. (Veure Figures 34 i 35)

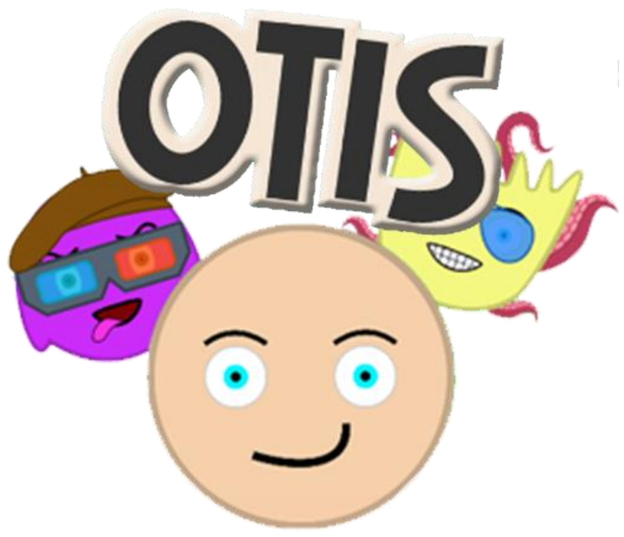


Figura 34: Logotip d'*Otis*

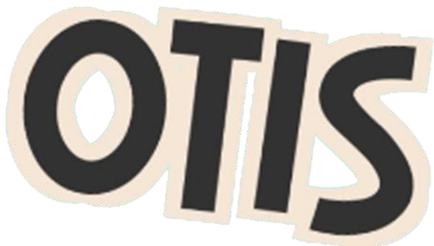


Figura 35: Logotip d'*Otis* sense les mascotes

5.8. Estil artístic

Per aquest projecte s'ha escollit un estil artístic molt bàsic, tots els elements del videojoc tenen colors llisos i una vora del mateix color, però una mica més fosca. Aquesta vora li aporta al joc un estil més infantil, ja que la gran majoria de dibuixos animats per a nens i nenes utilitzen aquestes vores, encara que en molts casos són sempre de color negre. El que s'ha buscat en tot moment amb aquest estil és no perdre la coherència del joc, i que absolutament tots els elements segueixin el mateix patró, ja siguin objectes, espais o, fins i tot, les interfícies. També s'ha escollit un estil molt simple per la mancança d'artistes al projecte, ja que soc l'única persona que desenvolupa aquest videojoc i la part artística no és el meu punt fort.

D'altra banda, s'han fet objectes amb formes molt simples, perquè fer-los complexos amb aquest estil hagués estat un error greu, ja que es perdria tota la harmonia que es busca en un videojoc per infants. A continuació es pot veure a la Figura 36 un seguit d'objectes amb aquest estil que s'ha comentat.

Figura 36: Diversos objectes d'*Otis*

6. Implementació

La part d'implementació és fonamental en un videojoc, ja que se li ha de donar vida a tot el que s'ha dissenyat a l'apartat anterior. D'aquesta manera, a continuació s'explicarà com han estat implementats tots els aspectes d'*Otis* a nivell tècnic i de forma específica.

6.1. Mascota

La mascota virtual és l'ànima del joc, l'únic i principal personatge i a qui l'usuari li dona tota l'atenció necessària. És per això que essencialment ha de ser la part més ben implementada del joc, ja que els errors que hi puguin haver en qualsevol aspecte de la mascota es notaran fàcilment per l'usuari. En aquest sub-apartat s'explicarà l'estructura de la mascota i com estan implementades totes les parts.

6.1.1. Estructura

La estructura de la mascota està representada per 3 capes, tal i com es pot veure a la Figura 37 i s'explica a continuació:

1. Els **atributs** de la mascota. Aquest atributs fan referència a cada part de la mascota: el cos, la boca, els ulls, les celles i els complements. Així doncs, aquí es guarden tots els aspectes d'aquestes parts: la posició, la forma, el color, la mida, etc.

2. L'**estat** de la mascota. En aquesta capa s'emmagatzemen les dades en relació al benestar de la mascota, això inclou la gana, la brutícia, l'energia i la felicitat.

3. La capa **visual**. Aquesta capa representa tot allò que veu l'usuari, és la representació en base a les dades que hi ha a les dues capes anteriors. Segons com siguin els atributs de la mascota, el jugador la veurà d'una manera o d'una altra. D'altra banda, l'estat també pot fer variar l'aspecte de la mascota, ja sigui perquè té molta brutícia o perquè no té energia.

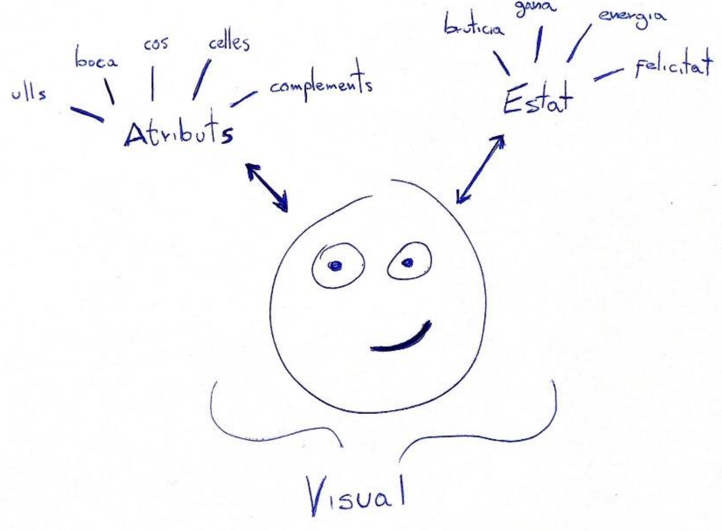


Figura 37: Primer esbós de l'estructura de la mascota

6.1.1.1. Atributs

La implementació dels atributs està a l'*script* anomenat *mascotaAtributs.cs*. En aquest *script* s'ha declarat una instància en un *Singleton*, fent que aquest objecte es declari una vegada a l'inici del joc i no s'instanciï més, ja que els atributs de la mascota són únics. Dins d'aquest *script* també s'ha creat una estructura "atribut" (Veure Figura 38) per representar cada part de la mascota. Aquesta estructura consta d'una varible *string* que fa referència al nom, una variable *Sprite* que representa la forma d'aquesta part, una variable de tipus *Color* que és el color de la part, una variable *float* que representa la mida de la part i una variable de tipus *Vector3* que fa referència a la posició de la part. Cada objecte atribut també té 6 variable de tipus *float* que representen els màxims i mínims de posició i mida, és a dir, la mida màxima i mínima que pot tenir la part i el màxim que pot estar situada cap a l'esquerra, a la dreta, amunt i avall. Aquesta estructura també té funcions per canviar totes aquestes variables als objectes i per inicialitzar-les.

```
public struct atribut
{
    public string nom;
    public Sprite forma;
    public Color partColor;
    public float mida;
    public Vector3 posicio;

    float MAX_MIDA, MIN_MIDA, MAX_ESQ, MAX_DRE, MAX_AMUNT, MAX_AVALL;
```

Figura 38: Variables de l'estructura "atribut"

6.1.1.2. Estat

L'estat actual de la mascota està implementat a l'*script* anomenat *mascotaState.cs*. Aquí hi ha 4 variables de tipus *float* que representen la brutícia, la felicitat, la gana i l'energia de la mascota, igual que un seguit de constants que controlen l'augment o decrement d'aquestes variables, el temps que tarden en augmentar o disminuir i el valor inicial que se li dona a cada variable (Veure Figura 39). També hi ha 3 variable privades booleanes que controlen si la mascota està jugant, dutxant-se o dormint, les quals es fan servir per saber si els valors de les variables que s'han esmentat prèviament han d'augmentar o disminuir, igual que una constant que s'utilitza per marcar en quin moment és urgent que el jugador s'interessi per l'estat de la mascota. Es tracta d'un número que si és superior al valor de l'energia, la felicitat o la gana, se li marcarà al jugador d'alguna manera al HUD.

L'*script mascotaState.cs* també té funcions encarregades d'augmentar o disminuir les variables, és a dir, per controlar tot l'estat de la mascota en funció del comportament i del que estigui fent el jugador. A més, aquest *script* té un *Singleton* implementat com els atributs, ja que l'estat de la mascota també és únic.

```
private float gana, felicitat, energia, bruticia;

private const float INITIAL_GANA = 85;
private const float INITIAL_FELICITAT = 95;
private const float INITIAL_ENERGIA = 70;
private const float INITIAL_BRUTICIA = 0;

private const float AUGMENT_BRUTICIA = 1;
private const float TEMPS_AUGMENT_BRUTICIA = 3;

private const float DECREMENT_GANA = 1;
private const float TEMPS_DECREMENT_GANA = 2;

private const float DECREMENT_FELICITAT = 1;
private const float TEMPS_DECREMENT_FELICITAT = 2;

private const float DECREMENT_ENERGIA = 1;
private const float TEMPS_DECREMENT_ENERGIA = 1;

private bool dormint, dutxant, jugant;

private const float ATRIBUT_URGENT = 15;
```

Figura 39: Variables i constants de l'*script mascotaState.cs*

6.1.1.3. Visual

La part visual de la mascota, és a dir l'aspecte que veu el jugador en tot moment, està implementada a l'*script* anomenat *mascotaVisual.cs*. Aquest *script* es crida cada vegada que s'inicia una escena on hi apareix la mascota, ja que és l'encarregat de consultar l'estat i els atributs de la mascota per mostrar-la d'una forma o d'una altra. Consta d'un *array* que fa referència a totes les parts de la mascota i d'un seguit de variables per controlar les taques que li apareixen a la mascota segons la seva brutícia (Veure Figura 40). Aquesta brutícia es mostra a través d'una funció *Update* que bàsicament modifica la transparència de les taques segons com de bruta està la mascota.

```
public class mascotaVisual : MonoBehaviour
{
    public partController[] parts;

    public GameObject taques;
    public int nTaques;

    private float[] transpTaques;
    private float minBruticia = 30;
```

Figura 40: Variables l'*script mascotaVisual.cs*

Com es pot veure a la figura anterior, les parts emmagatzemades a l'*array* són de tipus *partController*, una classe implementada a l'*script partController.cs*. Aquí és on gestionen totes les modificacions visuals de cada part, és a dir, on es canvia la posició del *Transform* per moure les part, on es canvia el valor d'escala per modificar la mida, on es canvia l'*Sprite* del component *Sprite Renderer* per modificar la imatge, etc.

6.1.2. Animacions

Les animacions de la mascota també són un aspecte molt important, perquè són una de les millors maneres de donar-li vida, no només a la mateixa mascota, sinó també al videojoc en general. És per això que en aquest projecte s'ha optat per animar la mascota en 3 aspectes diferents: el parpelleig i moviment dels ulls, simular la respiració quan la mascota està descansant i el moviment de la boca quan menja. A continuació s'explicarà com s'han implementat aquestes 3 animacions.

Primerament, per fer el parpelleig dels ulls, s'han creat dos *GameObjects* que encapsulen cada ull, i se'ls hi ha afegit un component *Animation* al qual se li ha associat una animació que canvia l'escala de l'ull, fent-lo petit i tornant-lo a fer gran per dona l'efecte de parpelleig (Veure Figures 41 i 42).

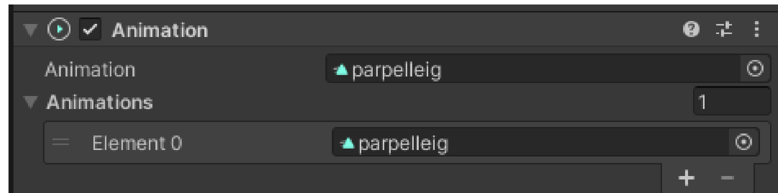


Figura 41: Component *Animation* per mostrar el parpelleig dels ulls

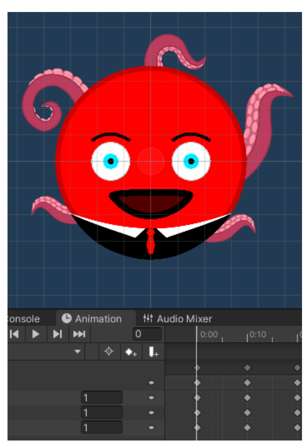


Figura 42: Diversos *frames* de l'animació de parpelleig

Per fer el moviment dels ulls es va crear un *script* anomenat *eyesAnimation.cs*, i es va afegir un component *Animation* als ulls. D'aquesta manera, l'*script* s'encarrega d'agafar les animacions associades a aquest component i les mostra de manera aleatoria. Es va triar fer-ho d'aquesta manera perquè, a nivell de desenvolupament, és la manera més senzilla, ja que un cop aquest *script* ja està fet, només s'han de crear les animacions i associar-les al component. Llavors el propi *script* s'encarrega de gestionar-les i mostrar-les, sense haver de posar enlloc quantes animacions hi ha ni haver de modificar cap altre paràmetre (Veure Figura 43).

```
1 referencia
IEnumerator setAnimation()
{
    while (true)
    {
        int idAnim = (int)Random.Range(0, nAnims);
        anim.Play(anims[idAnim]);

        if (ull != null)
        {
            ull.GetComponent<Animation>().Play(anims[idAnim]);
        }

        yield return new WaitForSeconds(Random.Range(2.5f, 3));
    }
}
```

Figura 43: Funció de l'*script eyesAnimation.cs* que mostra les animacions aleatòriament

Per fer la respiració de la mascota quan està descansant s'ha creat una animació que modifica l'escala de la mascota, donant un efecte de que agafa aire i el deixa anar (Veure Figura 44).

Figura 44: Animació de respiració de la mascota

Finalment, per fer l'animació de la mascota menjant es va fer una funció en un *script* anomenat *foodImage*.cs, el qual controla la imatge del menjar que se li dona a la mascota. Aquest *script* s'explicarà amb més detall en un dels apartats següents, on es comentaran tots els elements que hi ha als diferents espais del joc. Pel que fa a la funció que gestiona l'animació (Veure Figura 45), primerament canvia la forma de la boca de la mascota per una boca oberta, i després modifica l'escala d'aquesta boca per donar l'efecte de que la mascota està mastegant (Veure Figura 46).

```
IEnumerator animacioMenjar()
{
    boca.canviForma(bocaMenjar);
    float midaBoca = boca.gameObject.transform.localScale.y;
    float tempsAnim = 0.05f;

    boca.gameObject.transform.localScale = new Vector3(boca.gameObject.transform.localScale.x, midaBoca / 1.5f, boca.gameObject.transform.localScale.z);
    yield return new WaitForSeconds(tempsAnim);
    boca.gameObject.transform.localScale = new Vector3(boca.gameObject.transform.localScale.x, midaBoca / 3, boca.gameObject.transform.localScale.z);
    yield return new WaitForSeconds(tempsAnim);
    boca.gameObject.transform.localScale = new Vector3(boca.gameObject.transform.localScale.x, midaBoca / 8, boca.gameObject.transform.localScale.z);
    yield return new WaitForSeconds(tempsAnim);
    boca.gameObject.transform.localScale = new Vector3(boca.gameObject.transform.localScale.x, midaBoca / 3, boca.gameObject.transform.localScale.z);
    yield return new WaitForSeconds(tempsAnim);
    boca.gameObject.transform.localScale = new Vector3(boca.gameObject.transform.localScale.x, midaBoca / 1.5f, boca.gameObject.transform.localScale.z);
    yield return new WaitForSeconds(tempsAnim);
    boca.gameObject.transform.localScale = new Vector3(boca.gameObject.transform.localScale.x, midaBoca / 3, boca.gameObject.transform.localScale.z);
    yield return new WaitForSeconds(tempsAnim);
    boca.gameObject.transform.localScale = new Vector3(boca.gameObject.transform.localScale.x, midaBoca / 8, boca.gameObject.transform.localScale.z);
    yield return new WaitForSeconds(tempsAnim);

    boca.gameObject.transform.localScale = new Vector3(boca.gameObject.transform.localScale.x, midaBoca, boca.gameObject.transform.localScale.z);
    boca.canviForma(bocaMascota);
}
```

Figura 45: Funció per animar la boca de la mascota quan està menjant

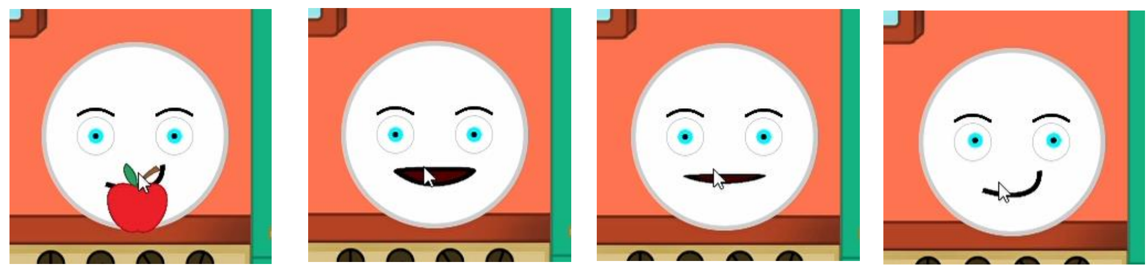


Figura 46: Animació de la mascota quan està menjant

6.1.3. Modificacions gràfiques puntuals

En algunes situacions del joc s'ha decidit canviar una mica l'aspecte de la mascota per donar-li més vida, és a dir, amb el mateix objectiu que tenen les animacions. A l'apartat anterior s'ha analitzat una d'aquestes situacions, quan la mascota està menjant, on a part de l'animació canvia l'aspecte de la boca de la mascota. A part d'aquesta situació, es fan modificacions puntuals quan la mascota té poca energia i quan està descansant.

Primerament, en el moment en què l'energia de la mascota és baixa, se li formen unes ulleres als ulls de la mascota (Veure Figura 47). Per fer això, es canvia la forma de l'ull pel mateix però amb aquestes ulleres. Aquesta funcionalitat està implementada a l'*script partController.cs*, que gestiona cada part de la mascota, i varia la forma de l'ull segons l'energia que la mascota té (Veure Figura 48).

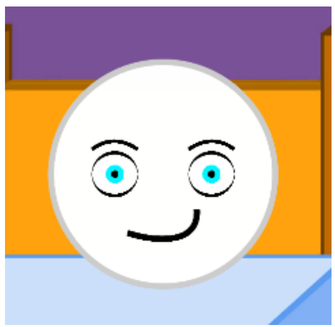
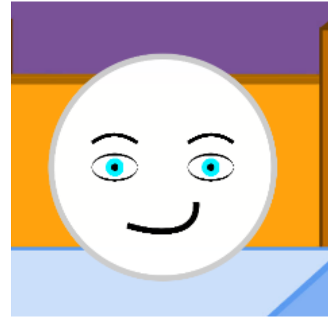



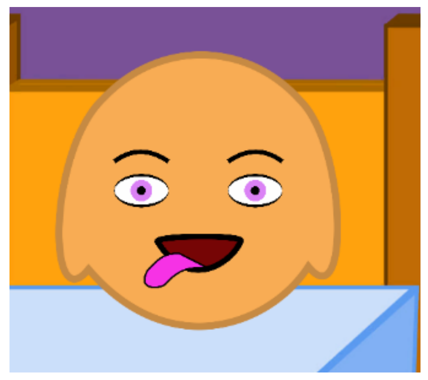
Figura 47: Ulleres que se li formen a la mascota quan té poca energia

```
Mensaje de Unity | 0 referencias
void FixedUpdate()
{
    if ((nom == "ullE" || nom == "ullD"))
    {
        if (!dormint)
        {
            mascotaAtributs.instance.canviForma(this, getFormaUllSon(mascotaState.instance.getEnergia() < 25));
        }

    }
}
```

Figura 48: Part de l'*script partController.cs* que canvia la forma de l'ull segons l'energia de la mascota

D’altra banda, quan la mascota està descansant, no només es fa l’efecte de respiració vist a l’apartat anterior, sinó que també es canvia la forma de la boca i dels ulls (Veure Figura 49). Aquest canvi es fa des d’un *script* anomenat *roomState.cs* (Veure Figura 50), el qual controla tots els elements de l’habitació, d’aquesta manera sap quan la mascota està descansant i quan no. Aquest *script* s’analitzarà de manera més profunda en apartats posteriors, on s’analitzaran els elements que hi ha a cada espai.



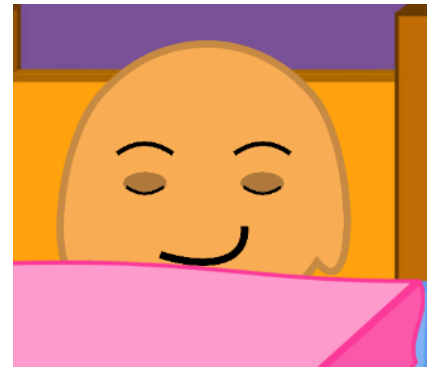


Figura 49: Canvis en ulls i boca de la mascota quan dorm

```
ullE.canviForma(ullTancat);
ullE.dormir();
ullE.transform.parent.GetComponent<Animation>().clip = null;
ullE.transform.parent.GetComponent<Animation>().Stop();
ullE.transform.GetChild(0).gameObject.SetActive(false);

ullD.canviForma(ullTancat);
ullD.dormir();
ullD.transform.parent.GetComponent<Animation>().clip = null;
ullD.transform.parent.GetComponent<Animation>().Stop();
ullD.transform.GetChild(0).gameObject.SetActive(false);

boca.canviForma(bocaTancada);
```

Figura 50: Part de l’*script roomState.cs* on es modifiquen ulls i boca de la mascota quan dorm

6.1.4. Prefab Final

Un cop definida l'estructura de la mascota i després d'implementar les parts corresponents, interessa convertir-la en *Prefab* per poder fer-la servir en qualsevol dels espai del videojoc, podent modificar el comportament, mida, posició, etc. sense haver de modificar gran part del codi i, sobretot, d'una manera ràpida i senzilla.

D'aquesta manera, el *Prefab* final de la mascota està compost per un seguit de parts: el cos, les celles, la boca i els quatre complements. També consta de dos *GameObjects* que encapsulen cada ull, i aquests ulls contenen els iris corresponents, els quals contenen cada un una pupil·la. Finalment també consta d'un *GameObject* que conté les taques que es visualitzen quan la mascota està bruta (Veure Figura 51).

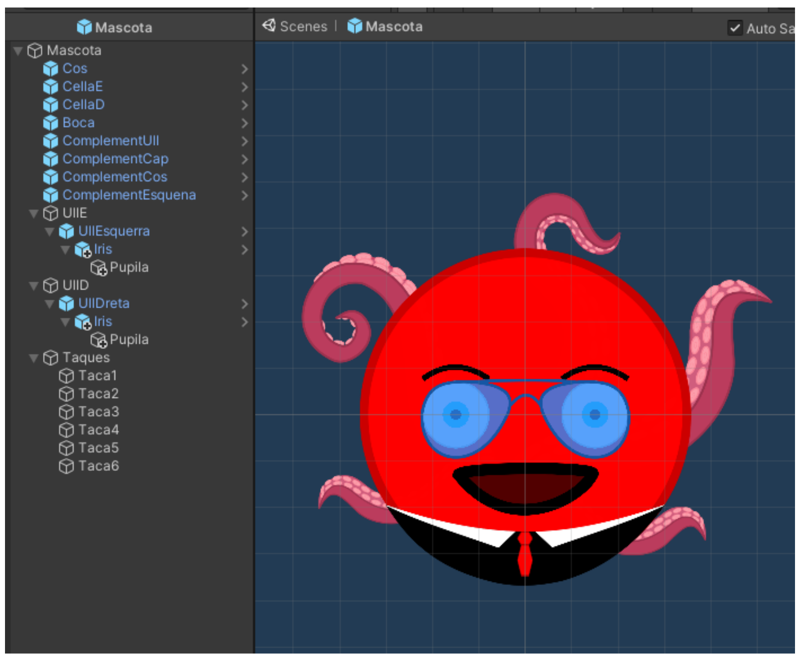


Figura 51: *Prefab* final de la mascota

La implementació d'aquest *Prefab* va ser tot un repte, perquè l'objectiu era que es pogués adaptar a qualsevol situació, ja sigui en un espai interior, en un minijoc o als espais de transició entre un espai i un altre, i per això mateix es va dedicar molt temps a acabar de desenvolupar-ho de manera correcta, però un cop tot el procés va acabar, ha estat un dels aspectes més importants pel desenvolupament del videojoc.

6.2. Espais

Després d'estudiar la implementació de la mascota, s'entrarà en el desenvolupament d'un altre element essencial del videojoc: els espais. Tal i com s'explica a l'apartat 5.2, els espais que hi ha a *Otis* són els següents: l'espai exterior, l'espai interior de la casa, el laboratori, la cuina, l'habitació, el lavabo, l'espai de minijocs i els 4 minijocs disponibles. En aquest apartat s'analitzarà com ha estat la implementació de cada un d'aquests espais i tots els elements que els conformen.

6.2.1. Exterior

L'espai exterior és el primer que es troba el jugador quan prem el botó de jugar, el que permet accedir a tres espais importants com són la casa, el laboratori i els minijocs. Es tracta d'un espai on la mascota es mou verticalment per entrar en un dels altres espais, localitzats a l'esquerra(Veure Figures 53).

En aquest espai hi ha els objectes que es poden veure a continuació, a la Figura 52.

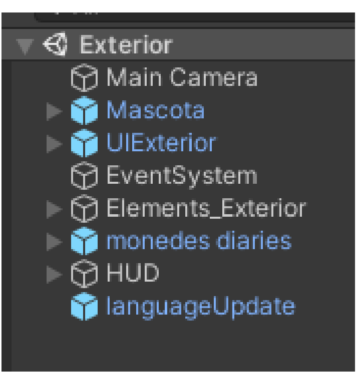
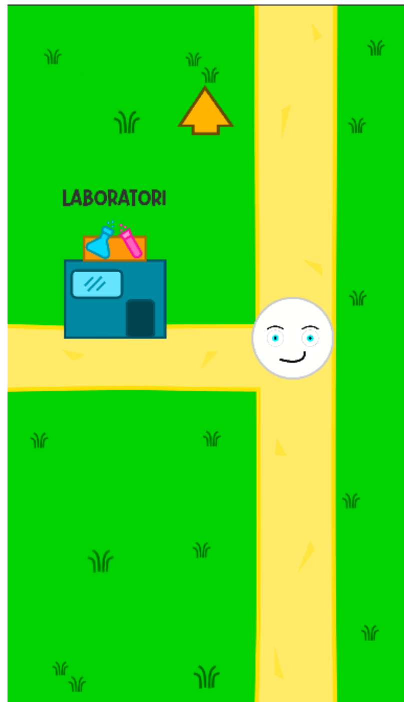
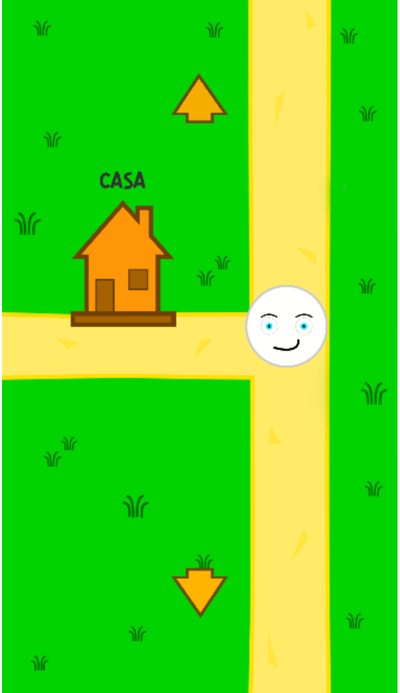


Figura 52: Objectes de l'exterior a la finestra de jerarquia

- **Mascota**: És el *Prefab* que s'ha comentat a l'apartat 6.1.4, on s'hi ha afegit un *script* anomenat *mascotaEspais.cs* que gestiona els moviments de la mascota en espais verticals.
- **UIExterior**: *Prefab* que mostra les fletxes que utilitza el jugador per moure's per l'espai verticalment.
- **Elements Exterior**: *GameObject* que conté els elements que apareixen en aquest espai: la imatge de fons i els botons per accedir als altres espais.
- **Monedes diàries:** *Prefab* que gestiona la roda per regalar monedes, la qual apareix un cop cada dia que el jugador entra el joc.
- **HUD**: Informació que es mostra per pantalla al jugador, comentada a l'apartat 5.6.1.
- **LanguageUpdate**: Gestor de textos per canviar d'idioma (s'aprofundirà als següents apartats).





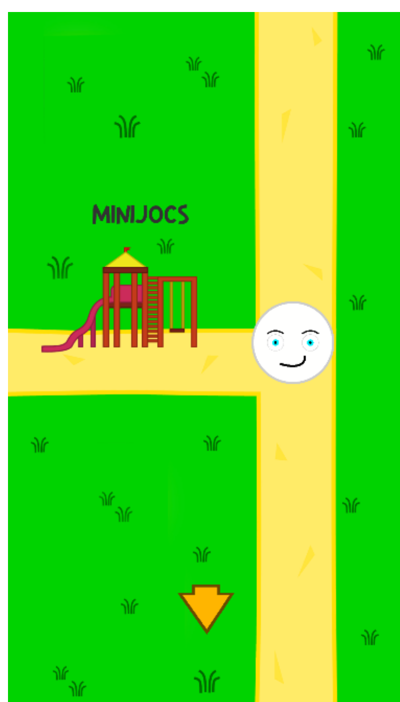


Figura 53: Espai exterior

6.2.2. Interior

A l'espai interior de la casa s'hi accedeix per la icona de la casa que hi ha a l'exterior (com es pot veure a la Figura 53), i aquest permet al jugador accedir a 3 espais: el lavabo, la cuina i l'habitació. L'estructura és similar a la de l'espai exterior (Veure Figures 54 i 55), l'únic que canvia, a part de l'estètica, és la distribució dels botons per accedir als espais. Aquests botons a l'espai interior es troben al mig, per donar la sensació que la mascota està entrant i sortint dels 3 diferents espais.

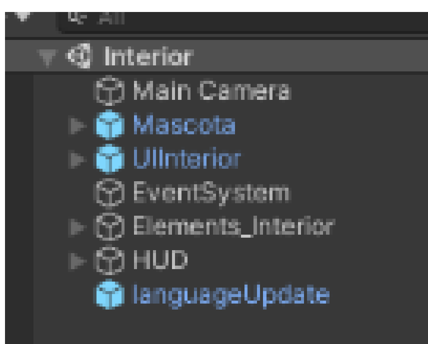


Figura 54: Objectes de l'interior de la casa a la finestra de jerarquia

- **Mascota**: És el *Prefab* que s'ha comentat a l'apartat 6.1.4, on s'hi ha afegit un *script* anomenat *mascotaEspais.cs* que gestiona els moviments de la mascota en espais verticals.
- **UIInterior**: *Prefab* que mostra les fletxes que utilitza el jugador per moure's per l'espai verticalment.
- **Elements Interior**: *GameObject* que conté els elements que apareixen en aquest espai: la imatge de fons i els botons per accedir als altres espais.
- **HUD**: Informació que es mostra per pantalla al jugador comentada a l'apartat 5.6.1.
- **LanguageUpdate**: Gestor de textos per canviar d'idioma (s'aprofundirà als següents apartats).

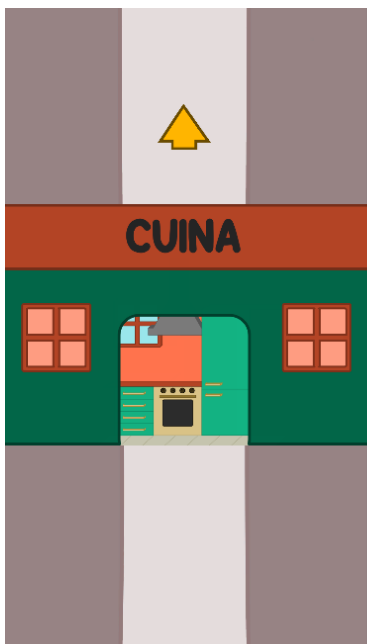


Figura 55: Espai interior de la casa

6.2.3. Laboratori

El laboratori és l'espai on més s'hi ha treballat a nivell d'implementació, perquè, tal i com s'ha dit en diferents apartats, la mecànica de modificar i personalitzar la mascota es volia que fos la principal i la que cridés més l'atenció del jugador. Així doncs, en aquest espai el jugador té totes les opcions de modificació de les diferents parts de la mascota (Veure Figura 56), i a continuació s'explicaran les diferents funcionalitats implementades i la distribució de les parts de la mascota que es poden modificar.

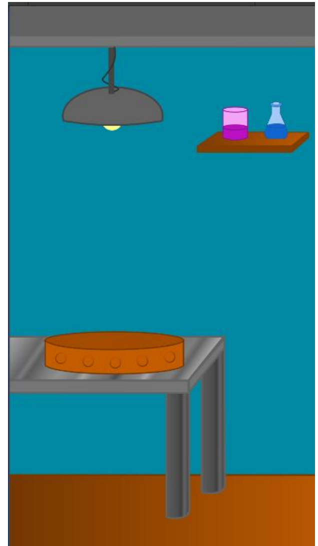


Figura 56: Espai del laboratori

6.2.3.1. Objectes de l'espai

En aquest espai hi ha els següents elements que es poden veure a la Figura 57, els quals es comentaran posteriorment.

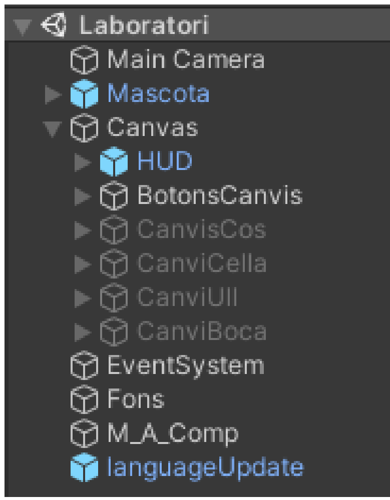


Figura 57: Objectes del laboratori a la finestra de jerarquia

- **Mascota**: És el *Prefab* que s'ha comentat a l'apartat 6.1.4, on no s'hi ha hagut d'afegir cap *script* addicional.
- **HUD**: Informació que es mostra per pantalla al jugador comentada a l'apartat 5.6.1.
- **Botons Canvis**: Panell que gestiona la distribució dels botons que permeten accedir a les funcionalitats de cada part que es pot modificar.
- **Canvis Cos:** Gestor de funcionalitats que permeten modificar el cos de la mascota.
- **Canvis Cella:** Gestor de funcionalitats que permeten modificar les celles de la mascota.
- **Canvis Ull:** Gestor de funcionalitats que permeten modificar els ulls de la mascota.
- **Canvis Boca:** Gestor de funcionalitats que permeten modificar la boca de la mascota.
- **Fons:** Imatge de decoració del laboratori.
- **M_A_Comp:** *GameObject* que gestiona el fet de mostrar o amagar els complements de la mascota quan el jugador la està modificant.
- **LanguageUpdate**: Gestor de textos per canviar d'idioma (s'aprofundirà als següents apartats).

6.2.3.2. Funcionalitats

Tal i com s'ha comentat a l'Apartat 6.1.1.1, els atributs de la mascota poden ser modificats en diferents aspectes: la forma, la mida, el color i la posició. Per implementar aquestes modificacions, s'ha creat un *script* anomenat *buttonClicked.cs*, que té una sèrie de paràmetres que el fan adaptable a qualsevol tipus de modificacions. L'objectiu d'aquest *script* és associar-lo a qualsevol botó que realitzi una modificació a la mascota, i que, segons els paràmetres que se li seleccionin a l'*script*, es faci una modificació o una altra. A continuació es comentarà l'estructura d'aquest *script*.

```
public bool color, mida, posicio, forma, vertical, complement;
public partController[] parts;
public float nMida;

private Button b;
private Image im;
Mensaje de Unity | 0 referencias
void Start()
{
    b = this.GetComponent<Button>();
    im = this.GetComponent<Image>();
    b.onClick.AddListener(TaskOnClick);
}
```

Figura 58: Variables i funció *Start()* de l'*script buttonClicked.cs*

Com es pot veure a la Figura 58, l'*script buttonClicked.cs* té una sèrie de variables booleanes públiques. Aquestes fan referència a quina modificació se li està fent a la part seleccionada. També té un *array* de parts, que fa referència a les parts que s'estan modificant, igual que un valor *float* que marca com s'ha de modificar aquesta part. Per exemple, si es volgués modificar la mida dels ulls, es marcaria la variable booleana *mida*, es posaria a l'*array* els dos ulls de la mascota, i es posaria com a valor de l'última variable un número que faria referència a com de grans o petits es volen fer els ulls.

En aquesta figura també es pot veure que es fa referència al botó que es prem per fer l'acció corresponent, ja que se li ha d'associar la funció encarregada de dur a terme les modificacions segons els valors de les variables, i a la imatge del botó, ja que aquesta pot contenir informació necessària per fer la modificació. Per exemple, si es vol canviar la forma del cos de la mascota, es prem un botó amb la forma desitjada. Així doncs aquest *script* ha de canviar la forma del cos actual per la imatge del botó, que és la nova forma que vol posar el jugador. Totes aquestes modificacions es fan cridant les funcions de l'*script mascotaAtributs.cs* passant-li com a paràmetres les parts que s'han de modificar i altres factors en funció del canvi que s'està fent. D'aquesta manera, l'*script buttonClicked.cs* es mira cada una de les parts de l'*array* quan el botó es prem, i segons el valor de les variables booleanes crida una funció o una altra a la instància que emmagatzema la informació dels atributs de la mascota. A continuació s'explicaran més específicament com es fan cada una d'aquestes modificacions.

6.2.3.2.1. Canvi de forma

Quan es vol canviar la forma de qualsevol part, es crea un botó que tindrà la imatge de les diferents formes que hi ha disponibles, i s'associa a aquest botó l'*script buttonClicked.cs* marcant a l'inspector la variable *Forma* i afegint a l'*array* de parts la part de la mascota que es vol modificar. El valor *N Mida* no cal posar-lo perquè en un canvi de forma només es canvia la imatge de la part que es vol modificar (Veure Figura 59).

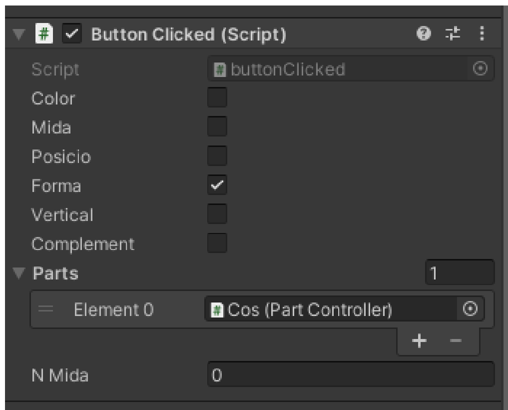


Figura 59: *Script buttonClicked.cs* associat al botó que canvia la forma del cos

Quan estan seleccionats aquests paràmetres a l'inspector, l'*script buttonClicked.cs* comprova que el nombre de parts que s'han de modificar sigui parell o imparell. Si és parell vol dir que hi ha dos elements que han de canviar, com podrien ser els ulls o les celles, en aquest cas no només s'ha de canviar la forma de la part, sinó que una de forma de les parts s'ha d'invertir per no perdre la simetria de la mascota. D'altra banda, si el nombre parts és imparell vol dir que se n'ha de modificar una i només canviant la imatge ja estaria la modificació feta (Veure Figures 60 i 61).

```
mascotaAtributs.instance.canviForma(parts[i], im.sprite);
if (i % 2 != 0)
{
    parts[i].invertir();
}
```

Figura 60: Part de *l'script buttonClicked.cs* que s'executa quan es canvia la forma d'una part

```
3 referencias
public void canviForma(partController part, Sprite novaForma)
{
    string nom = part.getNom();

    atribut a = new atribut();
    a = dAtributs[nom];
    a.canviForma(novaForma);
    dAtributs[nom] = a;

    setPart(part);
}
```

Figura 61: Funció de l'*script mascotaAtributs.cs* que canvia la forma d'una part

Un altre aspecte que cal comentar sobre la implementació del canvi de forma, és que s'ha creat un *script* anomenant *stockPart.cs* que gestiona totes les formes que hi ha d'una part. Es va implementar aquest *script* per facilitar la feina a l'hora d'afegir noves formes. La idea és que quan es crea una nova forma, només s'ha d'afegir a la carpeta corresponent del projecte, segons el nom de cada part. D'aquesta manera, és aquest *script* el que s'encarrega de gestionar el nombre de formes que hi ha per cada part i mostrar-les totes sense haver de tocar cap paràmetre de cap objecte (Veure Figura 62).

```
0 referencias
public void changeImage(bool endavant)
{
    if (endavant)
    {
        if (idPart >= nParts) idPart = 1;
        else idPart += 1;
    }
    else
    {
        if (idPart == 1) idPart = nParts;
        else idPart -= 1;
    }

    imatge.image.sprite = Resources.Load<Sprite>("Sprites/Mascota/" + nomPart + "/" + nomPart + idPart);
}
```

Figura 62: Funció de l'*script stockPart.cs* que accedeix a les carpetes del projecte corresponents per mostrar totes les formes d'una part

6.2.3.2.2. Canvi de mida

Si es vol canviar de mida d'una part, es crea el botó que s'ha de prémer per fer aquesta part més gran o més petita i se li associa l'*script buttonClicked.cs*. Seguidament es marca la variable booleana *Mida* i s'afegeix a l'*array* les parts que es volen modificar. També s'ha de posar a l'última variable el valor d'augment o disminució de la part, si es vol fer la part més gran ha de ser un valor positiu, en canvi aquest valor ha de ser negatiu si es vol fer la part més petita (Veure Figura 63).

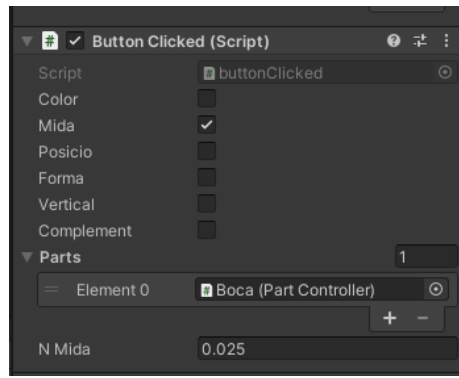


Figura 63: *Script buttonClicked.cs* associat al botó augmenta la mida de la boca

Amb aquests paràmetres seleccionats a l'inspector, l'*script buttonClicked.cs* crida una funció a la instància que emmagatzema els atributs de la mascota, passant-li com a paràmetres cada part que es vol modificar i el número d'augment o disminució. Aquesta instància és la que s'encarrega primerament de mirar que la part no tingui la mida màxima en cas d'augment, ni que tingui la mida mínima en cas de disminució, i si no és cap d'aquests casos, és la que realitza els canvis de mida corresponents (Veure Figures 64 i 65).

```
if (mida)
{
    mascotaAtributs.instance.canviMida(parts[i], nMida);
}
```

Figura 64: Part de *l'script buttonClicked.cs* que s'executa quan es canvia la mida d'una part

```
1 referencia
public void canviMida(partController part, float m)
{
    string nom = part.getNom();

    atribut a = new atribut();
    a = dAtributs[nom];
    a.canviMida(m);
    dAtributs[nom] = a;

    setPart(part);
}
```

Figura 65: Funció de l'*script mascotaAtributs.cs* que canvia la mida d'una part

6.2.3.2.3. Canvi de posició

La funcionalitat de canvi de posició d'una part és una mica més complexa que les anteriors, ja que varia si es vol canviar la posició vertical o horitzontal, igual que si només es vol moure o si es vol ajuntar o separar dues parts. És per això que es va crear una variable booleana anomenada *Vertical*, i només amb aquesta variable i les altres que s'havien creat es pot gestionar tots els moviments diferents de les parts, tal i com es pot veure a l'esquema de la Figura 66 i al codi de la Figura 67. Així doncs, es seguirà utilitzant l'*script buttonClicked.cs* pels canvis de posicions i es cridaran diferents funcions de l'*script mascotaAtributs.cs* segons la funcionalitat en concret.

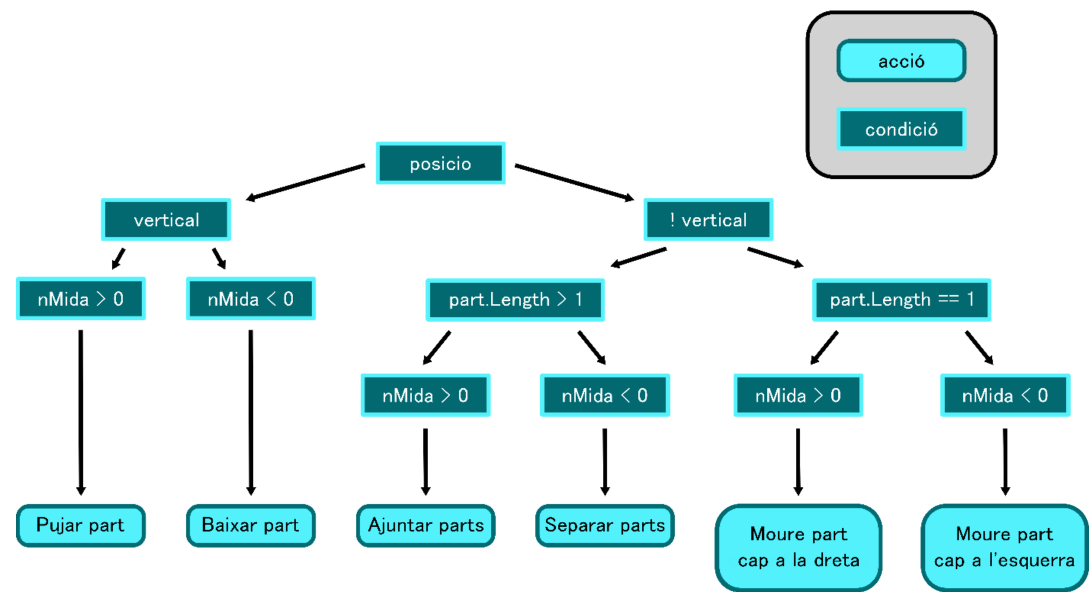


Figura 66: Esquema de condicions de l'*script buttonClicked* per saber quin tipus de canvi de posició s'ha de fer

```
if (posicio & !vertical)
{
    if (parts.Length > 1)
    {
        mascotaAtributs.instance.canviPosicioH2(parts[i], nMida);
    }
    else
    {
        mascotaAtributs.instance.canviPosicioH(parts[i], nMida);
    }

}
if (posicio & vertical)
{
    mascotaAtributs.instance.canviPosicioV(parts[i], nMida);
}
i += 1;
```

Figura 67: Codi de l'*script buttonClicked.cs* que representa el flux de condicions de l'esquema de la Figura 65

Totes les funcions que es criden de l'*script mascotaAtributs.cs* relacionades amb el moviment utilitzen una estructura similar a la funció que per canviar la mida de la part, en aquestes funcions s'utilitzen els límits establerts per cada part: el màxim o mínim d'amunt que poden estar, el màxim o mínim d'avall que poden estar i els mateix per l'esquerra i la dreta. A la següent figura es pot veure un exemple de com són aquestes funcions, en concret es mostra la funció que s'utilitza per moure les parts de la mascota verticalment.

```
1 referencia
public void canviPosicioV(float p)
{
    //PUJAR
    if (p > 0)
    {
        if (posicio.y + p <= MAX_AMUNT)
        {
            posicio = new Vector3(posicio.x, posicio.y + p, posicio.z);
        }
    }
    //BAIXAR
    else
    {
        if (posicio.y + p >= MAX_AVALL)
        {
            posicio = new Vector3(posicio.x, posicio.y + p, posicio.z);
        }
    }
}
```

Figura 68: Funció de l'*script mascotaAtributs.cs* per moure les parts de la mascota verticalment

6.2.3.2.4. Canvi de color

El canvi de color de les parts de la mascota és l'única funcionalitat que no utilitza l'*script buttonClicked.cs*, ja que es va voler utilitzar un nou mètode d'*Input*, és a dir, que no fossin botons amb colors que s'haguessin de prémer per establir-lo com a color de la part, a causa de que si es volen posar molts colors hi ha molta feina en anar creant botons i poden ocupar moltíssim espai a la pantalla. És per això que es va optar per posar una paleta de colors amb un marcador (Veure Figura 69), d'aquesta manera el jugador podria optar a moltíssims més colors i fer la mascota encara més personalitzada.

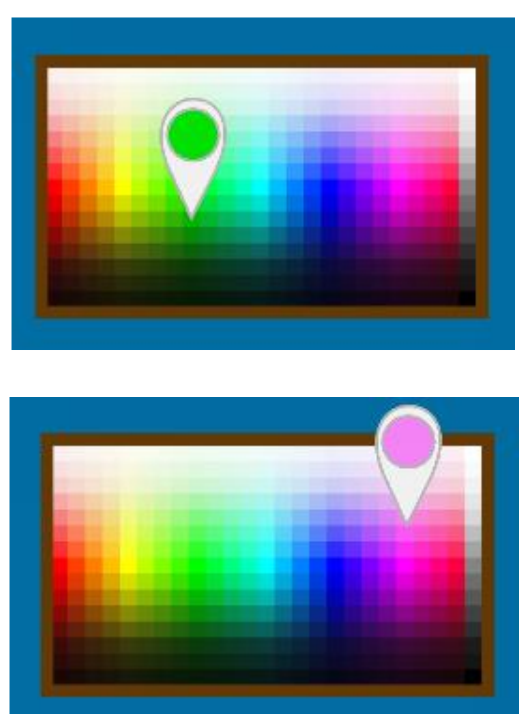


Figura 69: Paleta i marcador per canviar el color de les parts de la mascota

D'aquesta manera, el funcionament d'aquesta paleta està implementat en un *script* anomenat *colorPicker.cs*. Aquest *script* té un *array* de totes les parts a les quals se li ha de canviar el color, a més d'una funció *Update()* que s'executa a cada *frame*, on primerament comprova si el jugador està tocant la paleta. Si això es compleix, mostra el marcador amb el color que està tocant i canvia el color de les parts de l'*array* pel color seleccionat, fent que el jugador pugui veure com queda aquell color que ha seleccionat a la mascota instantàniament. El canvi de color es fa cridant a la instància que emmagatzema tota la informació dels atributs, igual que s'ha fet a les funcionalitats anteriors.

Entrant a la part més tècnica d'aquesta funcionalitat, per implementar aquest canvi de color a partir d'una paleta primer s'obté el component *RectTransform* de la paleta, el qual és essencial en imatges 2D ja que conté característiques com la mida i la posició de la imatge. Seguidament es converteix les coordenades de la pantalla en coordenades dins d'aquest component, d'aquesta manera sabrem exactament si el jugador està tocant la paleta i en quina posició. Això es farà també obtenint l'altura i l'amplada de la paleta, i fent servir aquests paràmetres per establir uns màxims i mínims. Finalment es creen noves coordenades, les quals tindran valors d'entre 0 i 1 utilitzant aquests màxims i mínims. D'aquesta manera, s'obtindrà la textura del color de la paleta que s'està seleccionant, el qual es troba en aquestes coordenades, i es convertirà aquesta textura en color, el qual es mostrarà tan al marcador com a les parts a les que s'ha de canviar el color. Aquest color és el mateix que es passa per paràmetre a la funció de l'*script* *mascotaAtributs.cs* per efectuar els canvis de colors necessaris. A la següent figura es mostra tot el codi que s'ha explicat en aquest paràgraf.

```
Mensaje de Unity | 0 referencias
void Update()
{
    if (RectTransformUtility.RectangleContainsScreenPoint(Rect, Input.mousePosition) && Input.GetMouseButton(0))
    {
        marker.SetActive(true);
        marker.transform.position = new Vector3(Input.mousePosition.x, Input.mousePosition.y + 70, Input.mousePosition.z);

        Vector2 delta;
        RectTransformUtility.ScreenPointToLocalPointInRectangle(Rect, Input.mousePosition, null, out delta);

        float width = Rect.rect.width;
        float height = Rect.rect.height;
        delta += new Vector2(width * .5f, height * .5f);

        float x = Mathf.Clamp(delta.x / width, 0f, 1f);
        float y = Mathf.Clamp(delta.y / height, 0f, 1f);

        int texX = Mathf.RoundToInt(x * ColorTexture.width);
        int texY = Mathf.RoundToInt(y * ColorTexture.height);

        Color color = ColorTexture.GetPixel(texX, texY);

        onColorPreview?.Invoke(color);

        int i = 0;
        while (i < parts.Length)
        {
            mascotaAtributs.instance.canviColor(parts[i], color);
            i++;
        }
    }
    else
    {
        marker.SetActive(false);
    }
}
```

Figura 70: Funció *Update()* de l'*script colorPicker.cs*

6.2.3.2.4. Mostrar i amagar complements

Finalment, una última funcionalitat del laboratori és la de mostrar i amagar els complements de la mascota quan aquesta s'està modificant. S'ha implementat aquesta funcionalitat per fer-li més còmode la personalització de la mascota al jugador, ja que depenent de quins complements tingui posats la mascota poden fer que el jugador no vegi els canvis que està fent. Per exemple, si la mascota porta posades unes ulleres opaques i el jugador està fent canvis als ulls de la mateixa mascota, podrà tenir l'opció d'amagar les ulleres per veure els canvis que està fent als ulls. Per implementar aquesta funcionalitat, s'han creat uns botons que quan es premen activen o desactiven els complements de la mascota, però també s'ha hagut de crear un objecte que gestioni a través d'un *script* anomenat *showHideComp.cs* si s'han de mostrar els botons per mostrar els complements o els botons que amaguen els complements (Veure Figures 71, 72 i 73).

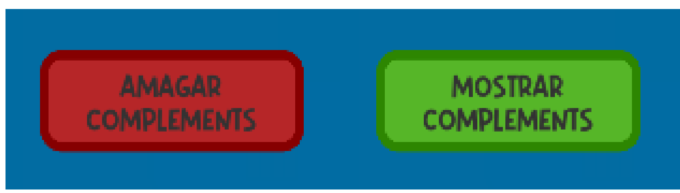
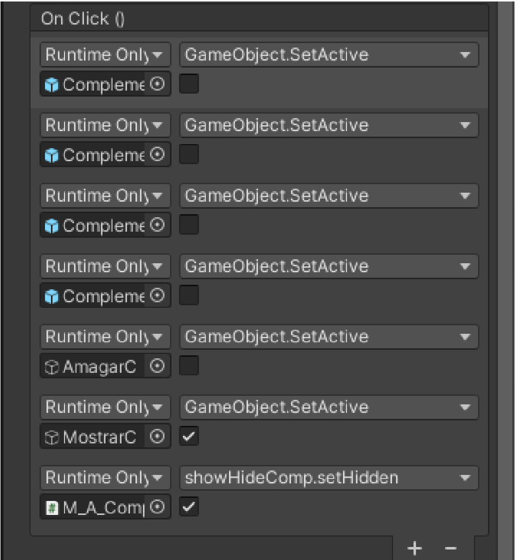


Figura 71: Botons per mostrar i amagar els complements de la mascota



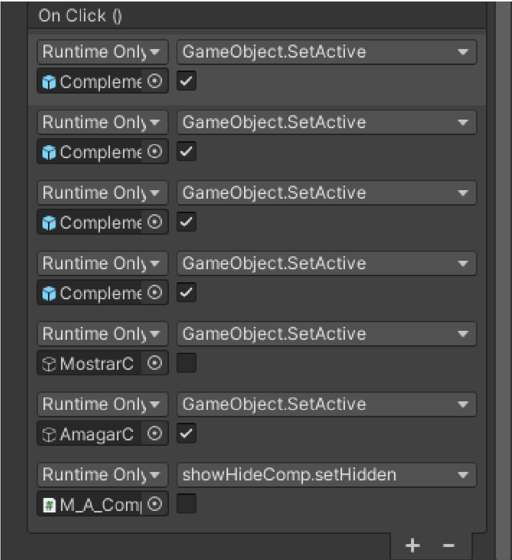


Figura 72: Funcionalitats dels botons d'amagar i mostrar complements respectivament

```
0 referencias
public void uptadeButton(int pos)
{
    hideButtons[pos].SetActive(!hidden);
    showButtons[pos].SetActive(hidden);
}

0 referencias
public void setHidden(bool state)
{
    hidden = state;
}
```

Figura 73: Funcions de l'*script showHideComp.cs* que gestiona els botons que s'han de mostrar i els que no

6.2.3.3. Distribució de les parts

Un cop estudiades les diverses funcionalitats en l'àmbit de modificació de parts, en aquest apartat es veurà com s'han distribuït aquestes funcionalitats a cadascuna i com ha quedat tot cohesionat a l'espai del laboratori. Tal i com s'ha anat explicant als apartats anteriors, s'ha implementat de manera que, per dur a terme cada funcionalitat, només s'ha d'afegir un botó i associar-li l'*script buttonClicked.cs* amb els paràmetres corresponents, a excepció del canvi de color per al que s'ha implementat un *Prefab* anomenat *colorPicker* que només s'ha d'afegir allà on es vulgui efectuar aquest canvi, i associar-lo a les parts a les que es vol efectuar aquest canvi.

Per organitzar totes les diferents modificacions que es poden fer a la mascota, s'han creat diferents botons (Veure Figura 74) que obren diversos menús, on es troben les modificacions que es poden efectuar a cada part. A continuació es veurà com s'han distribuït aquestes modificacions de cada part afegint-hi les funcionalitats que s'han explicat als apartats anteriors.

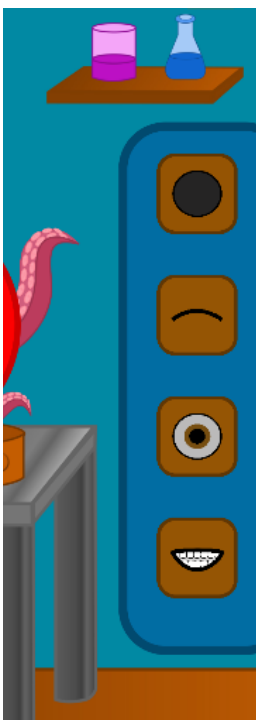


Figura 74: Panell amb els botons de les modificacions de cada part

6.2.3.3.1. Modificació del cos

Al cos de la mascota només se li poden fer dos tipus de modificacions, la forma i el color, ja que variar la mida o la posició podria portar problemes de compatibilitat amb altres aspectes del projecte, com per exemple els minijocs. D'aquesta manera, dins del menú de modificació del cos hi ha dos panells, tal i com es pot veure a les Figures 75 i 76.

Figura 75: Panell 1 del menú de modificació del cos de la mascota

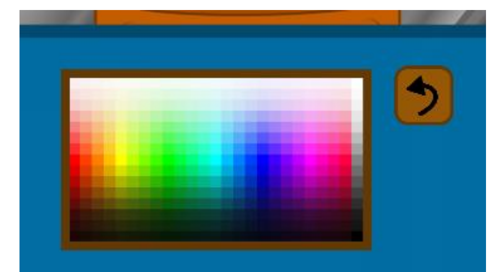


Figura 76: Panell 2 del menú de modificació del cos de la mascota

Panell 1. Canvi de forma: En aquest panell hi ha 6 botons. La funcionalitat de canvi de forma es fa amb 3 d'ells, les dues fletxes per mirar la forma següent o l'anterior i la imatge de la forma que es pot canviar a la mascota. Les dues fletxes utilitzen la funció *chamgeImage()* de l'*script stockPart.cs* vist a l'Apartat 6.2.3.2.1, i l'altre botó té associat l'*script buttonClicked.cs*, amb la variable booleana *Forma* seleccionada i amb el cos de la mascota dins de l'*array* de parts que es modifiquen. Els altres 3 botons serveixen per mostrar o amagar els complements, sortir del menú de modificació del cos de la mascota i per obrir el següent panell del menú i poder canviar el color del cos de la mascota.

Panell 2. Canvi de color: Al segon panell hi ha el botó per tornar al primer panell del menú i el *Prefab colorPicker*, utilitzat per canviar el color de les diferents parts de la mascota. En aquest cas, s'hi associa el cos de la mascota com a part modificable.

6.2.3.3.2. Modificació de les celles

A les celles de la mascota se li poden fer 3 tipus de canvis: se li pot modificar la forma, la mida i la posició. D'aquesta manera, dins del menú de modificació de les celles, hi ha dos panells, tal i com es pot veure a les Figures 77 i 78.

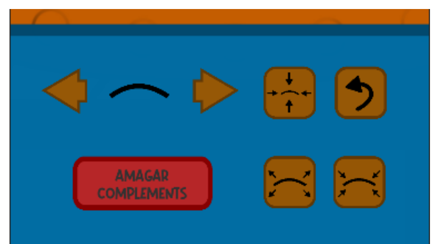


Figura 77: Panell 1 del menú de modificació de les celles de la mascota

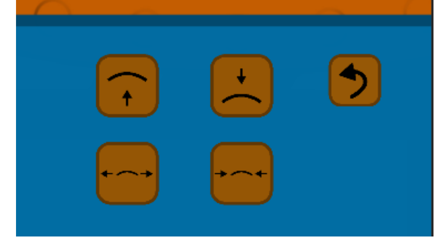


Figura 78: Panell 2 del menú de modificació de les celles de la mascota

Panell 1. Canvi de forma + Canvi de mida: En aquest panell hi ha 8 botons. La funcionalitat de canvi de forma es fa amb 3 d'ells, les dues fletxes per mirar la forma següent o l'anterior i la imatge de la forma que es pot canviar a la mascota. Les dues fletxes utilitzen la funció *chamgeImage()* de l'*script stockPart.cs* vist a l'Apartat 6.2.3.2.1, i l'altre botó té associat l'*script buttonClicked.cs*, amb la variable booleana *Forma* seleccionada i amb les celles de la mascota dins de l'*array* de parts que es modifiquen. 2 d'aquests 8 botons s'utilitzen per augmentar i disminuir la mida de les celles. Ambdós botons tenen el mateix *script buttonClicked.cs* associat amb els mateixos paràmetres, l'únic que varia és el paràmetre de *nMida*: al botó d'augment de mida és un valor positiu i al de disminució de mida és un valor negatiu. Els altres 3 botons serveixen per mostrar o amagar els complements, sortir del menú de modificació de les celles de la mascota i per obrir el següent panell del menú i poder canviar la posició de les celles de la mascota.

Panell 2. Canvi de posició: Al segon panell hi ha el botó per tornar al primer panell del menú i 4 botons per moure les celles. Tots aquests 4 botons tenen el mateix *script buttonClicked.cs* associat amb la mateixa variable booleana *Posicio* seleccionada, a part de tenir les dues celles a l'*array* de parts. El que sí canvia és que, als botons de pujar i baixar les celles, també està seleccionada la variable *Vertical*. A més, tan al botó de pujar les celles com al de separar-les, el valor de la variable *nMida* és positiu, mentre que als altres dos botons és un valor negatiu.

6.2.3.3.3. Modificació dels ulls

Als ulls de la mascota se li poden fer tots els tipus de canvis possibles: se li pot modificar la forma, la mida, la posició i el color. D'aquesta manera, dins del menú de modificació dels ulls, hi ha tres panells, tal i com es pot veure a les Figures 79, 80 i 81.

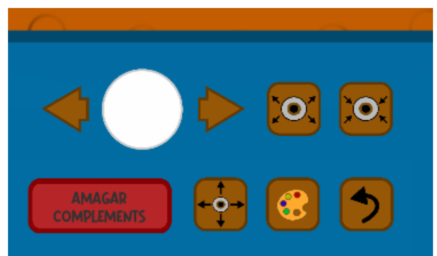


Figura 79: Panell 1 del menú de modificació dels ulls de la mascota

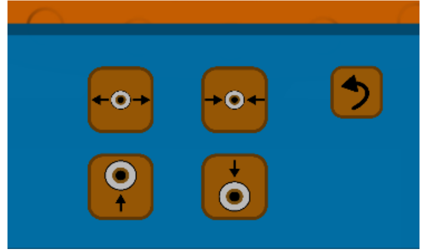


Figura 80: Panell 2 del menú de modificació dels ulls de la mascota

Figura 81: Panell 3 del menú de modificació dels ulls de la mascota

Panell 1. Canvi de forma + Canvi de mida: En aquest panell hi ha 9 botons. La funcionalitat de canvi de forma es fa amb 3 d'ells, les dues fletxes per mirar la forma següent o l'anterior i la imatge de la forma que es pot canviar a la mascota. Les dues fletxes utilitzen la funció *chamgeImage()* de l'*script stockPart.cs* vist a l'Apartat 6.2.3.2.1, i l'altre botó té associat l'*script buttonClicked.cs*, amb la variable booleana *Forma* seleccionada i amb els ulls de la mascota dins de l'*array* de parts que es modifiquen. 2 d'aquests 8 botons s'utilitzen per augmentar i disminuir la mida dels ulls. Ambdós botons tenen el mateix *script buttonClicked.cs* associat amb els mateixos paràmetres, l'únic que varia és el paràmetre de *nMida*: al botó d'augment de mida és un valor positiu i al de disminució de mida és un valor negatiu. Els altres 4 botons serveixen per mostrar o amagar els complements, sortir del menú de modificació de les celles de la mascota i per obrir els següents panells del menú i poder canviar la posició i el color dels ulls de la mascota.

Panell 2. Canvi de posició: Al segon panell hi ha el botó per tornar al primer panell del menú i 4 botons per moure els ulls. Tots aquests 4 botons tenen el mateix *script buttonClicked.cs* associat amb la mateixa variable booleana *Posicio* seleccionada, a part de tenir els dos ulls a l'*array* de parts. El que sí canvia és que als botons de pujar i baixar els ulls també està seleccionada la variable *Vertical*. A més, tan al botó de pujar els ulls com al de separar-los, el valor de la variable *nMida* és positiu, mentre que als altres dos botons és un valor negatiu.

Panell 3. Canvi de color: Al tercer panell hi ha el botó per tornar al primer panell del menú i el *Prefab colorPicker*, utilitzat per canviar el color de les diferents parts de la mascota. En aquest cas, s'hi associa l'iris dels ulls de la mascota com a parts modificables.

6.2.3.3.4. Modificació de la boca

A la boca de la mascota se li poden fer 3 tipus de canvis: se li pot modificar la forma, la mida i la posició. D'aquesta manera, dins del menú de modificació de la boca, hi ha dos panells, tal i com es pot veure a les Figures 82 i 83.

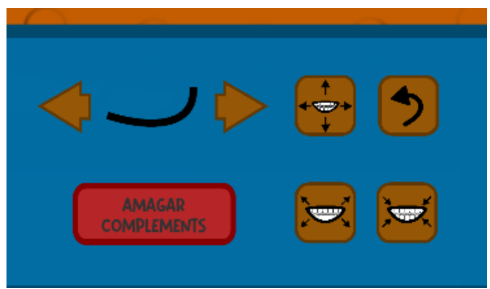


Figura 82: Panell 1 del menú de modificació de la boca de la mascota

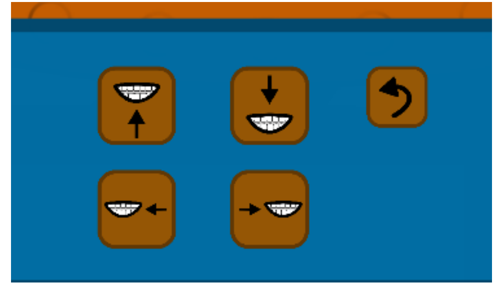


Figura 83: Panell 2 del menú de modificació de la boca de la mascota

Panell 1. Canvi de forma + Canvi de mida: En aquest panell hi ha 8 botons. La funcionalitat de canvi de forma es fa amb 3 d'ells, les dues fletxes per mirar la forma següent o l'anterior i la imatge de la forma que es pot canviar a la mascota. Les dues fletxes utilitzen la funció *chamgeImage()* de l'*script stockPart.cs* vist a l'Apartat 6.2.3.2.1, i l'altre botó té associat l'*script buttonClicked.cs*, amb la variable booleana *Forma* seleccionada i amb la boca de la mascota dins de l'*array* de parts que es modifiquen. 2 d'aquests 8 botons s'utilitzen per augmentar i disminuir la mida de la boca. Ambdós botons tenen el mateix *script buttonClicked.cs* associat amb els mateixos paràmetres, l'únic que varia és el paràmetre de *nMida*: al botó d'augment de mida és un valor positiu i al de disminució de mida és un valor negatiu. Els altres 3 botons serveixen per mostrar o amagar els complements, sortir del menú de modificació de la boca de la mascota i per obrir el següent panell del menú i poder canviar la posició de la boca de la mascota.

Panell 2. Canvi de posició: Al segon panell hi ha el botó per tornar al primer panell del menú i 4 botons per moure la boca. Tots aquests 4 botons tenen el mateix *script buttonClicked.cs* associat amb la mateixa variable booleana *Posicio* seleccionada, a part de tenir la boca a l'*array* de parts. El que sí canvia és que als botons de pujar i baixar la boca també està seleccionada la variable *Vertical*. A més, tan al botó de pujar la boca com al de moure-la cap a la dreta, el valor de la variable *nMida* és positiu, mentre que als altres dos botons és un valor negatiu.

6.2.3.4. Animacions de panells i menús

Un cop les funcionalitats van estar implementades i distribuïdes, es va decidir animar les transicions entre menús i panells per donar un toc més professional al joc. Aquestes animacions consisteixen en fer zoom a les parts que s'estan modificant, perquè el jugador les vegi millor, i fer una animació d'aparèixer i desaparèixer pels panells, donant l'efecte de que es superposen uns amb els altres (Veure Figures 84 i 85). Per fer això es va crear un *script* anomenat *panelController.cs*. Aquest *script* l'únic que fa és executar animacions dels panells, així que a part d'associar aquests *script* als panells s'ha d'afegir un component Animator, crear les animacions i afegir-les al component. Després l'únic que s'ha de fer és afegir dues tasques als botons que es premen quan es realitzen aquestes animacions: escollir l'animació que s'ha d'executar i cridar una funció de l'*script panelController.cs* que inicia l'animació (Veure Figura 86).

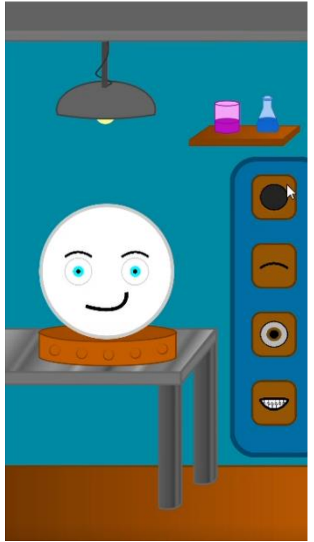
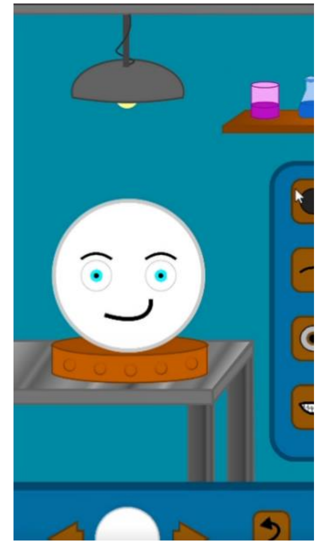
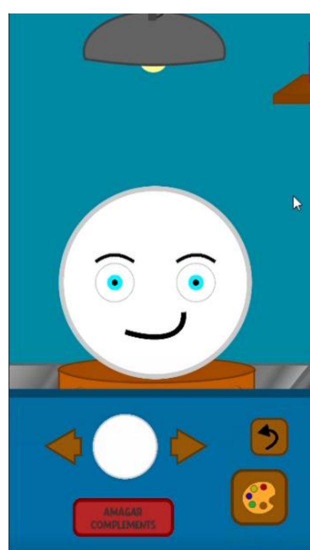




Figura 84: Animació fent zoom a les parts de la mascota quan es modifiquen

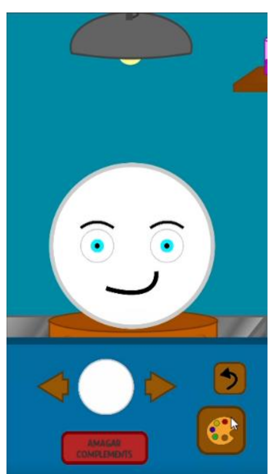
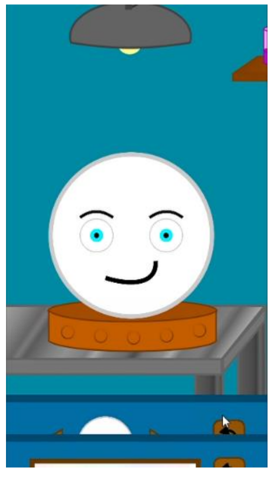
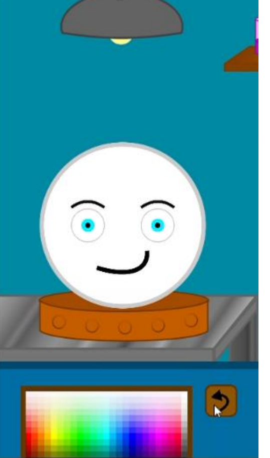
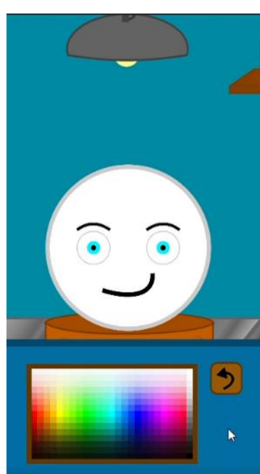





Figura 85: Animació de superposició entre panells

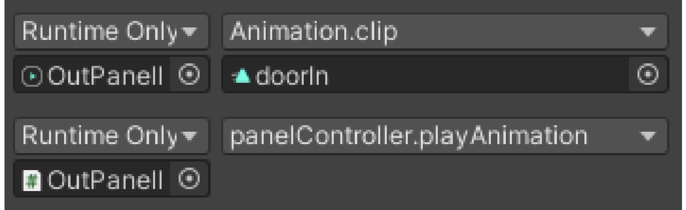


Figura 86: Exemple de tasques que se li associen als botons que han d'executar una animació

6.2.4. Cuina

La cuina és un altre dels espais més importants del joc, ja que és on se li dona de menjar a la mascota i alhora és on es compra aquest menjar (Veure Figura 87). A nivell d'implementació és tot un repte el fet de desenvolupar aquest espai sense tenir en compte quants tipus de menjar s'implementaran, ja que, tenint un temps i uns recursos tan limitats, el millor que es pot fer és un desenvolupament que sigui òptim per tenir uns mínims, però que també permeti ampliar el contingut del joc mb facilitat.

Figura 87: Espai de la cuina

6.2.4.1. Objectes de l'espai

En aquest espai hi ha els següents elements que es poden veure a la Figura 88, els quals es comentaran posteriorment.

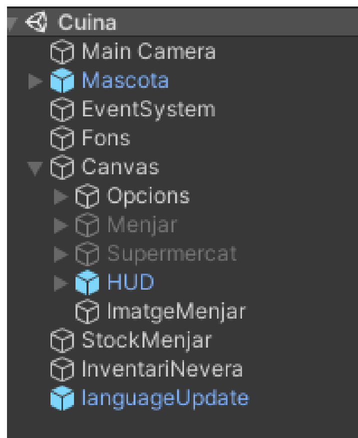


Figura 88: Objectes del la cuina a la finestra de jerarquia

- **Mascota**: És el *Prefab* que s'ha comentat a l'apartat 6.1.4, on no s'hi ha hagut d'afegir cap *script* addicional.
- **Fons:** Imatge de decoració de la cuina.
- **Opcions:** Menú que conté les diferents opcions d'accions que es poden fer a la cuina.
- **Menjar:** Menú que mostra el menjar que té el jugador per donar-li a la mascota.
- **Supermercat:** Menú que mostra el menjar que pot comprar el jugador.
- **HUD**: Informació que es mostra per pantalla al jugador comentada a l'apartat 5.6.1.
- **ImatgeMenjar:** Imatge que s'instancia a la posició del ratolí quan el jugador li dona de menjar a la mascota.
- **StockMenjar:** Objecte que gestiona tot el menjar que hi ha disponible al joc i que pot comprar el jugador.
- **InventariNevera:** Objecte que gestiona el menjar que té el jugador per donar-li a la mascota.
- **LanguageUpdate**: Gestor de textos per canviar d'idioma (s'aprofundirà als següents apartats).

6.2.4.2. Funcionalitats

A continuació s'explicarà la implementació de les dues mecàniques que hi ha a la cuina, que són les de donar de menjar a la mascota i comprar aquest menjar.

6.2.4.2.1. Comprar menjar

Abans d'entrar en detall de la implementació de la compra de menjar, s'ha d'explicar com es va dissenyar aquesta mecànica. Des d'un principi es va fer una llista amb diferents aliments que podrien aparèixer al joc, després es van organitzar aquests aliments en diferents categories i es van anar afegint més aliments per a què cada categoria tingués el mateix número d'aliments.

Sabent com es volia organitzar tot, i tal i com s'ha dit al principi d'aquest apartat, les mecàniques de la cuina han de ser desenvolupades de forma òptima, pensant que hi ha d'haver uns mínims al joc però tenint en compte que s'ha de poder ampliar el contingut de forma senzilla. És per això que es va crear un *script* anomenat *generadorMenjar.cs* que s'encarrega de gestionar les diferents categories d'aliments que hi pot haver al joc i mostrar el menjar corresponent. Es va crear també un *Prefab* anomenat *CategoriaMenjar* que conté els botons per accedir a les diferents categories de menjar i els botons que fan referència a cada aliment (Veure Figure 89 i 90).

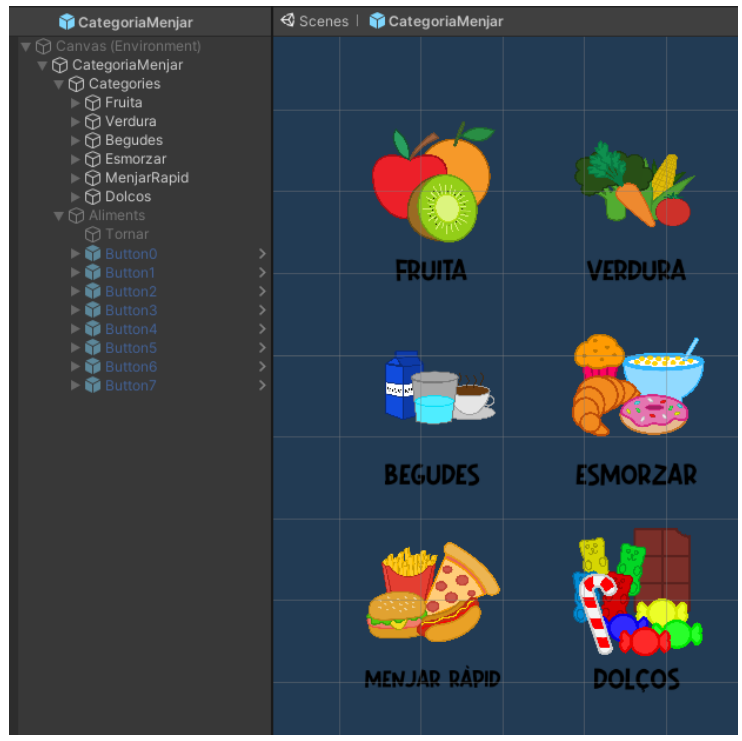


Figura 89: Botons de les categories de menjar del *Prefab CategoriaMenjar*

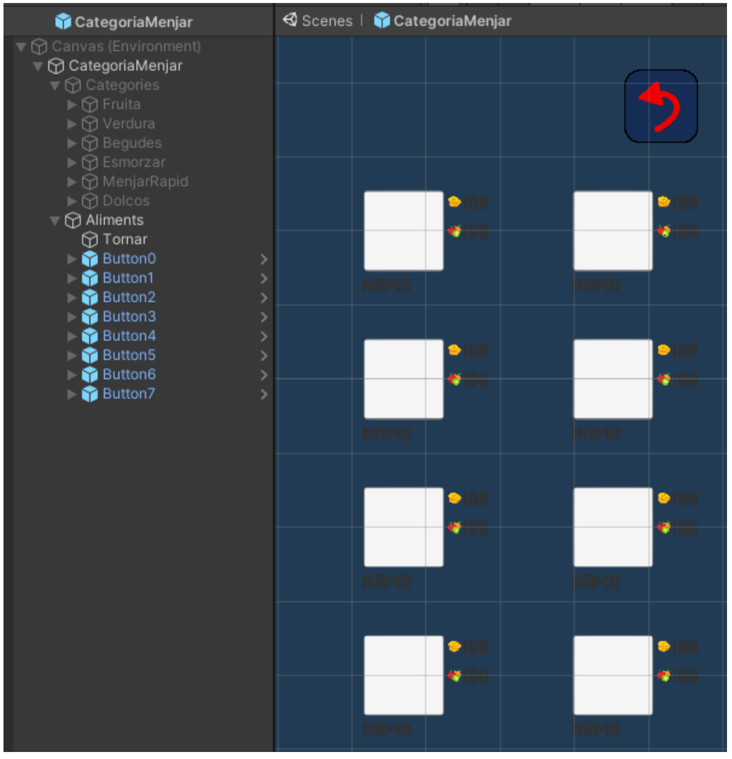


Figura 90: Botons dels diferents aliments del *Prefab CategoriaMenjar*

D’aquesta manera, l’*script generadorMenjar.cs* té una variable que és un *array* de botons. Aquests botons són els 6 que es veuen a la Figura 89, i té una funció anomenada *updButtons()* que executa cada botó quan és premut. Segons el botó que sigui, aquesta funció actualitza l’aspecte dels altres 8 botons que es veuen a la Figura 90. Aquest mètode d’implementació permet anar afegint categories de menjar sense haver d’afegir botons d’aliments per a cada una d’aquestes categories, sinó que hi ha sempre uns 8 botons que s’actualitzen segons la categoria corresponent.

Abans d’explicar com funciona la funció *updButtons()*, cal que s’expliqui com estan organitzats els diferents aliments. Així doncs, per gestionar tot el menjar del joc es va crear un *script* anomenat *foodStock.cs*. En aquest *script* se li va posar un *Singleton*, d’aquesta manera només hi ha una instància en tot el joc. També s’hi va crear una classe anomenada *food*, que conté tota la informació de cada aliment i funcions per accedir a aquesta informació. Els atributs d’aquest objecte són el nom identificador de l’aliment, el preu, el valor de gana que li treu a la mascota i el nom de l’aliment en 3 idiomes.

Finalment també es van crear dues variables per emmagatzemar tot el menjar del joc, un diccionari anomenat *dFood*, que té com a clau el nom de la categoria i com a valor la llista de tots els aliments d’aquella categoria, i un altre diccionari anomenat *categoryFood*, que té com a clau el nom identificador de l’aliment i com a valor el nom de la categoria a la que pertany (Veure Figura 91).

```
70 referencias
public class food
{
    string nom, categoria, nomCAT, nomCAST, nomENG;
    int valor, gana;
```

```
Dictionary<string, List<food>> dFood = new Dictionary<string, List<food>>();
Dictionary<string, string> categoryFood = new Dictionary<string, string>();
```

Figura 91: Classe *food* i variables de l’*script foodStock.cs*

Amb aquesta estructura, quan l'*script foodStock.cs* s'executa per primera i única vegada, a la funció *Start()* es creen els diferents aliments amb una funció *init()* que s'utlitza com a constructor de la classe *food*, s'emmagatzemen al primer diccionari segons cada categoria i s'omple el segon diccionari amb els noms de cada aliment i la categoria corresponent (Veure Figures 92 i 93).

```
//FRUITA
List<food> lFruita = new List<food>();

food apple = new food();
apple.init("apple", 10, 5, "fruita", "POMA", "MANZANA", "APPLE");
lFruita.Add(apple);

food banana = new food();
banana.init("banana", 10, 5, "fruita", "PLÀTAN", "PLÁTANO", "BANANA");
lFruita.Add(banana);

food cherries = new food();
cherries.init("cherries", 10, 5, "fruita", "CIRERES", "CEREZAS", "CHERRIES");
lFruita.Add(cherries);

food kiwi = new food();
kiwi.init("kiwi", 10, 5, "fruita", "KIWI", "KIWI", "KIWI");
lFruita.Add(kiwi);

food orange = new food();
orange.init("orange", 10, 5, "fruita", "TARONJA", "NARANJA", "ORANGE");
lFruita.Add(orange);

food pear = new food();
pear.init("pear", 10, 5, "fruita", "PERA", "PERA", "PEAR");
lFruita.Add(pear);

food strawberry = new food();
strawberry.init("strawberry", 10, 5, "fruita", "MADUIXA", "FRESA", "STRAWBERRY");
lFruita.Add(strawberry);

food watermelon = new food();
watermelon.init("watermelon", 10, 5, "fruita", "SÍNDRIA", "SANDÍA", "WATERMELON");
lFruita.Add(watermelon);


dFood.Add("fruita", lFruita);
```

Figura 92: Exemple de creació d'aliments d'una categoria

```
//OMPLIR CATEGORIES
foreach (KeyValuePair<string, List<food>> pair in dFood)
{
    foreach(food f in pair.Value)
    {
        categoryFood.Add(f.getNom(),pair.Key);

    }


}
```

Figura 93: Bucle *for* per omplir el diccionari amb cada aliment i la categoria corresponent

També cal comentar que per utilitzar aquesta mecànica, tal i com està implementada, cal desar les imatges de cada aliment en la carpeta corresponent, ja que cada una de les carpetes fa referència a una categoria de menjar i els *scripts* encarregats de gestionar l'aliment accedeixen a aquestes carpetes, segons l'aliment que s'hagi de mostrar, comprar o donar a la mascota (Veure Figura 94).

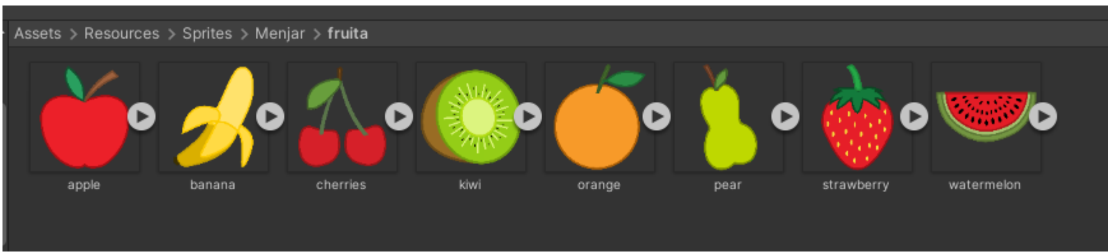


Figura 94: Carpeta d'imatges dels aliments de la categoria de fruita

D'aquesta manera, tornant a la explicació de com funciona el *Prefab CategoriaMenjar*, cada vegada que es prem un botó d'una categoria d'aliments, aquest executa la funció *updButtons()*. Aquesta funció, segons la categoria que faci referència el botó que s'ha premut, accedeix a la carpeta corresponent d'aquella categoria i, per a cada un dels botons dels aliments, actualitza la informació. Per exemple, si es prem el botó de la categoria de fruita, la funció fa que els 8 botons d'aliments facin referència a les 8 fruites disponibles del joc, mostrant la imatge i la informació corresponents. Per mostrar la imatge ja s'ha comentat que s'accedeix a la carpeta que toca, d'altra banda per actualitzar la informació (que és el nom de l'aliment, el valor o preu i el percentatge de gana que treu a la mascota) crida a la instància del joc que té l'*script foodStock.cs* associat i s'obtenen totes les dades de cada aliment, les quals estan emmagatzemades als diccionaris que s'han esmentat anteriorment (Veure Figura 95).

```
0 referencias
public void updButtons(string s)
{
    categoria = s;

    string carpetaAssets = Application.dataPath;
    string carpetaMenjar = carpetaAssets + "/Resources/Sprites/Menjar/" + s;

    DirectoryInfo dir = new DirectoryInfo(carpetaMenjar);
    FileInfo[] info = dir.GetFiles("*.png");

    for (int i = 0; i < buttons.Length; i++)
    {
        int posPunt = info[i].Name.IndexOf(".");
        string nom = info[i].Name.Substring(0, posPunt);
        buttons[i].image.sprite = Resources.Load<Sprite>("Sprites/Menjar/" + s + "/" + nom);

        int valor = foodStock.instance.getValor(categoria, nom);
        int gana = foodStock.instance.getGana(categoria, nom);

        string nomPerMostrar = foodStock.instance.getNomM(categoria, nom, languageController.instance.lanToNum());

        buttons[i].transform.GetChild(0).GetComponent<Text>().text = nomPerMostrar;
        buttons[i].transform.GetChild(1).GetComponent<Text>().text = valor.ToString();
        buttons[i].transform.GetChild(2).GetComponent<Text>().text = gana.ToString() + "%";
    }


}
```

Figura 95: Funció *updButtons()* de l'*script generadorMenjar.cs*

Un cop mostrats els aliments corresponents a cada categoria, s’ha d’implementar la funcionalitat de poder comprar aquests aliments. A cada un dels botons que fan referència a cada aliment se li ha associat com a tasca quan es prem, una nova funció de l’*script generadorMenjar.cs* anomenada *comprarMenjar()* (Veure Figures 96 i 97). Aquesta funció accedeix a la informació de l’aliment a través d’un nombre identificador, que també se li afegeix a cada botó, i comprova que aquell aliment es pugui comprar en funció de les monedes que tingui el jugador. Si això es compleix, li treu les monedes corresponents als jugador i compra aquest aliment, afegint-lo a un objecte que té associat un *script* anomenat *stockNevera.cs*, el qual s’encarrega d’emmagatzemar tot el menjar que té el jugador. A l’explicació de la següent funcionalitat s’analitza l’estructura d’aquest *script*.

```
0 referencias
public void comprarMenjar(int idMenjar)
{
    string carpetaAssets = Application.dataPath;
    string carpetaMenjar = carpetaAssets + "/Resources/Sprites/Menjar/" + categoria;

    DirectoryInfo dir = new DirectoryInfo(carpetaMenjar);
    FileInfo[] info = dir.GetFiles("*.png");

    int posPunt = info[idMenjar].Name.IndexOf(".");
    string nom = info[idMenjar].Name.Substring(0, posPunt);



    if (controladorMonedes.instance.monedesSuficients(foodStock.instance.getValor(categoria, nom))) {

        controladorMonedes.instance.treureMonedes(foodStock.instance.getValor(categoria, nom));
        stockNevera.instance.afegirMenjar(nom);
    }

}
```

Figura 96: Funció *comprarMenjar()* de l’*script generadorMenjar.cs*

Figura 97: Exemple de tasca associada al botó per comprar un aliment

6.2.4.2.2. Alimentar a la mascota

Per a la mecànica d'alimentar a la mascota, es va crear un *script* anomenat *stockNevera.cs* que gestiona els aliments que té el jugador, tal i com s'ha dit a l'apartat anterior, un altre *script* anomenat *fridgeInventory.cs,* que és el que mostra els aliments que té el jugador i el que els elimina quan se li dona a la mascota, i un últim *script* anomenat *foodImage()* que mostra la imatge de l'aliment que se li està donant a la mascota i gestiona l'animació de la boca de la mascota mastegant l'aliment. D'aquesta manera, l'estructura d'aquests *scripts* és la següent:

L'*script stockNevera.cs* té un *Singleton* que permet instanciar l'objecte amb aquest *script* una sola vegada. També té un diccionari on la clau és el nom de l'aliment i el valor és un nombre enter que representa la quantitat que té el jugador d'aquell aliment, i una llista amb els noms dels aliments que el jugador té (Veure Figura 98).

```
Dictionary<string, int> dNevera = new Dictionary<string, int>();

private List<string> foodNames = new List<string>();
```

Figura 98: Variables de l'*script stockNevera.cs*

D'altra banda, l'*script fridgeInventory.cs* també té un *Singleton* que permet instanciar l'objecte amb aquest *script* una sola vegada. També té com a variable la imatge que s'ha de mostrar quan el jugador no té cap aliment disponible, el botó que es prem quan es vol donar un aliment a la mascota, una referència a l'objecte que conté l'*script foodImage* i un nombre enter privat que serveix per gestionar el nombre d'aliments que s'han de mostrar (Veure Figura 99).

```
public Sprite noFoodImage;

public Button currentFood;

public foodImage foodIm;

private int idFood;
```

Figura 99: Variables de l'*script fridgeInventory.cs*

Finalment, l'*script foodImage.cs* conté una variable booleana per saber si s'ha de mostrar o no la imatge de l'aliment que se li està donant a la mascota, un *Canvas* i unes constants que serveixen com a límits per saber on està la boca de la mascota, la referència a la boca de la mascota i dues imatges per gestionar l'animació de la mascota mastegant (Veure Figura 100).

```
private bool active;

public Canvas myCanvas;

private float MAX_LEFT, MAX_RIGHT, MAX_UP, MAX_DOWN;

private Sprite bocaMascota;

public partController boca;

public Sprite bocaMenjar;
```

Figura 100: Variables de l'*script foodImage.cs*

Així doncs, hi ha un botó que quan es prem s'alimenta a la mascota. Aquest botó té una acció associada, que és la d'executar una funció anomenada *treureMenjar()* de l'*script fridgeInventory.cs*. Aquesta funció l'únic que fa és comprovar que el jugador tingui algun aliiment, i en cas de ser veritat, activa la imatge del menjar que el jugador ha d'arrossegar fins la boca de la mascota per alimentar-la. A més, el botó també té un component *Event Trigger*, que s'encarrega de gestionar què passa quan el jugador té l'aliment premut i quan el deixa de prémer. A través d'aquest *Event Trigger* s'activa o es desactiva la variable booleana de l'*script foodImage.cs,* i dins d'aquest mateix *script* es comprova *frame* per *frame*, amb la funció *FixedUpdate(),* si el jugador li dona de menjar o no a la mascota, utilitzant els límits que té de variables (Veure Figura 101).

```
void FixedUpdate()
{
    Vector2 pos;
    RectTransformUtility.ScreenPointToLocalPointInRectangle(myCanvas.transform as RectTransform, Input.mousePosition, myCanvas.worldCamera, out pos);
    transform.position = myCanvas.transform.TransformPoint(pos);
    Vector3 imagePos = myCanvas.transform.InverseTransformPoint(transform.position);

    if (active)
    {
        if (Input.GetMouseButton(0))
        {
            this.GetComponent<Image>().enabled = true;
        }
        else
        {
            this.GetComponent<Image>().enabled = false;
        }
    }
    else
    {
        if (!Input.GetMouseButton(0))
        {
            this.GetComponent<Image>().enabled = false;
        }
    }
    if (imagePos.x >= MAX_LEFT && imagePos.x <= MAX_RIGHT && imagePos.y >= MAX_DOWN && imagePos.y <= MAX_UP && this.GetComponent<Image>().enabled)
    {
        this.GetComponent<Image>().enabled = false;
        fridgeInventory.instance.alimentarMascota();
        StartCoroutine(animacioMenjar());
    }
}
```

Figura 101: Funció *FixedUpdate()* de l'*script foodImage.cs*

Tal i com es pot veure a la figura anterior, quan es comprova que s'està alimentant a la mascota, es crida la funció *alimentarMascota()* de la instància que té l'*script fridgeInventory.cs*. Aquesta funció s'encarrega de disminuir la gana de la mascota, a través de l'*script mascotaState.cs* i comprovant el percentatge de gana que s'ha de disminuir a través de l'*script foodStock.cs*, afegir punts de nivell al jugador, eliminar l'aliment que se li ha donat a la mascota i comprovar si al jugador se li ha acabat el menjar (Veure Figura 102).

```
1 referencia
public void alimentarMascota()
{
    if (stockNevera.instance.nFood() > 0)
    {
        string menjar = stockNevera.instance.getFoodName(idFood);
        stockNevera.instance.treureMenjar(menjar);

        mascotaState.instance.disminuirGana(foodStock.instance.getGana(foodStock.instance.getCategoria(menjar), menjar));

        lvlController.instance.afegirNivell(3);

        if (stockNevera.instance.getQuantityFoodByName(menjar) == 0)
        {
            stockNevera.instance.eliminarMenjar(menjar);

            if (stockNevera.instance.nFood() == 0)
            {
                currentFood.image.sprite = noFoodImage;
            }

            else
            {
                if (idFood == stockNevera.instance.nFood())
                {
                    idFood = 0;
                }
            }
        }

    }
}
```

Figura 102: Funció *alimentarMascota()* de l'*script fridgeInventory.cs*

6.2.4.3. Distribució de les funcionalitats

Un cop s'ha vist com s'han implementat les dues funcionalitats que es troben a la cuina, comprar menjar i alimentar a la mascota, seguidament s'analitzarà com s'han distribuït aquestes funcionalitats a l'espai de la cuina.

Primerament s'ha creat un panell a la part dreta de la pantalla amb dos botons, un carro de supermercat per comprar el menjar i una nevera per accedir al menjar que té el jugador i poder-li donar a la mascota (Veure Figura 103). Aquests dos botons obren dos panells per realitzar les dues funcionalitats que s'han comentat. A continuació s'analitzaran aquests dos panells.

Figura 103: Panell amb els botons de les funcionalitats de la cuina

6.2.4.3.1. Panell de comprar menjar

Quan es prem la icona del carro de supermercat, s'obre un panell amb el *Prefab CategoriaMenjar* que s'ha analitzat en apartats anteriors. D'aquesta manera, es veuen 6 botons amb totes les categories d'aliments, i quan es prem un d'aquests botons es veuen els 8 aliments d'aquella categoria corresponent (Veure Figura 104). També hi ha botons per tornar a l'espai de la cuina i per tornar al panell de les categories.

Figura 104: Panells per comprar el menjar de la mascota

6.2.4.3.1. Panell d'alimentar a la mascota

Quan es prem la icona de la nevera, s'obre un panell on es veuen els aliments que té el jugador disponible, amb la quantitat que té de cada un d'ells. Aquest número disminueix automàticament quan el jugador li dona l'aliment a la mascota, i si no li queden més unitats d'aquell menjar, es mostra el següent aliment que tingui el jugador. Si al jugador no li queda menjar, es veu una icona amb la nevera oberta i buida i un missatge que ho indica (Veure Figura 105).

Figura 105: Panells per donar menjar a la mascota

6.2.4.4. Animacions de panells i menús

De la mateixa manera que s'ha fet a l'espai del laboratori, s'han animats els panells que s'han vist a l'apartat anterior, per fer que el resultat sigui més atractiu estèticament. S'ha utilitzat l'*script panelController.cs* per gestionar aquestes animacions quan es premen els diferents botons dels panells i menús, seguint el mateix procediment que al laboratori. En aquest cas, s'ha tornat a optar per la superposició dels diferents panells, tan al menú de comprar menjar com al menú de donar menjar a la mascota (Veure Figures 106 i 107).

Figura 106: Animació de superposició dels panells per comprar el menjar de la mascota

Figura 107: Animació de superposició dels panells per donar menjar a la mascota

6.2.5. Lavabo

El lavabo és l'espai més simple a nivell d'implementació, ja que l'única funcionalitat és la de rentar a la mascota. A més, es tracta d'una mecànica que no necessita tampoc un disseny molt complex, ja que només s'ha de tenir en compte com es representarà la brutícia i com es traurà (Veure Figura 108).

Figura 108: Espai del lavabo

6.2.5.1. Objectes de l'espai

En aquest espai hi ha els següents elements que es poden veure a la Figura 109, els quals es comentaran posteriorment.

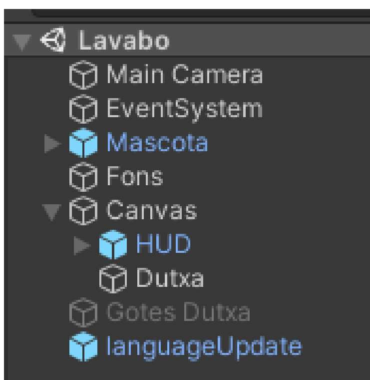


Figura 109: Objectes de l'espai del lavabo

- **Mascota**: És el *Prefab* que s'ha comentat a l'apartat 6.1.4, on no s'hi ha hagut d'afegir cap *script* addicional.
- **Fons:** Imatge de decoració del lavabo.
- **HUD**: Informació que es mostra per pantalla al jugador comentada a l'apartat 5.6.1.
- **Dutxa:** Botó que s'ha d'activar per rentar a la mascota.
- **Gotes Dutxa:** Sistema de partícules que apareix quan s'activa la dutxa, per representar el moment en que s'està rentant la mascota.
- **LanguageUpdate**: Gestor de textos per canviar d'idioma (s'aprofundirà als següents apartats).

6.2.5.2. Funcionalitats

Tal i com s'ha dit a la introducció d'aquest apartat, l'única funcionalitat que té el lavabo és la de rentar la mascota, i es va decidir que es representaria amb una dutxa. D'aquesta manera es va crear un *script* anomenat *controladorDutxa.cs*, el qual només té una variable booleana i una funció que serveix per activar o desactivar la dutxa segons aquesta variable. Aquesta funció en crida a una altra que està a l'*script* que controla l'estat de la mascota, i aquesta canvia el valor d'una altra variable booleana, la qual és clau al condicional de la funció que controla si la mascota s'està embrutant o rentant (Veure Figura 110). D'altra banda, a l'*script mascotaVisual.cs*, el qual controla l'aspecte de la mascota segons l' estat i els atributs, la funció *Update()* (que es crida cada *frame*) afegeix o disminueix la transparència d'unes taques que representen la brutícia de la mascota, en funció del que marqui l'estat (Veure Figures 111 i 112).

```
1 referencia
IEnumerator embrutar()
{
    while (true)
    {
        if (!dutxant)
        {
            if (bruticia < 100)
            {
                bruticia += AUGMENT_BRUTICIA;
                if (bruticia > 100)
                {
                    bruticia = 100;
                }
                yield return new WaitForSeconds(TEMPS_AUGMENT_BRUTICIA);
            }
        }

        else
        {
            if (bruticia > 0)
            {
                bruticia -= AUGMENT_BRUTICIA;
                if (bruticia <= 0)
                {
                    bruticia = 0;
                }
                else
                {
                    lvlController.instance.afegirNivell(0.2f);
                }
                yield return new WaitForSeconds(TEMPS_AUGMENT_BRUTICIA/5);
            }
        }

    }
}
```

Figura 110: Funció de l'*script mascotaState.cs* que modifica la brutícia de la mascota

```
void Update()
{
    if (mascotaState.instance.getBruticia() >= minBruticia)
    {
        for (int i = 0; i < nTaques; i++)
        {
            float quantitat = ((mascotaState.instance.getBruticia() - minBruticia) / (100 - minBruticia)); // 0 a 1
            float transparencia = transpTaques[i] * quantitat;

            Color c = taques.transform.GetChild(i).GetComponent<SpriteRenderer>().color;
            taques.transform.GetChild(i).GetComponent<SpriteRenderer>().color = new Color(c.r, c.g, c.b, transparencia);
        }
    }
}
```

Figura 111: Funció *Update()* de l'*script mascotaVisual.cs*

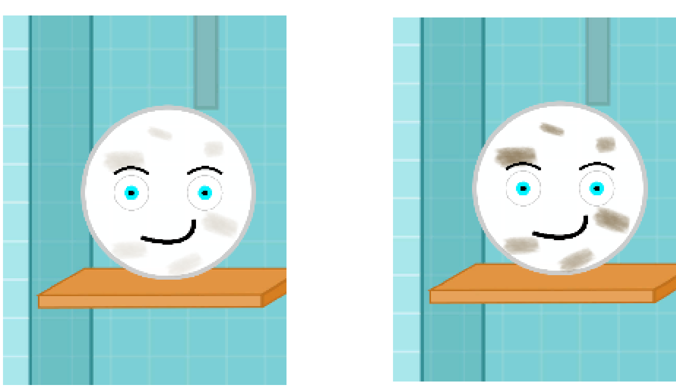


Figura 112: Mascota amb la meitat i amb el màxim de brutícia possible

Per acabar amb l'espai del lavabo, cal comentar com s'ha representat l'acció de rentar la mascota. A la dutxa se li ha posat per sobre un botó invisible, que quan aquest es prem, es crida la funció de l'*script controladorDutxa.cs*. Aquesta funció controla si s'ha engegat o parat la dutxa, i activa o desactiva un sistema de partícules que representen les gotes de la dutxa (Veure Figura 113).

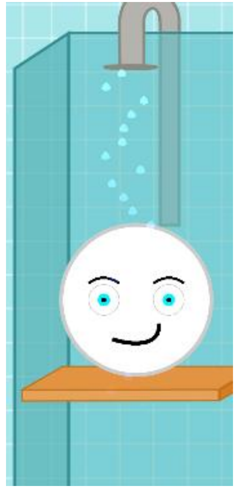


Figura 113: Partícules que representen les gotes de la dutxa

6.2.6. Habitació

L'habitació és l'últim espai que queda per explicar de l'interior de la casa. També és un espai important per les diverses funcionalitats que té, ja que s'hi pot comprar complements per a la mascota, vestir-la i posar-la a dormir. Pel que fa a la implementació d'aquest espai, ha estat un repte el fet de posar aquestes 3 mecàniques i que alhora no quedés molt carregat l'espai, com passa en algun dels jocs que s'han analitzat a l'estudi de mercat (Veure Figura 114).

Figura 114: Espai de l'habitació

6.2.6.1. Objectes de l'espai

En aquest espai hi ha els següents elements que es poden veure a la Figura 115, els quals es comentaran posteriorment.

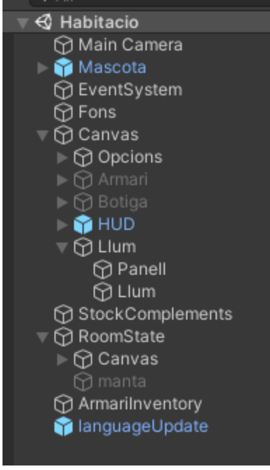


Figura 115: Objectes de l'espai de l'habitació

- **Mascota**: És el *Prefab* que s'ha comentat a l'apartat 6.1.4, on no s'hi ha hagut d'afegir cap *script* addicional.
- **Fons:** Imatge de decoració de l'habitació.
- **Opcions:** Menú que conté les diferents opcions d'accions que es poden fer a l'habitació.
- **Armari:** Menú que mostra els complements que té el jugador per vestir la mascota.
- **Botiga:** Menú que mostra els complements que pot comprar el jugador per vestir la mascota.
- **HUD**: Informació que es mostra per pantalla al jugador comentada a l'apartat 5.6.1.
- **Llum (1):** Menú que conté el panell per posar a dormir a la mascota.
- **Llum (2):** Objecte que s'activa per posar a dormir a la mascota.
- **StockComplements:** Objecte que guarda la informació de tots els complements que hi ha disponibles.
- **RoomState:** Objecte que controla i gestiona l'estat de l'habitació.
- **Manta:** Objecte que cobreix el llit de la mascota quan dorm.
- **ArmariInventory:** Objecte que gestiona els complements que té el jugador per posar-li a la mascota
- **LanguageUpdate**: Gestor de textos per canviar d'idioma (s'aprofundirà als següents apartats).

6.2.6.2. Funcionalitats

Tal i com s'ha dit a la introducció d'aquest apartat, en aquest espai es poden realitzar tres funcionalitats diferents: comprar complements, vestir a la mascota i posar-la a dormir. A continuació s'explicarà la implementació d'aquestes tres mecàniques.

6.2.6.2.1. Comprar complements

Pel que fa als complements de la mascota, es va decidir dividir-los en diferents zones de la mascota on es poden posar, i es va optar per les zones del cap, els ulls, el cos i l'esquena. També es va decidir que aquests complements, a part de tenir un nom i un preu a la botiga, tindrien un nivell al qual el jugador hauria d'arribar per poder-los comprar. D'aquesta manera, el jugador tindria una motivació extra per jugar i arribar a un nivell en concret, per acabar podent tenir cada un dels complements.

Per la compra de complements s'ha creat un *Prefab* anomenat *CategoriaComplement* que utilitza la mateixa estructura que el *Prefab CategoriaMenjar*, el qual s'ha analitzat a l'Apartat 6.2.4.2.1. Es tracta d'un objecte que té associat un *script* anomenat *generadorComplements.cs* que, segons quina categoria de complements s'esculli, mostra uns complements o uns altres actualitzant una sèrie de botons (Veure Figures 116 i 117).

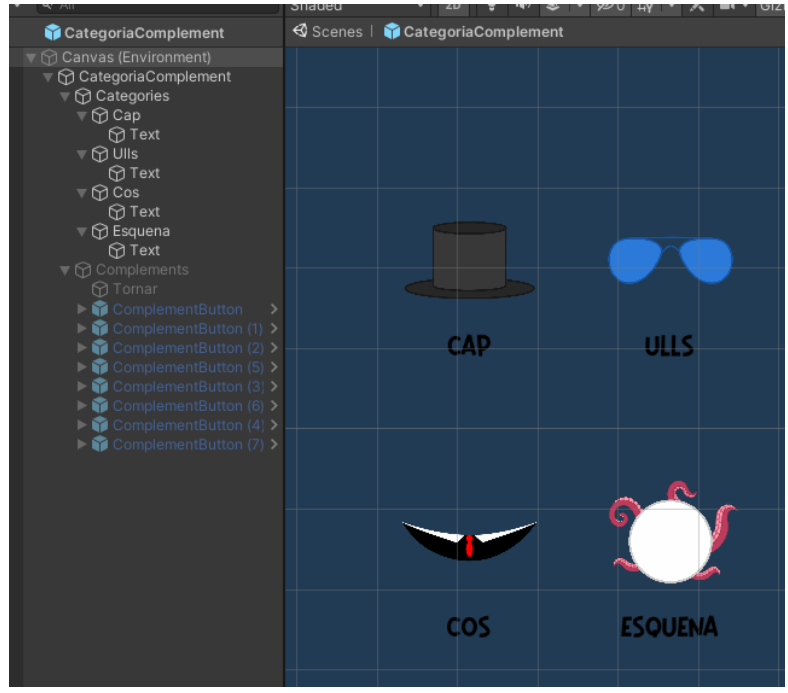


Figura 116: Botons de les categories de complements del *Prefab CategoriaComplements*

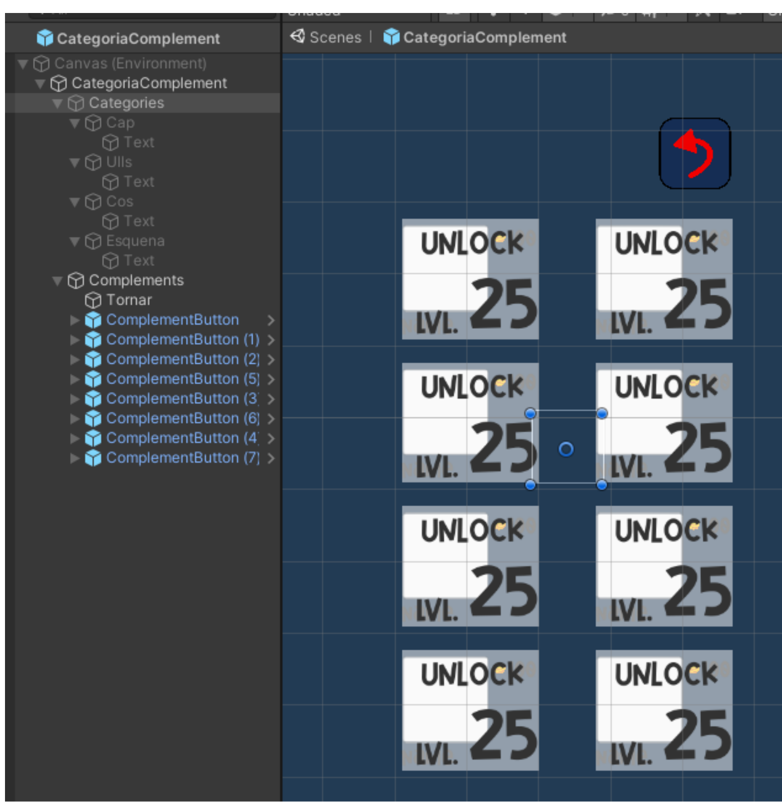


Figura 117: Botons dels diferents complements del *Prefab CategoriaComplements*

D’aquesta manera, l’*script generadorComplements.cs* té una variable que és un *array* de botons. Aquests botons són els 6 que es veuen a la Figura 116, i té una funció anomenada *updButtons()* que executa cada botó quan és premut. Segons el botó que sigui, aquesta funció actualitza l’aspecte dels altres 8 botons que es veuen a la Figura 117. Aquest mètode d’implementació permet anar afegint categories de complements sense haver d’afegir botons de complements per a cada una d’aquestes categories, sinó que hi ha sempre uns 8 botons que s’actualitzen segons la categoria corresponent.

Abans d’explicar com funciona la funció *updButtons()*, cal que s’expliqui com estan organitzats els diferents complements. Així doncs, per gestionar tots els complements del joc es va crear un *script* anomenat *complementsStock.cs*. En aquest *script* se li va posar un *Singleton*, d’aquesta manera només hi ha una instància en tot el joc. També s’hi va crear una classe anomenada *complement*, que conté tota la informació de cada complement i funcions per accedir a aquesta informació. Els atributs d’aquest objecte són el nom identificador del complement, el preu, la categoria a la que pertany, el nivell en el qual es desbloqueja i el nom del complement en 3 idiomes.

Finalment també es van crear dues variables per emmagatzemar tots els complements del joc: un diccionari anomenat *dComplements*, que té com a clau el nom de la categoria i com a valor la llista de tots els complements d’aquella categoria; i un altre diccionari anomenat *categoryComplement*, que té com a clau el nom identificador del complement i com a valor el nom de la categoria a la que pertany (Veure Figura 118).

```
50 referencias
public class complement
{
    string nom, categoria, nomCAT, nomCAST, nomENG;
    int valor, lvlBlock;
```

```
Dictionary<string, List<complement>> dComplements = new Dictionary<string, List<complement>>();

Dictionary<string, string> categoryComplement = new Dictionary<string, string>();
```

Figura 118: Classe *complement* i variables de l’*script complementsStock.cs*

Amb aquesta estructura, quan l'*script complementsStock.cs* s'executa per primera i única vegada, a la funció *Start()* corresponent es creen els diferents complements amb una funció *init()* que s'utlitza com a constructor de la classe *complement*, s'emmagatzemen al primer diccionari segons cada categoria i s'omple el segon diccionari amb els noms de cada complement i la categoria corresponent (Veure Figures 119 i 120).

```
//CAP
List<complement> lCap = new List<complement>();

complement cap = new complement();
cap.init("cap", 10, "cap", "GORRA", "GORRA", "CAP");
lCap.Add(cap);

complement crest = new complement();
crest.init("crest", 10, "cap", "CRESTA", "CRESTA", "CREST");
lCap.Add(crest);

complement hat = new complement();
hat.init("hat", 10, "cap", "BARRET", "SOBRERO", "HAT");
lCap.Add(hat);

complement reindeerEars = new complement();
reindeerEars.init("reindeer ears", 10, "cap", "ORELLES DE REN", "OREJAS DE RENO", "REINDEER EARS");
lCap.Add(reindeerEars);

complement woolCap = new complement();
woolCap.init("wool cap", 10, "cap", "GORRA DE LLANA", "GORRO DE LANA", "WOOL CAP");
lCap.Add(woolCap);

complement beret = new complement();
beret.init("beret", 10, "cap", "BOINA", "BOINA", "BERET");
lCap.Add(beret);

complement ponyTail = new complement();
ponyTail.init("pony tail", 10, "cap", "CUA DE CAVALL", "COLA DE CABALLO", "PONY TAIL");
lCap.Add(ponyTail);

complement witchHat = new complement();
witchHat.init("witch hat", 10, "cap", "BARRET DE BRUIXA", "SOMBRERO DE BRUJA", "WITCH HAT");
lCap.Add(witchHat);

dComplements.Add("cap", lCap);
```

Figura 119: Exemple de creació de complements d'una categoria

```
//EMPLENAR CATEGORIES
foreach (KeyValuePair<string, List<complement>> pair in dComplements)
{
    foreach (complement c in pair.Value)
    {
        categoryComplement.Add(c.getNom(), pair.Key);
    }
}
```

Figura 120: Bucle *for* per omplir el diccionari amb cada complement i la categoria corresponent

També cal comentar que per utilitzar aquesta mecànica, tal i com està implementada, cal desar les imatges de cada complement en la carpeta corresponent, ja que cada una de les carpetes fa referència a una categoria de complements i els *scripts* encarregats de gestionar cada complement accedeixen a aquestes carpetes, segons el complement que s'hagi de mostrar, comprar o posar a la mascota (Veure Figura 121).

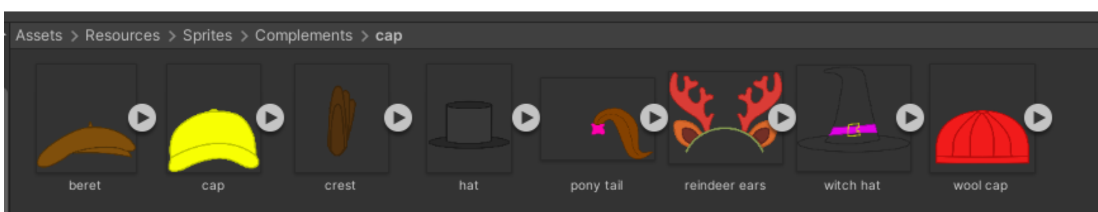


Figura 121: Carpeta d'imatges dels complements de la categoria de complements del cap

D'aquesta manera, tornant a la explicació de com funciona el *Prefab CategoriaComplement*, cada vegada que es prem un botó d'una categoria de complements, aquest executa la funció *updButtons()*. Aquesta funció, segons la categoria que faci referència el botó que s'ha premut, accedeix a la carpeta corresponent d'aquella categoria, i per a cada un dels botons dels complements, actualitza la informació. Per exemple, si es prem el botó de la categoria de complements del cap, la funció fa que els 8 botons de complements facin referència a als 8 complements del cap disponibles del joc, mostrant la imatge i la informació corresponent. Per mostrar la imatge ja s'ha comentat que s'accedeix a la carpeta que toca. D'altra banda, per actualitzar la informació (que és el nom del complement, el valor o preu i el nivell en el que es desbloqueja) crida a la instància del joc que té l'*script complementStock.cs* associat i s'obtenen totes les dades de cada complement, les quals estan emmagatzemades als diccionaris que s'han esmentat anteriorment (Veure Figura 122).

```
0 referencias
public void updButtons(string s)
{

    categoria = s;
    string carpetaAssets = Application.dataPath;
    string carpetaMenjar = carpetaAssets + "/Resources/Sprites/Complements/" + s;

    DirectoryInfo dir = new DirectoryInfo(carpetaMenjar);
    FileInfo[] info = dir.GetFiles("*.png");

    for (int i = 0; i < buttons.Length; i++)
    {
        if (categoria == "ulls" || categoria == "cos")
        {
            buttons[i].transform.localScale = new Vector3(buttons[i].transform.localScale.x,
                buttons[i].transform.localScale.x/2, buttons[i].transform.localScale.z);
            for (int j = 0; j <4; j++)
            {
                buttons[i].transform.GetChild(j).transform.localScale = new Vector3(buttons[i].transform.GetChild(j).transform.localScale.x,
                    buttons[i].transform.GetChild(j).transform.localScale.x * 2, buttons[i].transform.GetChild(j).transform.localScale.z);
            }
        }
        else
        {
            buttons[i].transform.localScale = new Vector3(buttons[i].transform.localScale.x,
                buttons[i].transform.localScale.x, buttons[i].transform.localScale.z);
            for (int j = 0; j < 4; j++)
            {
                buttons[i].transform.GetChild(j).transform.localScale = new Vector3(buttons[i].transform.GetChild(j).transform.localScale.x,
                    buttons[i].transform.GetChild(j).transform.localScale.x, buttons[i].transform.GetChild(j).transform.localScale.z);
            }
        }

        int posPunt = info[i].Name.IndexOf(".");
        string nom = info[i].Name.Substring(0, posPunt);
        buttons[i].image.sprite = Resources.Load<Sprite>("Sprites/Complements/Botons/" + s + "/" + nom);

        int valor = 0;
        if (categoria == "esquenaVisual") valor = complementsStock.instance.getValor("esquena", nom);
        else { valor = complementsStock.instance.getValor(categoria, nom); }

        string nomPerMostrar;
        if (categoria == "esquenaVisual") nomPerMostrar = complementsStock.instance.getNomM("esquena", nom, languageController.instance.lanToNum());
        else { nomPerMostrar = complementsStock.instance.getNomM(categoria, nom, languageController.instance.lanToNum());  }

        buttons[i].transform.GetChild(0).GetComponent<Text>().text = nom;
        buttons[i].transform.GetChild(1).GetComponent<Text>().text = valor.ToString();
    }
}
```

Figura 122: Funció *updButtons()* de l'*script generadorComplements.cs*

Cal fer un comentari en aquest *script* i és que alguns botons es tracten de formes diferents segons la categoria. Per començar, varia l'escala dels botons en el moment de mostrar-los, i això és perquè els complements dels ulls i del cos són més allargats que els de l'esquena i els del cap, i com que al final es mostren als mateixos 8 botons, si no es canvia dinàmicament la mida d'aquests botons, alguns dels complements es veurien distorsionats. D'altra banda, es pot veure que en el moment d'accedir a la carpeta dels complements de l'esquena es fa servir un nom diferent ("esquenaVisual"). Això és perquè amb els altres 3 tipus de complements es fa servir la mateixa imatge per col·locar a la mascota que per mostrar-la a la botiga, mentre que els complements de l'esquena utilitzen imatges diferents (Veure Figura 123).

Figura 123: Exemple d'imatges de complements de l'esquena. A l'esquerra la imatge que es col·loca a la mascota, i a la dreta la imatge que es mostra a la botiga

Un cop mostrats els complements corresponents a cada categoria, s’ha d’implementar la funcionalitat de poder comprar aquests complements. A la funció *Update()* de l’*script generadorComplements.cs* es comprova que els complements de la categoria que s’ha seleccionat es puguin comprar segons el nivell del jugador i segons l’inventari, ja que si el jugador ja té aquell complement no se li ha de poder permetre tornar-lo a comprar (Veure Figura 124).

```
// Update is called once per frame
Mensaje de Unity | 0 referencias
void Update()
{
    if (categoria != "-")
    {
        string subCategoria;
        if (categoria == "esquenaVisual") { subCategoria = "esquena"; }
        else { subCategoria = categoria; }

        string carpetaAssets = Application.dataPath;
        string carpetaMenjar = carpetaAssets + "/Resources/Sprites/Complements/" + subCategoria;

        DirectoryInfo dir = new DirectoryInfo(carpetaMenjar);
        FileInfo[] info = dir.GetFiles("*.png");

        for (int i = 0; i < buttons.Length; i++)
        {

            int posPunt = info[i].Name.IndexOf(".");
            string nom = info[i].Name.Substring(0, posPunt);

            buttons[i].enabled = !stockArmari.instance.teComplement(subCategoria, nom);
            buttons[i].transform.GetChild(3).gameObject.SetActive(stockArmari.instance.teComplement(subCategoria, nom));

            if (complementsStock.instance.getLvlBlock(subCategoria, nom) > lvlController.instance.getLvl())
            {
                buttons[i].transform.GetChild(4).gameObject.SetActive(true);
                buttons[i].transform.GetChild(4).transform.GetChild(0).GetComponent<Text>().text = complementsStock.instance.getLvlBlock(subCategoria, nom).ToString();
                buttons[i].enabled = false;
            }
            else
            {
                buttons[i].transform.GetChild(4).gameObject.SetActive(false);
            }
        }
    }

}
```

Figura 124: Funció *Update()* de l’*script generadorComplements.cs*

Seguidament, a cada un dels botons que fan referència a cada complement se li ha associat com a tasca, quan es prem, una nova funció de l’*script generadorComplements.cs* anomenada *comprarComplement()* (Veure Figures 125 i 126). Aquesta funció accedeix a la informació del complement a través d’un nombre identificador, que també se li afegeix a cada botó, i comprova que aquell complement es pugui comprar en funció de les monedes que tingui el jugador. Si això es compleix, li treu les monedes corresponents i compra aquest complement, afegint-lo a un objecte que té associat un *script* anomenat *stockArmari.cs*, el qual s’encarrega d’emmagatzemar tots els complements que té el jugador. A l’explicació de la següent funcionalitat s’analitza l’estructura d’aquest *script*.

```
0 referencias
public void comprarComplement(int idComp)
{
    if (categoria == "esquenaVisual") { categoria = "esquena"; }

    string carpetaAssets = Application.dataPath;
    string carpetaComp= carpetaAssets + "/Resources/Sprites/Complements/" + categoria;

    DirectoryInfo dir = new DirectoryInfo(carpetaComp);
    FileInfo[] info = dir.GetFiles("*.png");

    int posPunt = info[idComp].Name.IndexOf(".");
    string nom = info[idComp].Name.Substring(0, posPunt);

    if (controladorMonedes.instance.monedesSuficients(complementsStock.instance.getValor(categoria, nom)))
    {

        controladorMonedes.instance.treureMonedes(complementsStock.instance.getValor(categoria, nom));
        if (categoria == "cap")
        {
            stockArmari.instance.afegirComplementCap(nom);
        }

        else if(categoria == "ulls")
        {
            stockArmari.instance.afegirComplementUlls(nom);
        }

        else if (categoria == "cos")
        {
            stockArmari.instance.afegirComplementCos(nom);
        }

        else
        {
            stockArmari.instance.afegirComplementEsquena(nom);
        }
    }

}
```

Figura 125: Funció *comprarComplement()* de l'*script generadorComplements.cs*

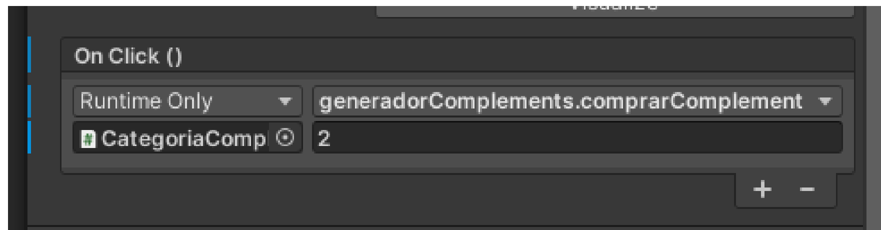


Figura 126: Exemple de tasca associada al botó per comprar un complement

6.2.6.2.2. Vestir a la mascota

Per la mecànica de vestir a la mascota es va crear un *script* anomenat *stockArmari.cs* que gestiona els complement que té el jugador, tal i com s'ha dit a l'apartat anterior, i un altre *script* anomenat *closetInventory.cs* que és el que mostra els complements que té el jugador de cada categoria. D'aquesta manera, l'estructura d'aquests *scripts* és la següent: L'*script stockArmari.cs* té un *Singleton* que permet instanciar l'objecte amb aquest *script* una sola vegada. També té quatre llistes amb els noms dels complements que té el jugador de cada categoria (Veure Figura 127).

```
private List<string> compHeadNames = new List<string>();
private List<string> compEyesNames = new List<string>();
private List<string> compBodyNames = new List<string>();
private List<string> compBackNames = new List<string>();
```

Figura 127: Variables de l'*script stockArmari.cs*

D'altra banda, l'*script closetInventory.cs* també té un *Singleton* que permet instanciar l'objecte amb aquest *script* una sola vegada. També té com a variable la imatge que s'ha de mostrar quan el jugador no té cap complement d'una categoria, els botons que es premen quan es vol vestir a la mascota amb un complement, i quatre nombres enters privats que serveixen per gestionar el nombre de complements que s'han de mostrar per a cada categoria (Veure Figura 128).

```
public Sprite noComp;

public Button currentCompHead, currentCompEyes, currentCompBody, currentCompBack;

private int idCompHead, idCompEyes, idCompBody, idCompBack;
```

Figura 128: Variables de l'*script closetInventory.cs*

A més, també s'ha utilitzat l'*script buttonClicked.cs*, el qual s'ha explicat a l'apartat del laboratori, perquè, de la manera en què està dissenyat el joc, els complements són una altra part que forma part de la mascota, i aquest *script* està fet expressament per modificar aquestes parts.

Així doncs, hi ha 4 botons que quan es premen es vesteix a la mascota amb els complements corresponents. Aquests botons són els que tenen associats l'*script buttonClicked.cs*, amb les variables de *Forma* i *Complement* seleccionades, igual que el complement que han de canviar a l'*array* de parts (Veure Figures 129 i 130).

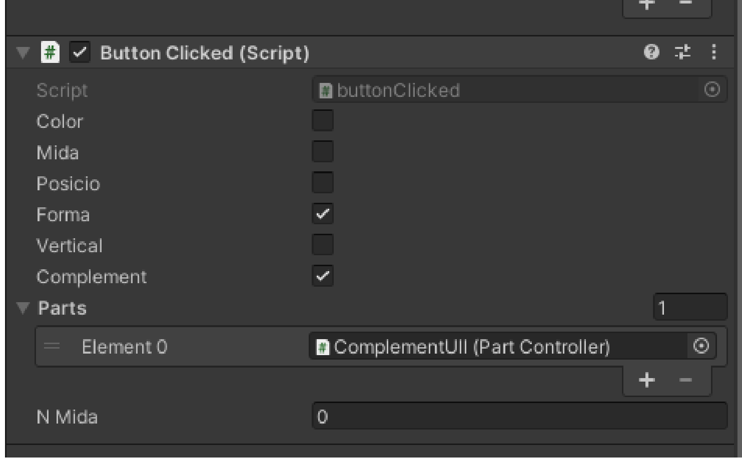


Figura 129: Exemple d'*script* que tenen associats els botons dels complements

```
if (forma)
{
    if (complement)
    {
        mascotaAtributs.instance.canviForma(parts[i], closetInventory.instance.getCompImageWithCategory(parts[i].nom));
    }
```

Figura 130: Part del codi de l'*script buttonClicked.cs* que s'executa quan aquestes dues variables estan seleccionades

6.2.6.2.3. Posar a dormir a la mascota

Finalment, l'última funcionalitat que es pot fer en aquest espai, és la de posar a dormir a la mascota. Per implementa aquesta mecànica es va crear un *script* anomenat *roomState.cs*, el qual gestiona i controla l'estat de l'habitació, és a dir, si s'ha posat a descansar a la mascota o no. Aquest *script* té una sèrie de variables: una variable booleana que comprova si la mascota està descansant o no, una imatge en negre que es col·loca a l'habitació per donar l'efecte de que els llums estan apagats, una imatge d'una manta que es posa sobre el llit quan la mascota dorm, una referència a la mascota per obtenir una sèrie de característiques, imatges dels ulls i de la boca de la mascota quan dorm i quan no, animacions dels ulls parpellejant i de la mascota respirant quan dorm, una referència als ulls i a la boca de la mascota, un vector que fa referència a la mida de la mascota quan no dorm i dues constants que controlen l'augment d'energia de la mascota quan està dormint (Veure Figura 131).

```
private bool lightsOn;
public GameObject blackImage, mascota, manta;

private Sprite ullMascota, bocaMascota;
private AnimationClip parpadeig;

public Sprite ullTancat, bocaTancada;

public partController ullE, ullD, boca;

public AnimationClip mascotaDormint;

private Vector3 mascotaScale;

private const float AUGMENT_ENERGIA = 2;
private const float TEMPS_AUGMENT_ENERGIA = 1;
```

Figura 131: Variables i constants de l'*script roomState.cs*

D'aquesta manera, s'ha creat una funció que fa totes les accions corresponents quan es posar a dormir a la mascota o quan es desperta (Veure Figura 132). Les accions són les següents:

- Activar o desactivar les imatges de la manta i de la transparència negra.
- Canviar la forma dels ulls i de la boca.
- Iniciar o parar l'animació de respirar mentre la mascota dorm.
- Augmentar o deixar d'augmentar l'energia de la mascota

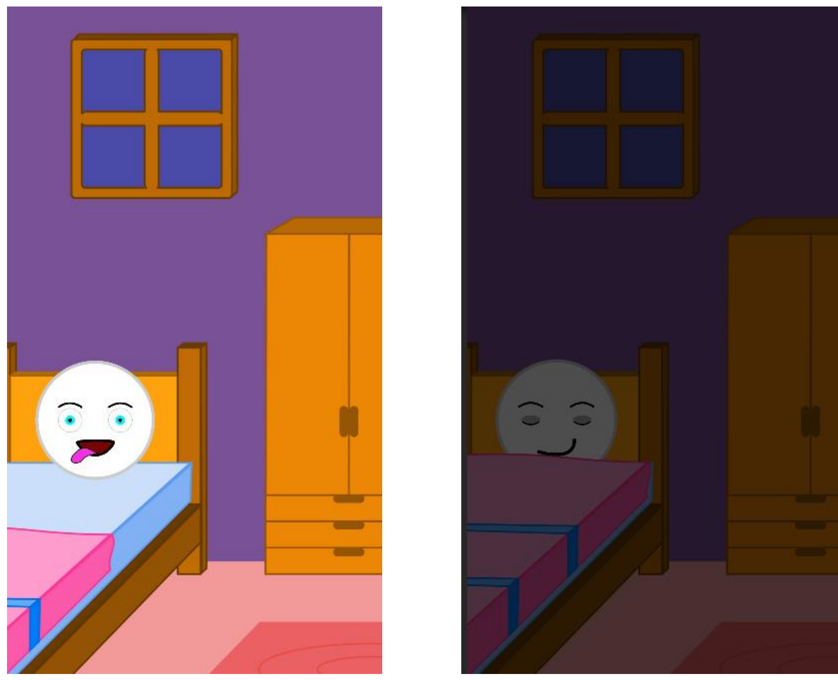


Figura 132: Habitació quan la mascota està dormint i quan no

6.2.6.3. Distribució de les funcionalitats

Un cop s'ha vist com s'han implementat les tres funcionalitats que es troben a l'habitació, comprar complements, vestir a la mascota i posar-la a dormir, seguidament s'analitzarà com s'han distribuït aquestes funcionalitats a l'espai de l'habitació.

Primerament s'ha creat un panell a la part dreta de la pantalla amb dos botons, una bossa per poder comprar els complements i un armari per accedir als complements que té el jugador. També s'ha creat un panell a la part esquerra de la pantalla amb un botó, un llum per posar a dormir a la mascota o despertar-la (Veure Figura 133). Els 2 botons del panell de la dreta obren panells per realitzar les funcionalitats que s'han comentat. A continuació s'analitzaran aquests 2 panells.

Figura 133: Panells amb els botons de les funcionalitats de l'habitació

6.2.6.3.1. Panell de comprar complements

Quan es prem la icona de la bossa, s'obre un panell amb el *Prefab CategoriaComplement* que s'ha analitzat en apartats anteriors. D'aquesta manera, es veuen 6 botons amb totes les categories de complements, i quan es prem un d'aquests botons es veuen els 8 complements d'aquella categoria corresponent (Veure Figura 134). També hi ha botons per tornar a l'espai de l'habitació i per tornar al panell de les categories.

Figura 134: Panells per comprar els complements de la mascota

6.2.6.3.2. Panell de vestir a la mascota

Quan es prem la icona de l'armari, s'obre un panell on es veuen els complements de cada categoria que té el jugador disponibles. Si el jugador no té cap complement d'alguna categoria, es veu una icona que ho representa. Si es prem qualsevol dels complements, automàticament se li col·loca a la mascota. També es pot escollir la opció de no posar-li cap complement (Veure Figura 135).

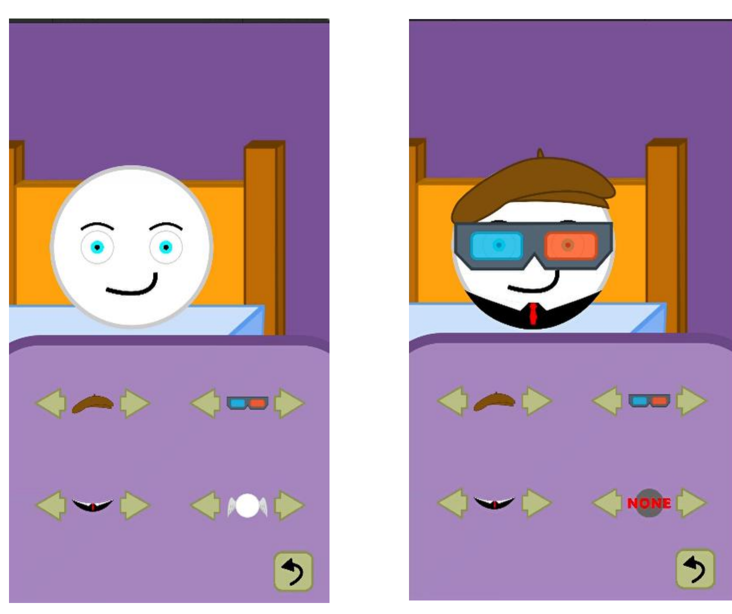


Figura 135: Panell per vestir a la mascota

6.2.6.4. Animacions de panells i menús

De la mateixa manera que s'ha fet als espais del laboratori i de la cuina, s'han animats els panells que s'han vist a l'apartat anterior, per fer que el resultat sigui més atractiu estèticament. S'ha utilitzat l'*script panelController.cs* per gestionar aquestes animacions quan es premen els diferents botons dels panells i menús, seguint el mateix procediment que al laboratori. En aquest cas, s'ha tornat a optar per la superposició dels diferents panells, tan al menú de comprar complements com al menú de vestir a la mascota. A aquest segon menú també se li ha afegit un zoom a la mascota per veure de més a prop els complements que se li col·loquen (Veure Figures 136 i 137).

Figura 136: Animació de superposició dels panells per comprar els complements de la mascota

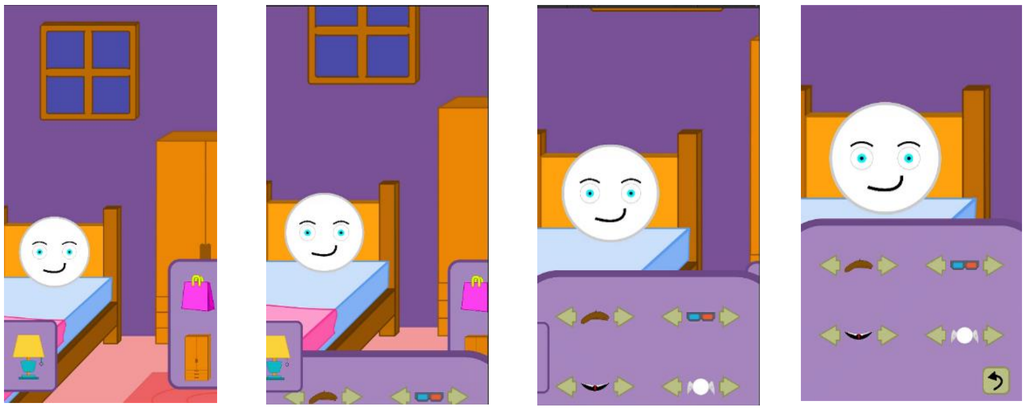


Figura 137: Animació de superposició dels panells i zoom per vestir a la mascota

6.2.7. Espai de minijocs

L'espai on el jugador pot escollir els diferents minijocs és bastant senzill a nivell d'implementació, ja que es tracta d'una escena on només hi ha quatre botons per accedir als quatre minijocs (Veure Figura 138). L'únic que s'ha hagut d'implementar és el fet de que la mascota no pugui jugar si té poca energia o molta gana, a través d'un *script* anomenat *gamesController.cs*, el qual es comunica amb l'*script* que emmagatzema dades de l'estat de la mascota (Veure Figures 139 i 140). Segons si a la mascota li falta energia, si té molta gana o si es compleixen ambdues situacions, es mostra un panell informatiu o un altre quan el jugador intenta accedir a un minijoc.

Figura 138: Espai de minijocs

Figura 139: Diferents panells que poden aparèixer si la mascota no pot jugar

```
0 referencias
public void playGame(string name)
{
    if (mascotaState.instance.canPlay() > 0)
    {
        cantPlay.SetActive(true);
        cantPlay.transform.GetChild(3).gameObject.SetActive(false);
        cantPlay.transform.GetChild(4).gameObject.SetActive(false);
        cantPlay.transform.GetChild(5).gameObject.SetActive(false);
        cantPlay.transform.GetChild(mascotaState.instance.canPlay()).gameObject.SetActive(true);

        string text = PlayerPrefs.GetString("name", "");
        if (PlayerPrefs.GetString("Language") == "cat")
        {
            text += " NO POT JUGAR ARA...";
        }
        else if (PlayerPrefs.GetString("Language") == "cast")
        {
            text += " NO PUEDE JUGAR AHORA...";
        }
        else
        {
            text += " CAN'T PLAY RIGHT NOW...";
        }

        cantPlay.transform.GetChild(2).GetComponent<Text>().text = text;
    }

    else
    {
        SceneManager.LoadScene(name);
    }

}
```

Figura 140: Part del codi que comprova si la mascota pot jugar o no, i gestiona els panells informatius que apareixen

6.3. Minijocs

Els minijocs també tenen molta importància a *Otis*, ja que és l'única manera d'augmentar la felicitat de la mascota i possiblement sigui la part més entretinguda pel jugador, juntament amb la part de modificació de la mascota. En aquest apartat s'explicarà la implementació de cada un dels quatre minijocs que s'han desenvolupat pel projecte.

6.3.1. Tir al plat

El Tir al plat va ser el primer minijoc que es va desenvolupar, i consisteix en que van sortint mascotes pels dos costats de la pantalla i el jugador ha de tocar el màxim de mascotes possibles. Hi ha un temps restant representat per una barra, el qual va disminuint mentre avança el joc. Si aquest temps arriba a 0, el joc es dona per acabat. El temps disminueix encara més si el jugador deixa passar una mascota sense tocar-la i augmenta si cada vegada que toca una mascota.

Per desenvolupar aquest minijoc es va crear un *script* anomenat *platsGenerator.cs*. Aquest actua com a *spawner* dels plats, que es representen amb la imatge de la mascota de cada jugador, i té un seguit de variables per controlar com es llancen aquests plats. Les variables són l'objecte plat que s'ha d'instanciar cada vegada que se'n llança un, el nombre de monedes que se li dona al jugador cada vegada que toca un plat i un conjunt de variables *float* que serveixen per establir màxims i mínims de segons entre que es llança un plat i se'n llança un altre, per controlar la velocitat dels plats i com aquesta va augmentant, per posicionar bé els plats cada vegada que s'instancien, etc. Es poden veure aquestes variables a la Figura 141.

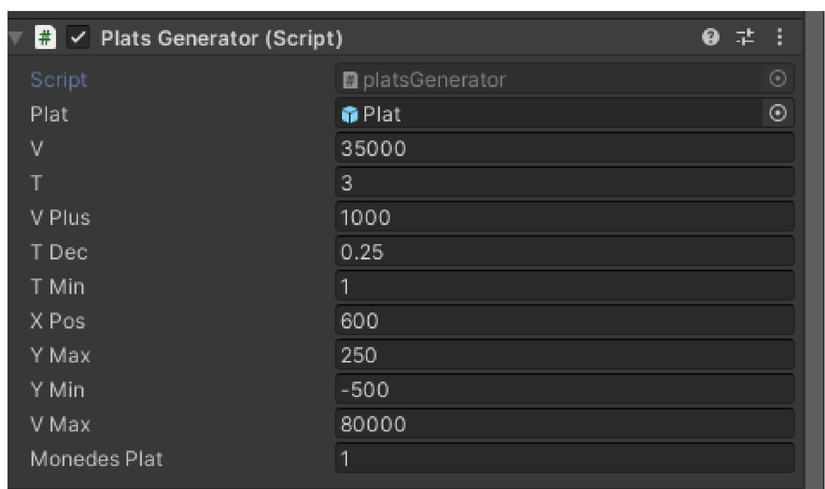


Figura 140: *Script* que té associat l'objecte *spawner* de plats

- **V:** Velocitat inicial dels plats
- **T:** Temps inicial entre la sortida d'un plat i un altre
- **V Plus:** Valor que augmenta la velocitat cada vegada que surt un plat
- **T Dec:** Valor que disminueix el temps entre la sortida d'un plat i un altre
- **T Min:** Temps mínim que hi pot haver entre la sortida d'un plat i d'un altre
- **X Pos:** Valor que pot obtenir, tan en positiu com en negatiu, l'eix x de la posició del plat quan surt
- **Y Max:** Valor màxim que pot obtenir l'eix y de la posició del plat quan surt
- **Y Min:** Valor mínim que pot obtenir l'eix y de la posició del plat quan surt
- **V Max:** Valor màxim que pot obtenir la velocitat del plat quan surt
- **Monedes Plat:** Nombre de monedes que es dona al jugador quan toca un plat.

Per fer els plats es va crear un *Prefab* anomenat *Plat*, al qual se li va crear i associar un *script* anomenat *platController.cs*. El *Prefab* consisteix en un botó al qual se li ha afegit un component *Rigidbody2D* i un *BoxCollider2D*, a més de l'*script* que s'ha esmentat prèviament. El fet de que sigui un botó és perquè és una forma senzilla de detectar quan el jugador toca el plat. A aquest botó se li va afegir també el *Prefab* de la mascota, al qual no se li va haver de fer cap modificació pel minijoc i d'aquesta manera poder veure el plat representat per la mascota del jugador (Veure Figura 141). A l'*script* del plat s'hi va fer una funció *init()* la qual a través d'una velocitat i una posició donades, situa el plat allà on toca i el llança, accedint al component *Rigidbody2D* i creant aquest moviment amb la funció *addForce()* que ja està implementada en el mateix component (Veure Figura 142). El component *BoxCollider2D* s'utilitza per comprovar quan el plat surt de la pantalla i d'aquesta manera eliminar-lo i penalitzar al jugador.

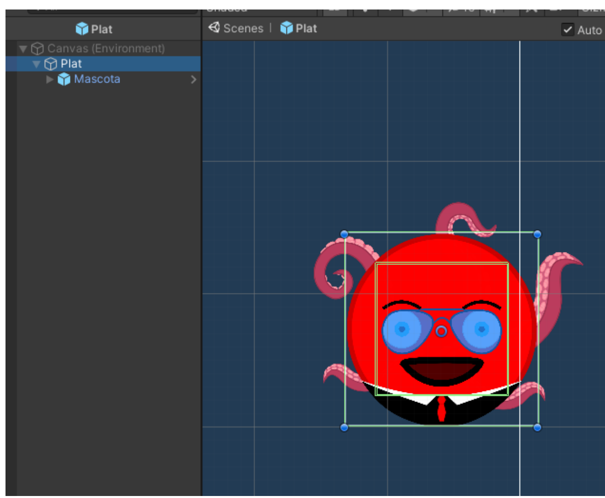


Figura 141: *Prefab Plat*

```
1 referencia
public void init(Vector2 pos, float fH, float fV)
{
    bool left = pos.x < 0;
    transform.localPosition = pos;

    if (left)
    {
        GetComponent<Rigidbody2D>().AddForce(new Vector2(fH, fV));
        degrees = -300;
    }
    else
    {
        GetComponent<Rigidbody2D>().AddForce(new Vector2(-fH, fV));
        degrees = 300;
    }

}
```

Figura 142: Funció *init()* de l'*script platController.cs*

D'aquesta manera, l'*script platsGenerator.cs* és l'encarregat d'anar instanciant *Prefabs Plat* i cridant la seva funció *init()* fins que el temps restant arribi a zero. Al minijoc se li va afegir també un fons i el resultat final és el de la següent figura.

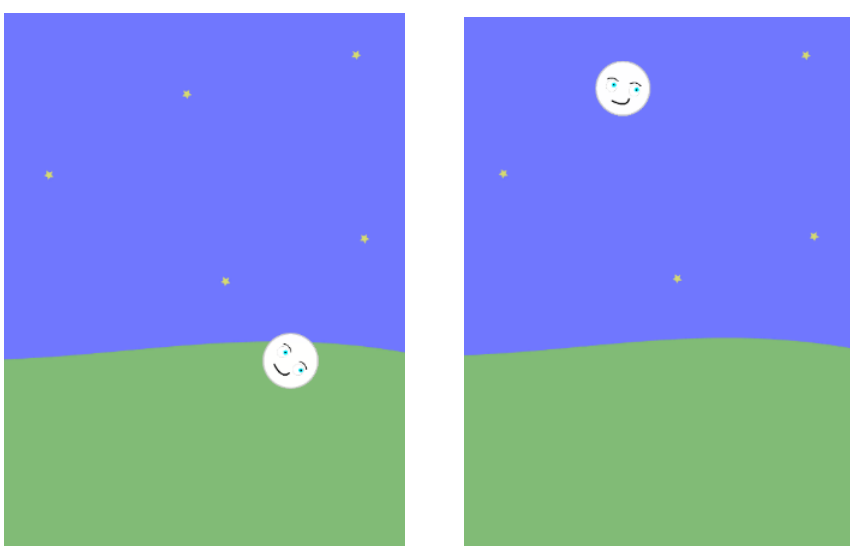


Figura 143: Resultat final del minijoc Tir al Plat

6.3.2. Escalada

El joc de l'escalada consisteix en què el jugador ha de demostrar ser àgil visualment i fer que la mascota escali el més amunt possible. Per fer-ho, van sortint dos arbres amb fustes, dels quals un d'ells té la fusta trencada i l'altre no, així que el jugador ha d'anar prement els arbres que tenen la fusta en bones condicions, perquè si s'equivoca finalitza la partida. A més ho ha de fer de la manera més ràpida possible perquè també hi ha un temps restant que va disminuint i que si arriba a zero també s'acaba la partida. De totes maneres, aquest temps augmenta també quan el jugador prem l'arbre corresponent.

D'aquesta manera, per implementar aquest minijoc, primerament es va crear l'objecte *spawner* d'arbres, que seria l'encarregat d'anar generant aleatòriament els arbres amb les fustes corresponents. Per fer-ho, es va crear un *script* anomenat *treesGenerator.cs* i se li va associar a l'objecte *spawner*. A aquest objecte també se li van afegir dos botons invisibles, encarregats de detectar quan el jugador toca la part esquerra de la pantalla o la dreta, és a dir, si toca un arbre o un altre. Aquests botons tenen associat com a tasca una funció de l'*script treesGenerator* anomenada *seguentEsq()*, la qual a través d'un valor booleà comprova si el jugador ha premut l'arbre de l'esquerra o el de la dreta i si era l'arbre correcte. Si és l'arbre correcte, es crida una altra funció encarregada de generar una nova parella d'arbres, en cas contrari es dona la partida per finalitzada. Per realitzar aquestes comprovacions, es va guardant dins d'una llista de caràcters els arbres correctes, es guarda una lletra "L" en cas de que l'arbre bo sigui el de l'esquerra i altrament es guarda un lletra "R". Cada vegada que passa una ronda de parelles d'arbres es modifica la llista amb els valors corresponents, d'aquesta manera sempre s'ha de mirar la mateixa posició de la llista per saber quin arbre és el correcte (Veure Figures 144 i 145).

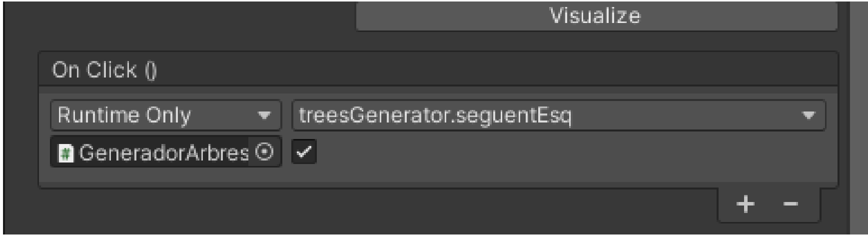


Figura 144: Tasca dels botons que comproven si el jugador ha tocat la pantalla i quina part ha tocat

```
0 referencias
public void seguentEsq(bool esq)
{
    if (clickable)
    {
        if (!started)
        {
            started = true;
            sliderController.instance.startGame();
        }

        bool condicio = ((esq && seq[2] == 'L') || (!esq && seq[2] == 'R'));

        if (condicio)
        {
            player.move(esq);
            seguentSequencia();
        }
        else
        {
            clickable = false;
            sliderController.instance.stopGame();
            end = true;
            f.final();
        }
    }

}
```

Figura 145: Funció de l'*script treesGenerator.cs* que comprova si s'ha premut l'arbre correcte

Amb l'*spawner* d'arbre generat, es va crear un *Prefab* anomenat *TreeImage* (Veure Figura 146) que representa cada arbre que surt, és a dir, és l'objecte que anirà instanciant l'*spawner*. Aquest consta d'un component *Image* i d'un *script* que es va crear anomenat *treeImage.cs*. Aquest conté un seguit de variables per controlar el moviment quan es prem correctament un arbre, un identificador per saber si es tracta de l'arbre correcte o no i dues funcions, una per inicialitzar les variables i una per efectuar el moviment de l'arbre (Veure Figura 147).

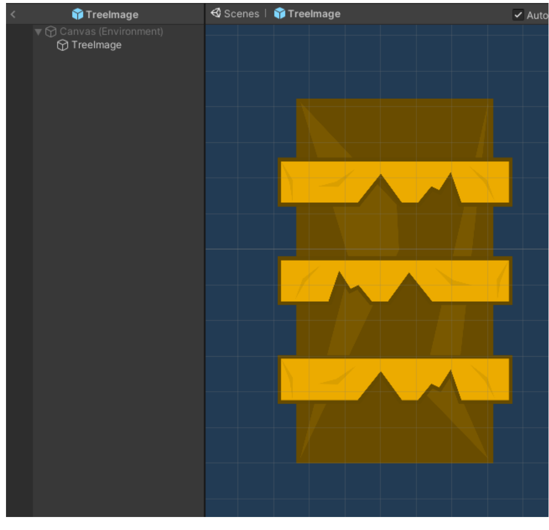


Figura 146: *Prefab TreeImage*

```
4 referencias
public void init(Sprite im, char s, int id, int maxIm)
{
    this.GetComponent<Image>().sprite = im;
    side = s;
    idIm = id;
    nImatges = maxIm;
}

2 referencias
public void seguent(Sprite imG, Sprite imB)
{
    if (idIm == nImatges - 1)
    {
        this.transform.localPosition = new Vector3(this.transform.localPosition.x, 3072+1024, this.transform.localPosition.z);
        if (treesGenerator.instance.getPrimer() == side)
        {
            this.GetComponent<Image>().sprite = imG;
        }
        else
        {
            this.GetComponent<Image>().sprite = imB;
        }
        idIm = 0;
    }
    else
    {
        idIm += 1;

    }
    StartCoroutine(move());
}
```

Figura 147: Funcions de l'*script treesImage.cs*

D'aquesta forma, l'objecte *spawner* d'arbres va generant parelles de *Prefabs TreeImage* i s'encarrega de comprovar que es premin els arbres correctes i seguir generant parelles, però també de finalitzar la partida quan el jugador prem l'arbre incorrecte.

Per últim, s'ha de crear el personatge que va escalant, i per fer-ho es va crearun *GameObject* al qual se li va afegitr el *Prefab* de la mascota del jugador. Seguidament es va crear un *script* anomenat *playerEscController.cs* i se li va associar a aquest *GameObject,* igual que també se li va afegir un component *Animation* i es van crear dues animacions de la mascota saltant d'un costat a l'altre. D'aquesta manera, l'*script* accedeix al component *Animation* i reprodueix una animació o una altra segons la situació del jugador (Veure Figura 148).

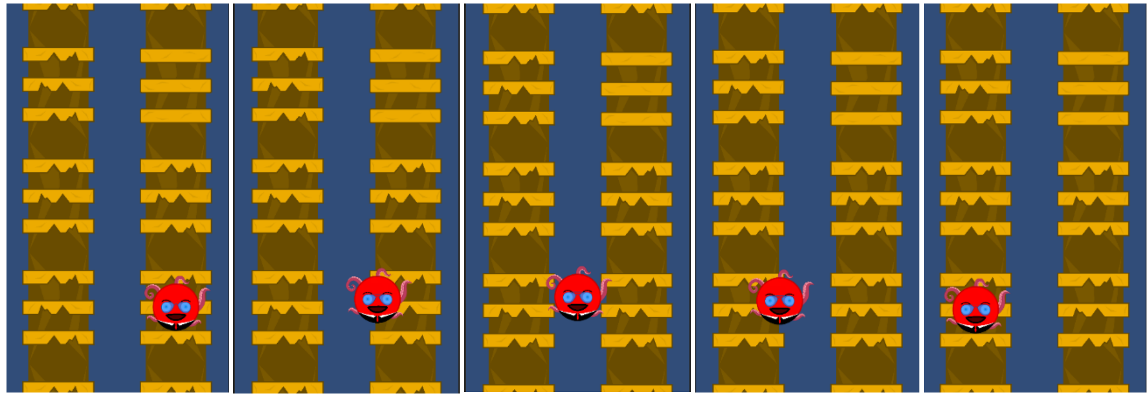


Figura 148: Animació de la mascota canviant d'arbre

Finalment es va afegir també un fons i el resultat final és el de la Figura 149.

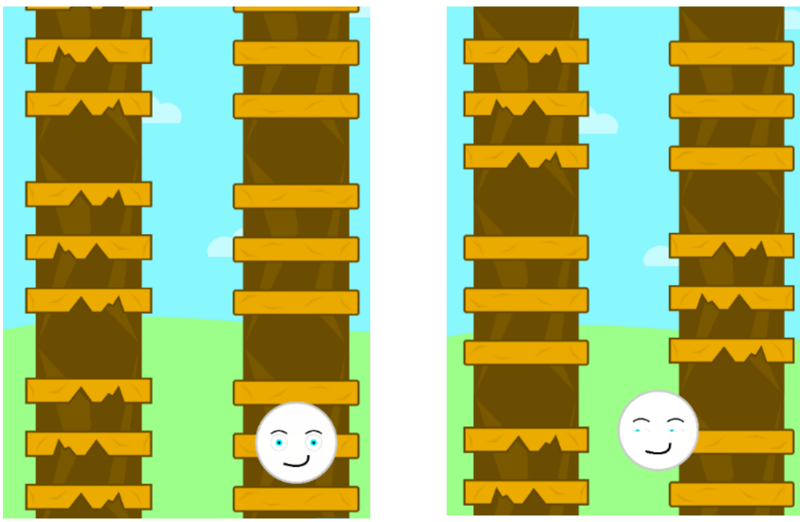


Figura 149: Resultat final del minijoc escalada

6.3.3. Memoritzar

El minijoc de memoritzar consisteix en veure quanta memòria pot arribar a tenir el jugador, ja que consisteix en què es mostra una seqüència de mascotes i el jugador la ha de memoritzar perquè posteriorment la haurà de repetir. La mecànica es basa en que cada vegada les seqüències són més llargues, però no el nombre d'opcions que se li donen al jugador, ja que sinó la corba de dificultat augmenta exponencialment i això influeix negativament en les ganes que pot arribar a tenir el jugador de tornar a jugar. És per això que el joc es divideix en rondes, i cada ronda mostra dues seqüències. Es comencen mostrant 4 imatges i una seqüència de mida 2. Seguidament s'augmenta la mida de la seqüència i es se'n mostra una altra. Un cop repetides aquestes dues seqüències de forma correcta, augmenta el nombre d'imatges però no la mida de la seqüència. Es va repetint aquest procés fins arribar a un màxim de nombre d'imatges, i un cop arribat a aquest nombre, es tornen a mostrar només 4 imatges, però amb la mida de la seqüència molt més llarga. D'aquesta manera la dificultat va augmentant però d'una manera molt més suau.

Per implementar aquest minijoc primerament es va crear un *script* anomenat *sequenceGenerator.cs*, l'encarregat d'anar generant seqüències i anar comprovant que la seqüència que repeteix el jugador sigui correcta. Aquest *script* té un seguit de variables públiques que serveixen per parametritzar totalment les seqüències es mostren. Aquestes variables són les següents:

- Mida de la imatge inicial que es mostra
- Nombre d’imatges que es mostren inicialment
- Mida inicial de la seqüència
- Nombre de seqüències per ronda
- Màxim nombre d’imatges que es mostren

A partir d’aquestes variables, i amb un seguit de funcions es van creant les seqüències i emmagatzemant-les per comprovar que s’estan repetint de forma correcta. De totes maneres, la primera funció que es crea és la que s’utilitza per situar les imatges a l’inici de cada ronda, anomenada *setImages()*, ja que depenent del nombre d’imatges que es mostren varia la mida de les imatges i la posició (Veure Figura 150).

```
1 referencia
public void setImages()
{
    scaleImage = espaiImatges / (imageSize * Mathf.Sqrt(nIm));
    float espaiEntreImatges = espaiSeparacio / (Mathf.Sqrt(nIm) - 1);


    for (int i = 0; i < nIm; i++)
    {
        int fila;
        if (i < Mathf.Sqrt(nIm)) { fila = 0; }
        else { fila = i / (int)Mathf.Sqrt(nIm); }

        int columna;
        if (i < Mathf.Sqrt(nIm)) { columna = i; }
        else { columna = i % (int)Mathf.Sqrt(nIm); }

        float midaImatge = imageSize * scaleImage;

        float posX = -320 + (midaImatge / 2);
        if (columna > 0)
        {
            posX += (midaImatge + espaiEntreImatges) * columna;
        }

        float posY = 320 - (midaImatge / 2);
        if (fila > 0)
        {
            posY -= (midaImatge + espaiEntreImatges) * fila;
        }

        GameObject im = Instantiate(imSeq) as GameObject;
        im.transform.SetParent(gameObject.transform);
        StartCoroutine(mostrarImatge(i, scaleImage, new Vector3(posX, posY, 1)));

    }

}
```

Figura 150: Funció per col·locar les imatges de l’*script sequenceController.cs*

També es va crear un *Prefab* per les imatges que es mostren, anomenat *ImSeq*. Aquest consisteix en un botó al qual se li afegeix el *Prefab* de la mascota del jugador, una imatge que inicialment és transparent, un component *Animation* i un seguit d'animacions depenent de si s'està mostrant la seqüència, si el jugador ha encertat la seqüència o si ha fallat (Veure Figura 151) i un *script* anomenat *imSeqController.cs*. Aquest *script* és l'encarregat de gestionar les animacions de les imatges i posicionant-les de forma correcta, ja que qui comprova si la seqüència és correcta o no és l'*script sequenceGenerator.cs*.

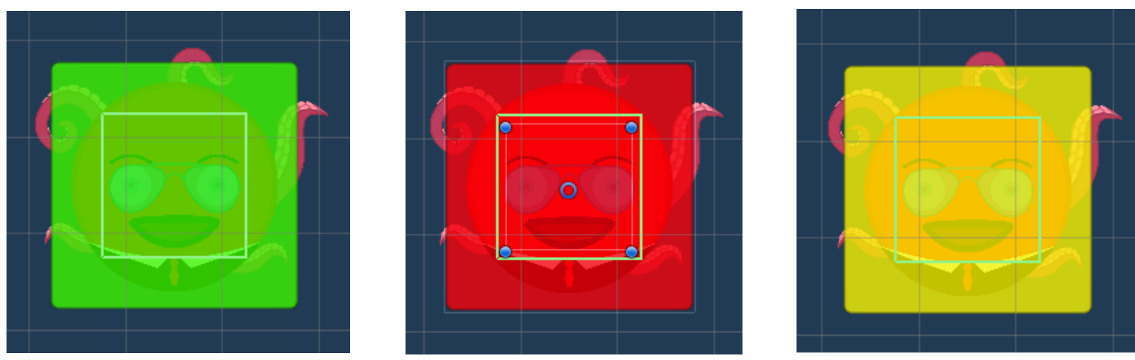


Figura 151: Diferents animacions del *Prefab ImSeq*

D'aquesta manera, cada vegada que s'inicia una ronda, després de col·locar les imatges amb la funció *setImages()* que s'ha comentat anteriorment, es crida la funció *iniciarRonda()* de l'*script sequenceGenerator.cs* (Veure Figura 152).

```
3 referencias
public void iniciarRonda()
{
    gameState = false;
    setImages();
    generarSequencia();
    StartCoroutine(mostrarSequencia());
}
```

Figura 152: Funció *iniciarRonda()* de l'*script sequenceGenerator.cs*

Seguidament, es mostra la seqüència amb una funció anomenada *mostrarSequencia()* (Veure Figura 153).

```
1 referencia
IEnumerator mostrarSequencia()
{
    yield return new WaitForSeconds(1);
    float offset = 0.5f;
    float timeAnim = this.transform.GetChild(seq[0]).GetComponent<imSeqController>().getShowAnimTime();
    for (int i = 0; i < lenSeq; i++)
    {
        this.transform.GetChild(seq[i]).GetComponent<imSeqController>().mostrarSequencia();

        yield return new WaitForSeconds(timeAnim + offset);
    }

    gameState = true;
}
```

Figura 153: Funció *mostrarSequencia()* de l'*script sequenceGenerator.cs*

I després es va comprovant que el jugador vagi repetint la seqüència de forma correcta, a través de la funció *checkImage()*, que està associada a cada *Prefab ImSeq* i es crida cada vegada que el jugador prem una imatge de la seqüència (Veure Figura 154).

```
1 referencia
public void checkImage(int id)
{
    if (seq[nSeqAct] == id)
    {
        this.transform.GetChild(id).GetComponent<imSeqController>().sequenciaCorrecta();

        if (nSeqAct < lenSeq - 1) {
            nSeqAct++;
        }
        else
        {
            nSeqAct = 0;
            StartCoroutine(finalitzarRonda());
        }
    }
    else
    {
        StartCoroutine(finalitzarPartida(id));
    }
}
```

Figura 154: Funció *checkImage()* de l'*script sequenceGenerator.cs*

I per acabar amb la mecànica, quan s'acaba una ronda es comprova si és l'última seqüència de la ronda. En cas de ser-ho es comprova si s'ha arribat al nombre màxim d'imatges per mostrar a cada ronda i, segons aquesta comprovació, s'augmenta el nombre d'imatges o es torna al nombre d'imatges inicial. Tot això es comprova a la funció *finalitzarRonda()* (Veure Figura 155).

```
1 referencia
IEnumerator finalitzarRonda()
{
    pointsController.instance.afegirPunts(1);
    pointsController.instance.afegirMonedes(lenSeq, new Vector2(0, -300), false);

    yield return new WaitForSeconds(1);

    //SEGUENT NIVELL
    if (rAct == rondes)
    {
        StartCoroutine(seguentNivell());
    }

    //MATEIX NIVELL MES SEQUENCIA
    else
    {
        StartCoroutine(seguentRonda());
    }
}
```

Figura 155: Funció *finalitzarRonda()* de l'*script sequenceGenerator.cs*

Finalment es va afegir també un fons i el resultat final és el de la Figura 156.

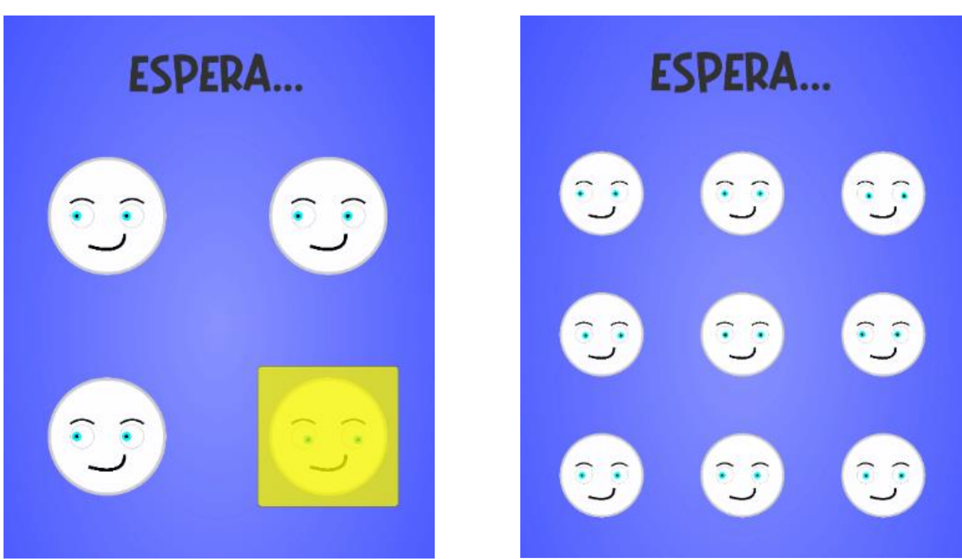


Figura 156: Resultat final del minijoc memoritzar

6.3.4. Plataformes

El joc de plataformes és l'últim minijoc que es va implementar, i consisteix en el clàssic joc de plataformes 2D que no té final, on van sortint diferents plataformes i cada vegada surten de manera més ràpida, fent que la dificultat del minijoc vagi augmentant. Per iniciar la implementació del minijoc, es va crear un objecte que actués com a *spawner* de plataformes, creant d'aquesta manera un *script* anomenat *plataformaGenerator.cs*. Aquest *script* conté diverses variables que fan que aquest *spawner* es pugui parametritzar fàcilment, les quals controlen la velocitat a la que surten les plataformes, la velocitat màxima, l'augment de velocitat entre una i altra, etc.

També es va crear un *Prefab Plataforma*, que tal i com diu el nom, representa cada plataforma que es genera. Com que la idea és que vagin apareixent plataformes de mides diferents, al *Prefab* se li van afegir 3 plataformes i l'*spawner* és l'encarregat de triar quina plataforma surt. També es va haver de crear un jugador, així que es va crear un *GameObject* buit, es va afegir el *Prefab* de la mascota del jugador i dos components: un *Rigidbody2D* i un *BoxCollider2D*. A més, també es va crear i se li va afegir un *script* anomenat *playerPlataformesController.cs*, encarregant-se del moviment del jugador.

Amb aquests 3 elements, per combinar-los i obtenir un bon resultat, es va fer que el jugador pogués fer saltar la seva mascota tocant-la, i si la mantenia aquesta tardava més estona en caure (Veure Figura 157).

```
Mensaje de Unity | 0 referencias
private void Update()
{
    if (viu)
    {
        if (isGrounded == true && Input.GetMouseButtonDown(0))
        {
            isGrounded = false;
            isJumping = true;
            jumpTimeCounter = jumpTime;
            rb.velocity = Vector2.up * jumpForce;

        }

        if (Input.GetMouseButton(0) && isJumping)
        {
            if (jumpTimeCounter > 0)
            {
                rb.velocity = Vector2.up * jumpForce;
                jumpTimeCounter -= Time.deltaTime;
            }
            else
            {
                isJumping = false;
            }
        }

        if (Input.GetMouseButtonUp(0))
        {
            isJumping = false;
        }
    }

    if (transform.position.y < -1600)
    {
        viu = false;
        spawner.finishGame();
        Destroy(this.gameObject);
    }
}
```

Figura 157: Funció *Update()* de l'*script playerPlataformesController.cs*

D'altra banda, pel que fa a l'*spawner* de plataformes, s'ha de controlar que el jugador pugui saltar totes les plataformes, és a dir, que l'espai entre una plataforma i una altra no sigui tan gran que ni saltant el màxim de temps possible el jugador no el pugui passar. Per fer aquest control, l'*spawner* cada vegada que genera una plataforma, crea la següent i calcula segons la velocitat de la plataforma que ha sortit, la mida, la velocitat a la que sortirà la nova plataforma i el temps màxim que pot estar saltant el jugador, quin temps com a molt pot tardar a fer aparèixer la nova plataforma. A partir d'aquí, un cop ja té aquest valor, s'espera un nombre de segons aleatori, el qual no pot superar aquest màxim i genera la nova plataforma, d'aquesta manera s'assegura que el jugador la pot saltar (Veure Figures 158 i 159).

```
1 referencia
IEnumerator spawnPlataformes()
{
    while (!end)
    {
        GameObject p = Instantiate(plataforma) as GameObject;
        p.transform.SetParent(gameObject.transform);

        if (tTardara == 0)
        {
            tArribada = calcTemps();
        }
        else
        {
            tArribada = tTardara;
        }

        p.transform.position = new Vector3(posIniciPlataforma, 0, 0);
        p.GetComponent<plataformaController>().spawnPlataforma(idPlat, speed, tempsEspera);

        idPlat = Random.Range(0, 3);

        if (speed < speedMax) speed += speedOffset;

        tTardara = calcTemps();

        float tWait = tArribada + (Random.Range(tArribada, tempsMaximSalt - tTardara));
        yield return new WaitForSeconds(tWait);

    }
}
```

Figura 158: Funció de l'*script plataformaGenerator.cs* encarregada de generar les plataformes

```
2 referencias
private float calcTemps()
{
    float dist = (posFinals[idPlat] * -1) + posIniciPlataforma;
    float temps = ((dist / speed) * tempsEspera);
    return temps;
}
```

Figura 159: Funció de l'*script plataformaGenerator.cs* encarregada de calcular el temps màxim que pot tardar en generar la nova plataforma

Finalment es va afegir també un fons i el resultat final és el de la Figura 160.

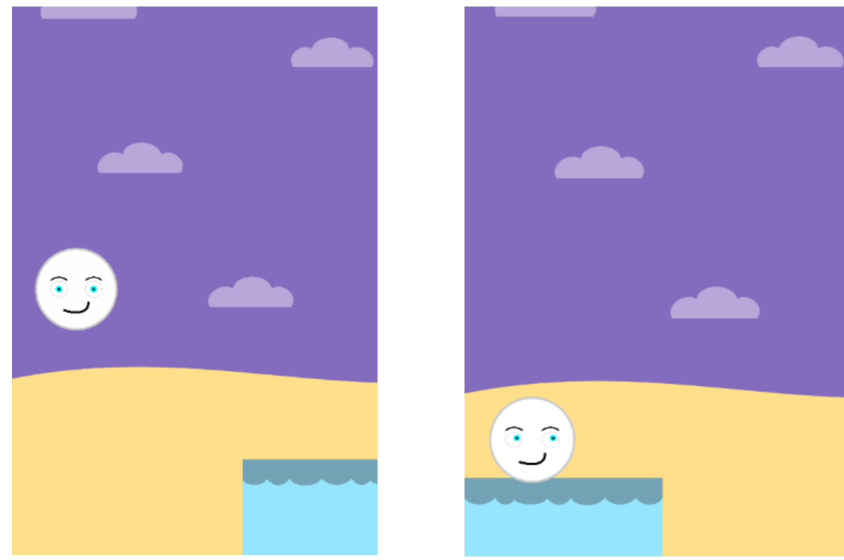


Figura 160: Resultat final del minijoc plataformes

6.4. Canvi d'idioma

Una funcionalitat que es va decidir fer com a extra va ser la de poder canviar l'idioma del joc. En principi no estava pensat que s'anés a fer, però a mida que avançava el projecte s'anava veient que hi hauria alguna manera de fer-ho de forma senzilla. Realment ha estat tot un repte perquè a nivell personal mai ho havia fet amb cap projecte, però el resultat va acabar sent més que satisfactori.

Per poder fer aquests canvis d'idiomes, es va decidir que el joc es traduiria al català, al castellà i a l'anglès. Es va optar per crear un *script* anomenat *languageController.cs* el qual emmagatzema totes les traduccions. Per fer-ho, es va crear una variable que era un diccionari on la clau era la paraula o oració en català i el valor una llista de traduccions, és a dir, una llista on el primer element era la mateixa paraula o oració en castellà i la segona el mateix però en anglès. D'aquesta manera, es va crear un nou *script* anomenat *languageUpdate.cs,* el qual s'instancia a cada escena i té controlats tots els textos que s'han de traduir. També es van haver de canviar tots els textos i posar-los en català, perquè el funcionament d'aquest *script* és el següent: accedeix a l'objecte que té emmagatzemades totes les traduccions i consulta quina és la traducció d'una paraula o oració en català, ja que la clau del diccionari on són totes aquestes traduccions són les paraules o oracions en català (Veure Figura 161). Si aquest procediment es fa amb tots els textos de l'escena, estarà tota la escena traduïda. És per això, que aquest objecte s'ha d'instanciar a cada escena i associar-hi tots els textos que hi ha a la mateixa escena (Veure Figura 162).

```
1 referencia
private void omplirDiccionaris()
{
    diccionari["NOM"] = d("NOMBRE", "NAME");
    diccionari["INTRODUEIX EL NOM"] = d("INTRODUCE EL NOMBRE", "ENTER NAME");
    diccionari["JUGAR"] = d("JUGAR", "PLAY");
    diccionari["FI DEL JOC"] = d("FIN DEL JUEGO", "GAME OVER");
    diccionari["PUNTS:"] = d("PUNTOS:", "POINTS:");
    diccionari["PUNTS MÀXIMS:"] = d("PUNTOS MÁXIMOS:", "HIGHSCORE:");
    diccionari["TORNAR A JUGAR"] = d("VOLVER A JUGAR", "PLAY AGAIN");
    diccionari["ENRERE"] = d("ATRÁS", "BACK");
    diccionari["LABORATORI"] = d("LABORATORIO", "LAB");
    diccionari["CASA"] = d("CASA", "HOME");
    diccionari["MINIJOCS"] = d("MINIJUEGOS", "MINIGAMES");
    diccionari["NIV."] = d("NIV.", "LVL.");
    diccionari["TORNAR A L'HABITACIÓ"] = d("VOLVER A LA HABITACIÓN", "BACK TO ROOM");
    diccionari["TORNAR A LA CUINA"] = d("VOLVER A LA COCINA", "BACK TO KITCHEN");
    diccionari["CAP"] = d("CABEZA", "HEAD");
    diccionari["ULLS"] = d("OJOS", "EYES");
    diccionari["COS"] = d("CUERPO", "BODY");
    diccionari["ESQUENA"] = d("ESPALDA", "BACK");
    diccionari["CUINA"] = d("COCINA", "KITCHEN");
    diccionari["HABITACIÓ"] = d("HABITACIÓN", "ROOM");
    diccionari["LAVABO"] = d("BAÑO", "BATHROOM");
    diccionari["ESCALADA"] = d("ESCALADA", "CLIMBING");
    diccionari["TIR AL PLAT"] = d("TIRO AL PLATO", "SHOT ON THE PLATE");
    diccionari["PLATAFORMES"] = d("PLATAFORMAS", "PLATFORMS");
    diccionari["ESPERA..."] = d("ESPERA...", "WAIT...");
    diccionari["SOM-HI!"] = d("¡VAMOS!", "GO!");
    diccionari["MEMORITZAR"] = d("MEMORIZAR", "MEMORY");
    diccionari["FRUITA"] = d("FRUTA", "FRUIT");
    diccionari["VERDURA"] = d("VERDURA", "VEGETABLES");
```

Figura 161: Part de les traduccions emmagatzemades a l'*script languageController.cs.*

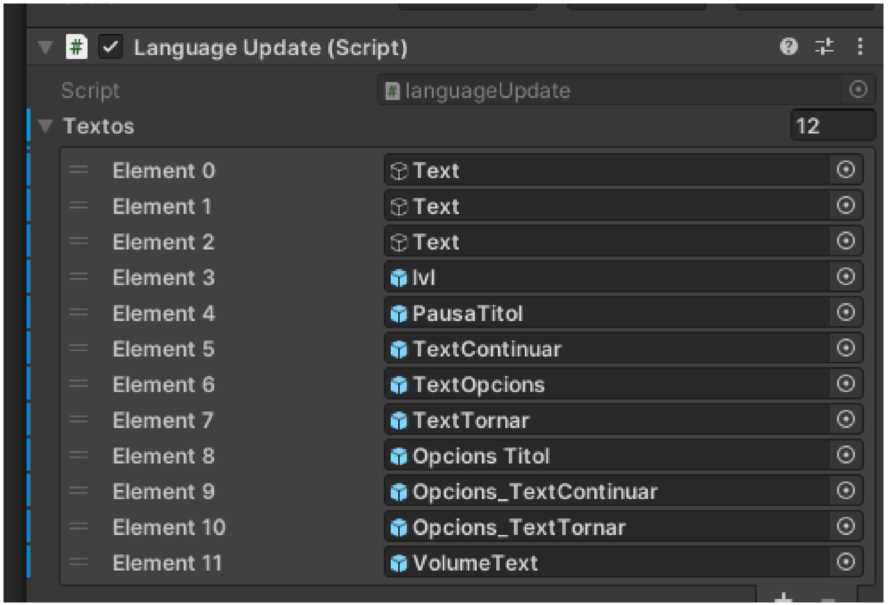


Figura 162: Exemple d'instància d'un objecte amb l'*script languageUpdate.cs* i tots els textos de l'escena que s'han de traduir

Finalment es va crear un *Prefab* anomenat *Idiomes* amb 3 botons, fent referència als 3 idiomes que es pot traduir el joc. També es va crear un *script* anomenat *menuIdiomes.cs* encarregat de canviar l'idioma segons els botó que es premés amb una funció de l'*script languageController.cs*, la qual també s'encarrega de cridar l'*script languageUpdate.cs* per actualitzar tots els textos del joc i traduir-los (Veure Figures 163 i 164).

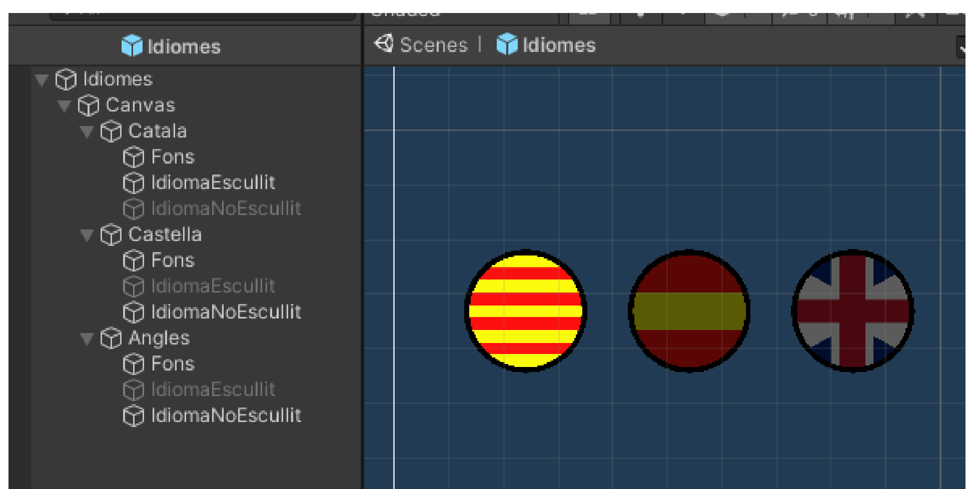


Figura 163: *Prefab Idiomes*

```
3 referencias
public void setLanguage(string l)
{
    language = l;
    PlayerPrefs.SetString("Language", l);

    int lNum;
    if (l == "cat") lNum = 0;
    else if (l == "cast") lNum = 1;
    else lNum = 2;
    languageUpdate.instance.updLan(lNum);
}
```

Figura 164: Funció de l'*script languageController.cs* que es crida sempre que es vol canviar d'idioma

Després de desenvolupar aquesta funcionalitat, es va veure que hi havia dos tipus de textos que no es podrien traduir amb aquesta funcionalitat. Aquests textos són els que es mostren a la interfície dels minijocs abans de jugar, els coneguts com tutorials, els quals s'analitzaran en els següents apartats, i els textos dels botons d'aliments i complements. Això és perquè es tracten de textos dinàmics, és a dir, el seu contingut varia via *script*, d'aquesta manera l'*script languageUpdate().cs* no pot obtenir una traducció ferma perquè el text canvia en diferents situacions.

Per solucionar el tema dels tutorials dels minijocs, es va afegir una nova variable a l'*script languageController.cs*, un diccionari on la clau és un identificador del text que s'ha de mostrar, específicament es tracta del nom del minijoc i de l'ordre dels textos, començant des de 0. Per exemple, si es tracta del segon text que s'ha de mostrar del minijoc d'Escalada, la paraula identificadora serà "*Escalada1*". Així doncs, el valor és una llista on el primer element és el text en català, el segon en castellà i el tercer en anglès (Veure Figura 165). D'aquesta manera, un mateix objecte *Text* podrà variar de contingut i seguirem tenint les traduccions.

```
textosTutorials["Escalada0"] = dTut("T'AGRADA ESCALAR PELS ARBRES? INTENTA ARRIBAR EL MÉS AMUNT POSSIBLE!",
                                  "¿TE GUSTA ESCALAR POR LOS ÁRBOLES ? ¡INTENTA LLEGAR LO MÁS ARRIBA POSIBLE!",
                                  "DO YOU LIKE CLIMBING THE TREES? TRY TO GET THERE AS FAR AS POSSIBLE!");
textosTutorials["Escalada1"] = dTut("FES CLIC A LES FUSTES QUE NO ESTIGUIN TRENCADES PER AVANÇAR",
                                  "HAZ CLIC EN LAS MADERAS QUE NO ESTEN ROTAS PARA AVANZAR",
                                  "CLICK ON WOODS THAT ARE NOT BROKEN TO MOVE FORWARD");
textosTutorials["Memoritzar0"] = dTut("SE'T MOSTRARÀ UNA SEQÜÈNCIA QUE HAURÀS DE MEMORITZAR",
                                  "SE TE MOSTRARÁ UNA SECUENCIA QUE DEBERÁS MEMORIZAR",
                                  "YOU WILL BE SEE A SEQUENCE THAT YOU MUST REMEMBER");
textosTutorials["Memoritzar1"] = dTut("QUAN SE T'ACABI DE MOSTRAR, FES CLIC A CADA POSICIÓ",
                                  "CUANDO SE TE ACABE DE MOSTRAR, HAZ CLIC EN CADA POSICIÓN",
                                  "WHEN YOU HAVE BEEN SHOWN, CLICK ON EACH POSITION");
textosTutorials["Memoritzar2"] = dTut("CADA VEGADA LES SEQÜÈNCIES SERAN MÉS DIFICILS, FINS ON ARRIBARÀS?",
                                  "CADA VEZ LAS SECUENCIAS SERÁN MÁS DIFICILES, ¿HASTA DÓNDE LLEGARÁS?",
                                  "EVERY TIME THE SEQUENCES WILL BE HARDER, HOW FAR WILL YOU GET THERE?");
textosTutorials["Tir al Plat0"] = dTut("FES CLIC A TOTES LES MASCOTES QUE VAGIN SORTINT",
                                  "HAZ CLIC EN TODAS LAS MASCOTAS QUE VAYAN SALIENDO",
                                  "CLICK ON ALL PETS COMING OUT");
textosTutorials["Tir al Plat1"] = dTut("I ESTIGUES ATENT PERQUÈ CADA VEGADA N'HI HAURÀ MÉS I MÉS RÀPIDES",
                                  "Y TIENES QUE ESTAR ATENTO PORQUE CADA VEZ HABRÁ MÁS Y MÁS RÁPIDAS",
                                  "AND BE CAREFUL BECAUSE EVERY TIME THERE WILL BE FASTER AND FASTER");

textosTutorials["Plataformes0"] = dTut("SALTA TOTES LES PLATAFORMES QUE PUGUIS PER ACONSEGUIR MÉS MONEDES",
                                  "SALTA TODAS LAS PLATAFORMAS QUE PUEDAS PARA CONSEGUIR MÁS MONEDAS",
                                  "JUMP ALL THE PLATFORMS YOU CAN TO GET MORE COINS");
textosTutorials["Plataformes1"] = dTut("FES CLIC AL TEU OTI PER SALTAR. SI EL MANTENS CLICAT, SALTARÀ DURANT MÉS ESTONA",
                                  "HAZ CLIC EN TU OTI PARA SALTAR. SI LO MANTIENES CLICADO, SALTARÁ DURANTE MÁS TIEMPO",
                                  "CLICK ON YOUR OTI TO JUMP. IF YOU KEEP IT CLICKED, IT WILL JUMP LONGER");
```

Figura 165: Traduccions dels tutorials emmagatzemades a l'*script languageController.cs.*

D'altra banda, pels textos dels botons dels aliments i dels complements, es va afegir les variables del nom d'aquests a les estructures de *food* i *complement*. D'aquesta manera, quan es vol obtenir la informació de l'aliment o del complement per mostrar-la, en comptes d'obtenir sempre el mateix nom, primerament es consulta a l'*script languageController.cs* en quin idioma està el joc i seguidament s'obté el nom en l'idioma que interessi (Veure Figura 166).

```
food strawberry = new food();
strawberry.init("strawberry", 10, 5, "fruita", "MADUIXA", "FRESA", "STRAWBERRY");
lFruita.Add(strawberry);

food watermelon = new food();
watermelon.init("watermelon", 10, 5, "fruita", "SÍNDRIA", "SANDÍA", "WATERMELON");
lFruita.Add(watermelon);
```

Figura 166: Exemple d'emmagatzemat de traduccions als aliments

6.5. Menús

De menús a *Otis* només n'hi ha dos, el de l'inici que serveix per canviar el nom de la mascota i el de pausa que s'utilitza a tots els espais. Pel que fa al primer menú, a nivell d'implementació no és gaire complex, ja que l'únic que s'ha fet és que a l'escena d'inici del videojoc, la qual s'aprofundirà en els següents apartat, quan es prem el botó de jugar s'obre un panell on es pot efectuar el canvi de nom abans d'iniciar el joc (Veure Figura 167).

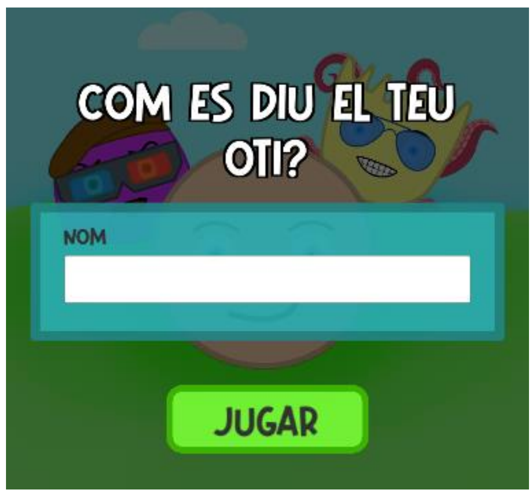


Figura 167: Menú d'inici per canviar el nom de la mascota

Aquest menú consta dels següents objectes: un panell de fons, un objecte de tipus *InputField* per escriure-hi el nom, un seguit de textos informatius i un botó per iniciar el joc. S'ha hagut de crear un *script* anomenat *mainMenu.cs*, el qual té una funció que desa el nom que se li ha posat a la mascota a través de la funcionalitats dels *PlayerPrefs*. El *PlayerPrefs* permet emmagatzemar variables del jugador úniques, és a dir, variables que cada persona que executi el joc tindrà valors diferents. En aquest cas poden ser el nom de la mascota, el nivell o les monedes (Veure Figura 168).

```
0 referencias
public void Play()
{
    if (name.text != "")
    {
        PlayerPrefs.SetString("name", name.text);
    }
    SceneManager.LoadScene("Exterior");
}
```

Figura 168: Funció utilitzada per canviar el nom de la mascota

D’altra banda, pel que fa al menú de pausa, es va decidir que tindria 3 botons, el de continuar amb el joc, el d’opcions i el de tornar a l’inici. Per controlar les funcionalitats d’aquests 3 botons, es va crear un *script* anomenat *pauseController.cs*. el qual es va associar a una icona de pausa i se li van crear 3 funcions per controlar la pausa i la continuació del joc i la tornada a l’inici (Veure Figura 169).

```
0 referencias
public void PauseGame()
{
    Time.timeScale = 0;
}
0 referencias
public void ResumeGame()
{
    Time.timeScale = 1;
}

0 referencias
public void tornarInici()
{
    SceneManager.LoadScene("Portada");
}
```

Figura 169: Funcions de l’*script pauseController.cs*

Tot seguit, es van crear els 3 botons. Al de continuar se li va associar com a tasca la funció *ResumeGame()* i al de tornar a l’inici la funció *tornarInici().* També es va afegir el *Prefab Idiomes* perquè es pogués canviar l’idioma del videojoc sempre que es pausi el joc (Veure Figura 170).

Figura 170: Botons del menú de pausa

Per acabar, només quedava el menús d'opcions, és per això que es va crear un objecte d'opcions que l'única funcionalitat és la de gestionar el volum del joc (Veure Figura 171).

Figura 171: Objecte que gestiona les opcions del joc

Per fer aquest objecte, es va crear un *script* anomenat *settingsController.cs* que gestiona la mostra de barres de volum segons l'àudio i que puja i baixa aquest volum amb els dos botons que hi ha als extrems de les barres (Veure Figures 172 i 173)

```
Mensaje de Unity | 0 referencias
private void Start()
{
    audioMixer.GetFloat("volume", out actVolume);

    int nBarsEliminar = (int)((actVolume / (80 / 5)) * -1);
    for(int i = 0; i < nBarsEliminar; i++)
    {
        volumeBars.transform.GetChild(4 - i).gameObject.SetActive(false);
    }

    nBars = 5 - nBarsEliminar;
}
2 referencias
public void SetVolume(float volume)
{
    audioMixer.SetFloat("volume", volume);
}
```

Figura 172: Funcions per mostrar més o menys barres en funció del volum del joc

```
0 referencias
public void upVolume()
{
    if (nBars < 5)
    {
        nBars++;
        volumeBars.transform.GetChild(nBars-1).gameObject.SetActive(true);
        actVolume += (80 / 5);
        SetVolume(actVolume);
    }
}

0 referencias
public void downVolume()
{
    if (nBars > 0)
    {
        nBars--;
        volumeBars.transform.GetChild(nBars).gameObject.SetActive(false);
        actVolume -= (80 / 5);
        SetVolume(actVolume);
    }
}
```

Figura 173: Funcions per pujar i baixar el volum del joc

Finalment, amb l'objecte d'opcions creat, es va associar aquest objecte al botó d'opcions del menú de pausa i amb això ja s'hauria implementat un *Prefab* anomenat *PausaUI*, el qual posteriorment s'integrarà a les interfícies del joc de manera senzilla.

6.6. Interfícies

Pel que fa a les interfícies del joc, s'han comentat els botons específics que hi ha a cada espai a l' apartat corresponent, així que en aquest apartat només cal que s'especifiqui com s'ha implementa el *HUD* i les interfícies específiques de cada minijoc.

6.6.1. HUD

Pel que fa al *HUD*, es va crear un *Prefab* anomenat *UI*, el qual conté tots els objectes necessaris per mostrar la informació que s'ha dissenyat a l'Apartat 5.6.1. Els objectes són els següents (Veure Figura 174):

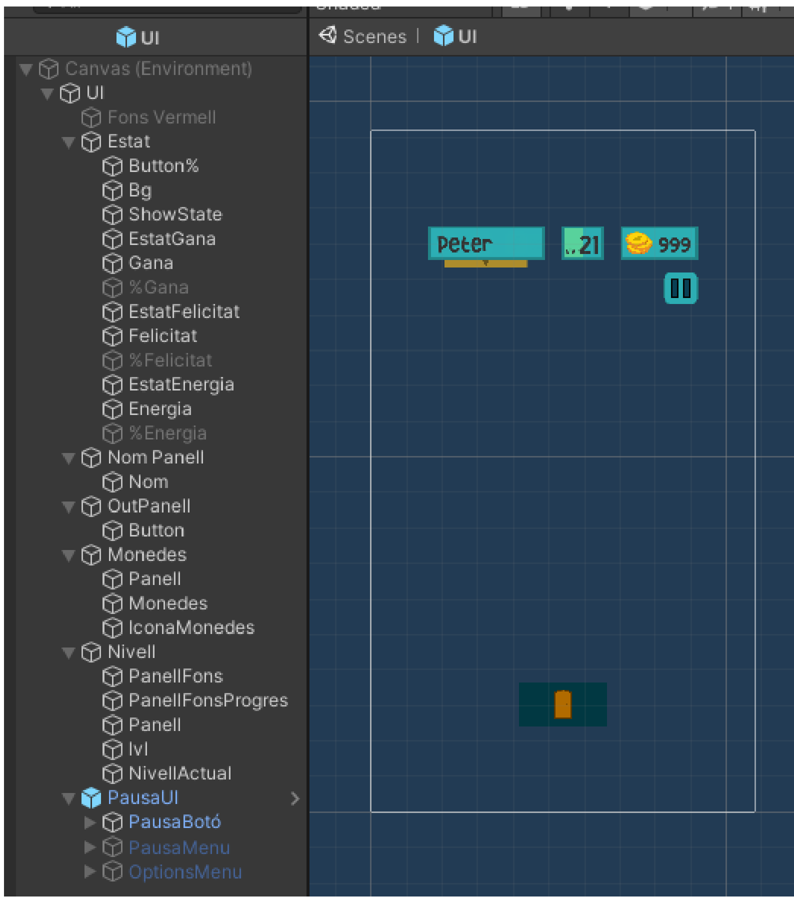
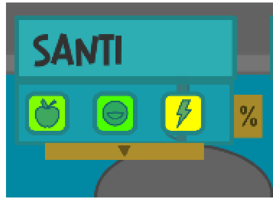


Figura 174: *Prefab UI*

- **Estat:** Objecte que gestiona la mostra d'informació relacionada amb l'estat de la mascota.
- **Button%:** Botó per canviar el mode de mostra de l'estat de la mascota
- **ShowState:** Botó encarregat de mostrar o amagar l'estat de la mascota
- **EstatGana:** Quantitat de gana que té la mascota, representada amb colors
- **Gana:** Icona que representa la gana de la mascota
- **%Gana:** Quantitat de gana que té la mascota, representada amb percentatges
- **EstatFelicitat:** Quantitat de felicitat que té la mascota, representada amb colors
- **Felicitat:** Icona que representa la felicitat de la mascota
- **%Felicitat:** Quantitat de felicitat que té la mascota, representada amb percentatges
- **EstatEnergia:** Quantitat d'energia que té la mascota, representada amb colors
- **Energia:** Icona que representa l'energia de la mascota
- **%Energia:** Quantitat d'energia que té la mascota, representada amb percentatges
- **Nom Panell:** Objecte que mostra el nom de la mascota
- **Out Panell:** Botó per sortir de l'espai en el que es troba la mascota
- **Monedes:** Nombre de monedes que té el jugador
- **NivellActual:** Nivell del jugador
- **PanellFonsProgres:** Punts de nivell que té el jugador representats amb una barra d'estat
- **PausaUI:** *Prefab* del menú de pausa del joc
- **Fons Vermell:** Objecte que serveix mer mostrar que algun dels estats de la mascota és molt baix

D'aquesta manera, es va decidir mostrar l'estat de la mascota amb tres icones, representant les diferents variables amb colors. Quan les icones són verdes, vol dir que l'estat de la mascota és bo, i quan són de color vermell vol dir que la mascota no està en un bon estat. Per mostrar l'estat de la mascota, es mostren la quantitat de gana que té, igual que la quantitat de felicitat i d'energia. Si una d'aquestes variables és molt baixa, automàticament es mostra un panell de color vermell. A més, el jugador pot escollir i vol veure les icones que representen aquestes variables o si les vol tenir amagades. També pot escollir en quin mode veu l'estat de la mascota, pot veure'l amb les icones amb els colors o veient directament els percentatge de cada variable (Veure Figura 175).



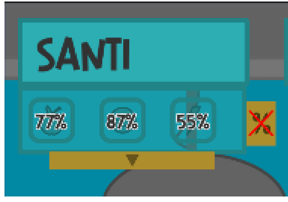


Figura 175: Visualització de l'estat de la mascota amb els diversos modes

Per implementar aquest *HUD* es van haver de crear 2 *scripts*. El primer, anomenat *UI.cs*, gestiona tota la informació que es mostra al *HUD*. Aquest té una sèrie de variables que fan referència a gairebé tots els objectes que s'acaben de mencionar. També té dues funcions, una *Update()* encarregada de mostrar el nom, el nivell i les monedes del jugador, igual que de mostrar el panell de color vermell si algun dels estats de la mascota és molt baix (Veure Figura 176). L'altra funció, anomenada *showPercent()* serveix per mostrar les icones amb els colors o amb els percentatges.

Figura 176: Panell que es mostra quan l'estat de la mascota no és bo

L'altre *script* que es va crear, anomenat *visualStats.cs*, serveix per obtenir la informació de l'estat de la mascota amb l'*script mascotaState.cs* i mostra-la amb els dos modes, mostrant els percentatges de quantitat o representant aquesta quantitat amb colors. Per fer aquest segon mode, es va crear en aquest *script* una funció anomenada *colorState()* (Veure Figura 177).

```
3 referencias
Color colorStat(string stat)
{
    float valor;
    if (stat == "energia")
    {
        valor = mascotaState.instance.getEnergia();
    }
    else if (stat == "felicitat")
    {
        valor = mascotaState.instance.getFelicitat();
    }
    else
    {
        valor = mascotaState.instance.getGana();
    }


    float colorChange;
    if (valor >= 50)
    {
        colorChange = (1.0f / 50.0f) * (100 - valor);
        return new Color(colorChange, 1, 0, 1);
    }
    else if (valor < 50 && valor > 0)
    {
        colorChange = 1 - ((1.0f / 50.0f) * (50 - valor));
        return new Color(1, colorChange, 0, 1);
    }
    else
    {
        return new Color(1, 0, 0, 1);
    }
}
```

Figura 177: Funció que mostra un color o un altre segons el valor de la variable

També es mostren el nivell de la mascota, el nom i les monedes que té el jugador, que són variables les quals el jugador manté cada vegada que s'executa el joc, ja que aquestes estan implementades amb la funcionalitat dels *PlayerPrefs*.

6.6.2. Interfícies dels minijocs

Tot i que cada minijoc ha de mostrar coses diferents, per implementar aquestes interfícies es va crear un *Prefab* anomenat *UIMIinijoc*, el qual conté tots els elements possibles que poden mostrar els minijocs. D'aquesta manera, per afegir les interfícies als minijocs només s'ha d'afegir aquest *Prefab* a les escenes, desactivar els objectes que innecessaris i concretar com s'afegeixen punts i monedes. Els objectes són els següents (Veure Figura 178):

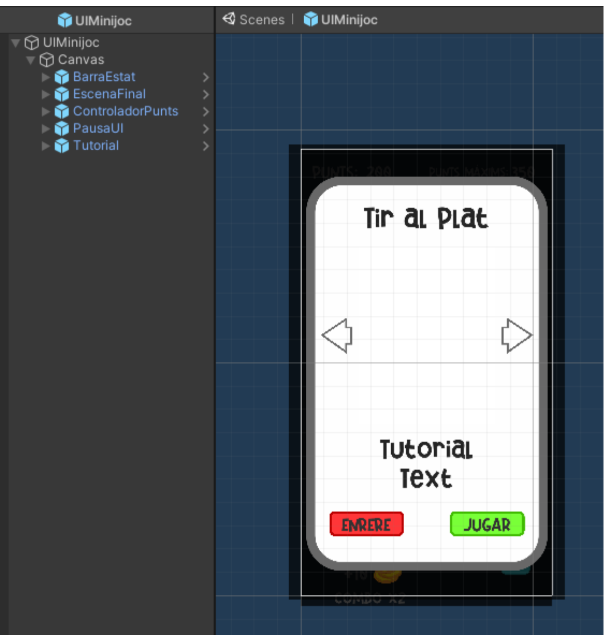


Figura 178: *Prefab UIMinijoc*

BarraEstat: *Slider* que representa el temps restant del minijoc. Aquest objecte té associat un *script* anomenat *sliderController.cs* que gestiona aquest temps, tenint com a variables el temps amb el que s'inicia el joc i el temps que es va disminuint a mesura que avança el joc. També té funcions per disminuir i augmentar el temps i per finalitzar el joc en cas que el temps arribi a zero (Veure Figura 179).

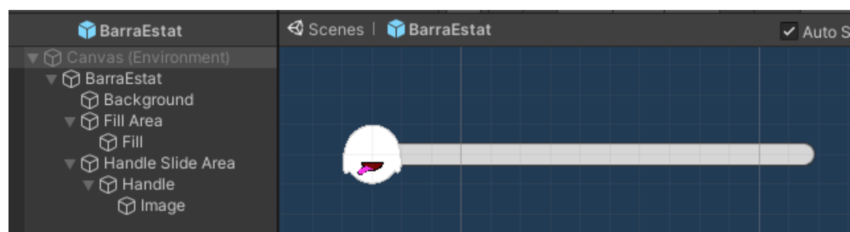


Figura 179: *Prefab BarraEstat*

EscenaFinal: Menú que es mostra quan s'acaba el minijoc, mostrant els punts que ha fet el jugador, el seu record personal i les monedes que ha obtingut a través d'un *script* anomenat *finalMinijoc.cs*. També té dos botons per sortir del minijoc o per tornar a jugar (Veure Figura 180).

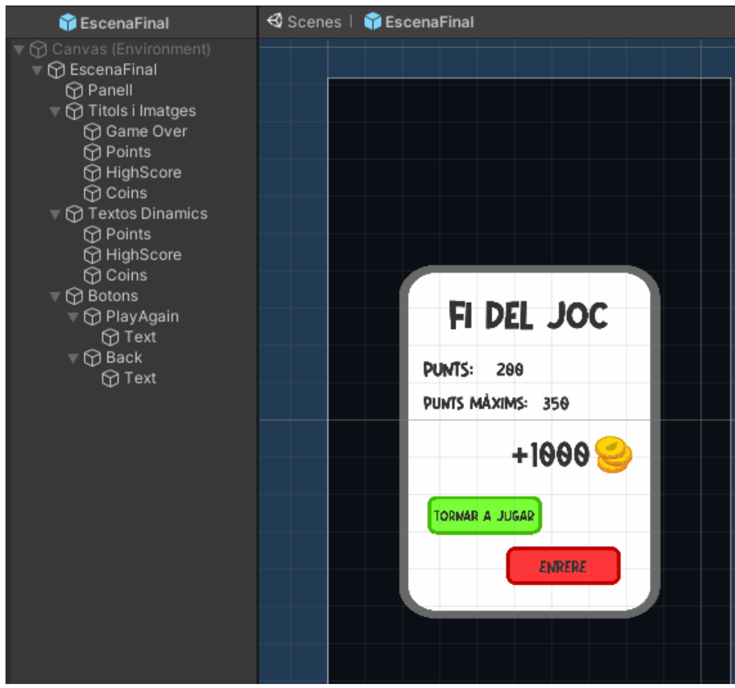


Figura 180: *Prefab EscenaFinal*

ControladorPunts: Tal i com indica el seu nom, es tracta d'un objecte que s'encarrega de mostrar els punts que porta el jugador al minijoc, a través d'un *script* anomenat *pointsController.cs*. També s'encarrega de mostrar en tot moment quin és el màxim de punts que ha fet el jugador en aquell minijoc i gestiona la suma de punts i de monedes. Pel que fa a la suma de monedes, el *Prefab* té un objecte amb una imatge de les monedes i dos textos: el nombre de monedes que es sumen i el *combo* que es pot fer en algun dels minijocs. Aquest objecte té una animació que es pot modificar amb l'*script* *pointsController.cs*, podent variar no només el nombre de monedes que s'afegeixen, sinó que també es pot establir la posició on s'instanci, de forma que no molesti a l'usuari mentre juga (Veure Figura 181).

Figura 181: Objecte que apareix quan se li sumen monedes al jugador

PausaUI: *Prefab* del menú de pausa, estudiat anteriorment a l'Apartat 6.5.

Tutorial: *Prefab* que mostra imatges i textos del funcionament del minijoc, abans que aquest comenci. Per implementar aquest objecte, es va crear un *script* anomenat *tutorialController.cs*, encarregat de gestionar el nombre d'imatges i textos que hi ha per a cada minijoc i mostrar-los en ordre. També té dos botons que permeten iniciar el minijoc o sortir (Veure Figura 182).

Figura 182: *Prefab Tutorial*

6.6.2.1. Interfícies de Tir al Plat

El minijoc de Tir al Plat és l'únic on apareix el concepte de *combo*. Cada vegada que el jugador toca un plat sense fallar, augmenta el valor del *combo*. Aquest valor serveix per augmentar el nombre de monedes que rep el jugador cada vegada que toca un plat. També és un dels minijocs on no es desactiva cap element del *Prefab UIMinijoc*, ja que els utilitza tot. Per implementar les funcionalitats de les interfícies, es va modificar l'*script* dels plats per augmentar el temps restant i afegir monedes i punts quan es toca un plat, igual que disminuir el temps restant i establir el combo a zero quan es falla un plat (Veure figura 183).

```
Mensaje de Unity | 0 referencias
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.tag == "collider")
    {
        Destroy(this.gameObject);

        if (platsGenerator.instance.getState())
        {
            sliderController.instance.decrementarValor(.1f);
        }

        pointsController.instance.setCombo(0);
    }
}

0 referencias
public void touched()
{
    if (platsGenerator.instance.getState())
    {
        sliderController.instance.augmentarValor(.02f);
        pointsController.instance.afegirPunts(1);
        pointsController.instance.afegirMonedes(platsGenerator.instance.mPlat(), transform.localPosition, true);

        Destroy(this.gameObject);
    }


}
```

Figura 183: Funcions modificades de l'*script platController.cs*

6.6.2.2. Interfícies d'Escalada

Al minijoc d'escalada tampoc se li desactiva cap element del *Prefab UIMinijoc*, i es modifica l'*script treesGenerator.cs* per augmentar el temps i afegir punts i monedes quan es prem l'arbre correcte (Veure Figura 184).

```
1 referencia
void seguentSequencia()
{
    sliderController.instance.augmentarValor(.001f);
    pointsController.instance.afegirPunts(1);

    imAct++;
    if (imAct == imMonedes)
    {
        pointsController.instance.afegirMonedes(1, new Vector2(0, -1000), false);
        imAct = 0;
    }
```

Figura 184: Part de l'*script treesGenerator.cs* que s'ha modificat

6.6.2.3. Interfícies de Memoritzar

Al minijoc Memoritzar, com que no hi ha temps restant, sinó que el joc s'acaba quan el jugador s'equivoca repetint una seqüència, es desactiva l'objecte *BarraEstat* del *Prefab UIMinijoc.* També es modifica l'*script sequenceGenerator.cs* per afegir monedes i punts cada vegada que es repeteix de forma correcta una seqüència (Veure Figura 185).

```
1 referencia
IEnumerator finalitzarRonda()
{
    pointsController.instance.afegirPunts(1);
    pointsController.instance.afegirMonedes(lenSeq, new Vector2(0, -300), false);

    yield return new WaitForSeconds(1);
```

Figura 185: Part de l'*script sequenceGenerator.cs* que s'ha modificat

6.6.2.4. Interfícies de Plataformes

Finalment, al minijocs Plataformes també es desactiva l'objecte *BarraEstat* del *Prefab UIMinijoc*, ja que el minijoc només finalitza quan la mascota no aconsegueix saltar una plataforma. També es modifica l'*script playerPlataformesController.cs* per afegir punts i monedes cada vegada que la mascota salta una nova plataforma (Veure Figura 186).

```
Mensaje de Unity | 0 referencias
private void OnCollisionEnter2D(Collision2D collision)
{
    if (primera)
    {
        pointsController.instance.afegirPunts(1);
        pointsController.instance.afegirMonedes(2, new Vector2(-179, -550), false);
        //SUMAR MONEDES I PUNTS
    }
```

Figura 186: Part de l'*script playerPlataformesController.cs* que s'ha modificat

6.6.3. Espais amb interfícies

A continuació es mostra a les Figures 187, 188 i 189 com queden tots els espais amb les interfícies afegides.

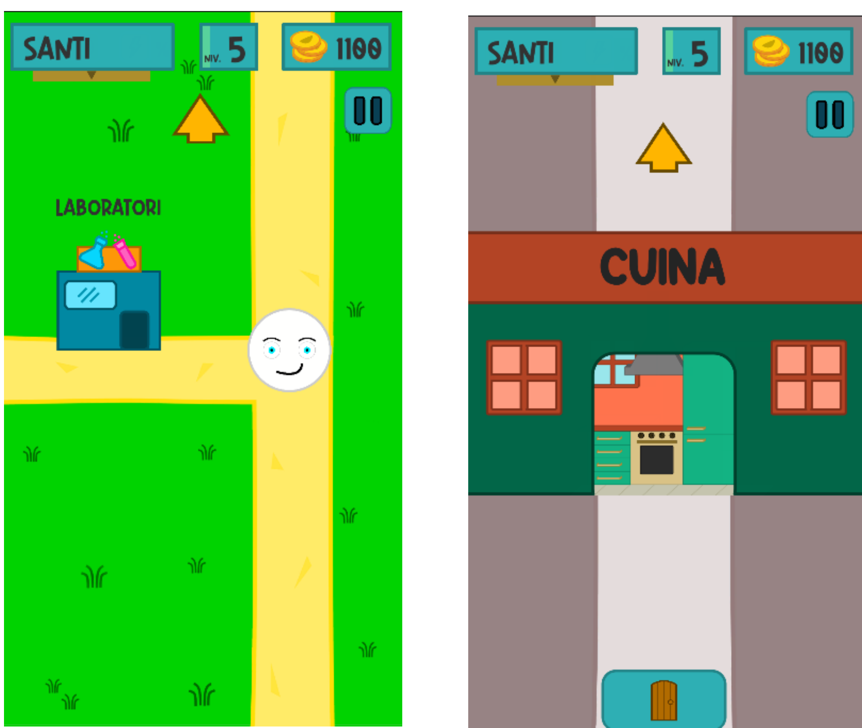


Figura 187: Espais exteriors amb les interfícies afegides

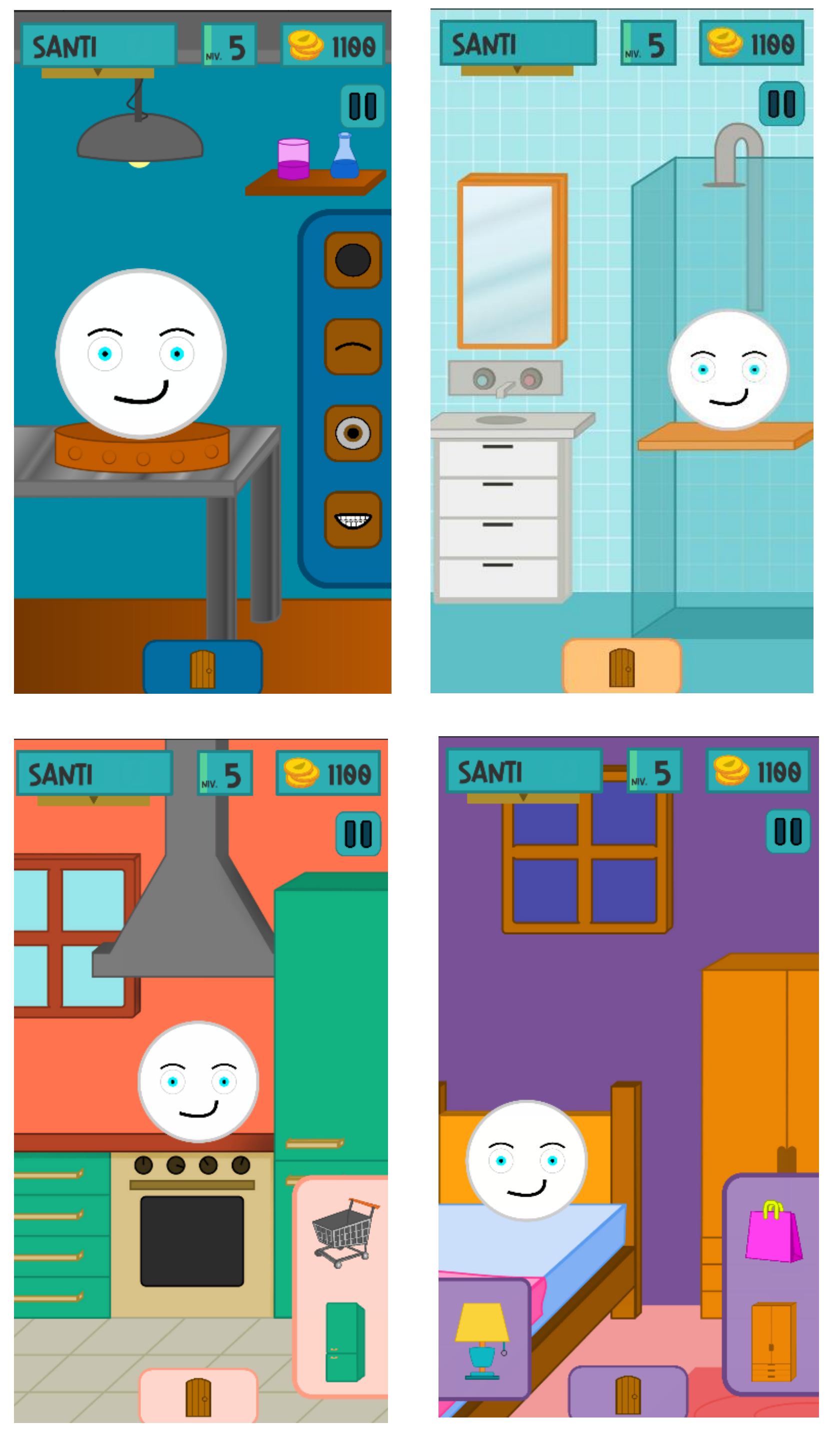


Figura 188: Espais interiors amb les interfícies afegides

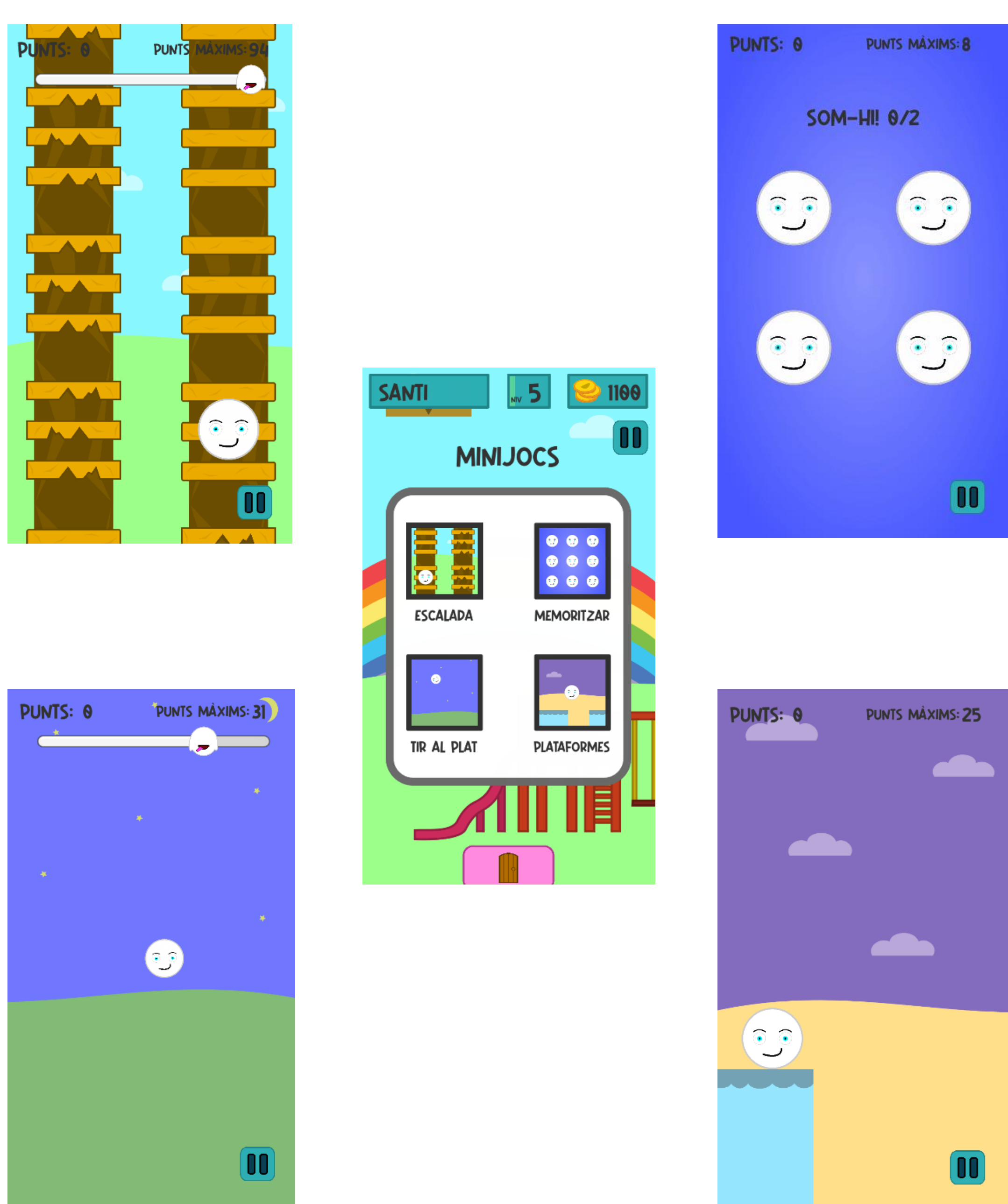


Figura 189: Minijocs amb les interfícies afegides

6.7. Inici del videojoc

Hi ha una escena del joc, anomenada *Portada*, que és la que es troba l'usuari quan inicia el joc. En aquest espai s'hi troba una foto de la portada, opcions per canviar d'idioma i un botó per iniciar el joc, el qual abans de portar al jugador a l'espai exterior, li mostra el menú per canviar el nom de la mascota (Veure Figura 190).

Figura 190: Portada d'*Otis*

A nivell d'implementació aquesta escena és molt important, ja que el fet de ser la primera escena que s'executa, és l'encarregada de crear un seguit d'objectes que emmagatzemen dades. Aquests objectes tenen la peculiaritat de que no es destrueixen en cap moment del joc i són únics, és a dir, s'instancien quan s'inicia el joc i no es tornen a instanciar mai més, ja que tenir les mateixes dades emmagatzemades diverses vegades no té cap mena de sentit. Aquests objectes són els següents (Veure Figura 191):

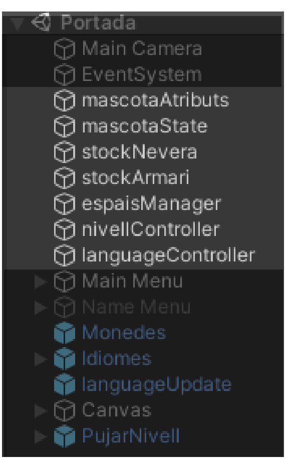


Figura 191: Objectes de l'escena *Portada*

- **mascotaAtributs:** Objecte que gestiona i emmagatzema les dades dels atributs de la mascota
- **mascotaState:** Objecte que gestiona i emmagatzema les dades de l'estat de la mascota
- **stockNevera:** Objecte que gestiona i emmagatzema els aliments que té el jugador
- **stockArmari:** Objecte que gestiona i emmagatzema els complements que té el jugador
- **espaisManager:** Objecte que controla a quin espai està la mascota en tot moment
- **nivellController:** Objecte que gestiona i emmagatzema el nivell del jugador
- **languageController:** Objecte que emmagatzema totes les traduccions del joc

6.8. Valors dels elements comprables

Tot i que s'ha explicat com estan implementats tan el menjar com els complements del videojoc, un últim pas que s'ha de fer per sumar aquests objectes al projecte és donar-los diferents valors. Pel que fa al menjar, s'ha d'afegir a cada aliment el nom en els tres idiomes, el preu, la categoria a la que pertany, el percentatge de gana que li treu a la mascota i una imatge. D'altra banda, a cada complement també se li ha d'afegir el nom en els tres idiomes, el preu, la categoria a la que pertany, el nivell en el qual es desbloqueja i una imatge.

De menjar s'han implementat un total de 6 categories (fruita, verdura, menjar ràpid, begudes i esmorzar), amb 8 aliments per a cada una d'elles. D'aquesta manera s'han arribat a afegir al joc un total de 48 aliments diferents. El preu que s'ha establert per a cada aliment va en funció del percentatge de gana que li treu a la mascota, com més ompli a la mascota més car serà aquest aliment. La relació preu-gana disminueix amb els aliments que omplen més, ja que sinó el jugador sempre compraria els aliments més barats.

I pel que fa als complements s'han implementat 8 complements per les 4 categories (cap, ulls, esquena i cos), així que s'han afegit al joc un total de 32 complements. El preu que s'ha establert per a cada complement va en funció del nivell en el qual es desbloqueja. Els complements que es desbloquegen amb nivells més alts són més cars, d'aquesta manera augmenta exponencialment la dificultat per obtenir cada un dels complements del joc.

Als Annexos de la memòria es mostren dues taules amb els valors que s'ha assignat a tots els aliments i complements del joc.

7. Proves i resultats

Durant el transcurs del desenvolupament, s'han anat realitzant diverses proves de forma contínua, per a comprovar el correcte funcionament de totes les mecàniques del videojoc que s'anaven implementant. Segons les proves que s'anaven fent, el disseny del joc podia arribar a variar una mica, però com a molt es van arribar a afegir funcionalitats, ja que es va dedicar molt temps a realitzar un disseny robust i es tenia molt clar en tot moment què i com s'havia d'implementar. Aquestes proves consistien en mirar si el joc anava fluït i com era l'experiència del jugador, i qui millor per provar aquests aspectes en un videojoc per infants, que els propis infants?

Tal i com s'ha comentat a la introducció de la memòria, quan es va iniciar el desenvolupament d'aquest projecte em trobava treballant de professor en una acadèmia de programació per nens. És per això que hi va haver la possibilitat de que els meus alumnes provessin les primeres mecàniques que s'anaven desenvolupant, i això va ser molt efectiu perquè aportaven solucions molt raonables. També tinc un germà de 13 anys que va seguir el desenvolupament del videojoc d'inici a fi, retornant-me un *feedback* que em va ser de gran ajuda, ja que ell està molt més acostumat que jo a jugar a videojocs d'aquest tipus, i les referències que té d'altres jocs van ser molt útils per acabar de definir algunes funcionalitats d'*Otis*.

Pel que fa a les proves de codi, aquestes es van anar realitzant constantment, utilitzant el mòdul *Debug* que proporciona *Unity*, utilitzant sempre la funció *Debug.Log()* i la pestanya de *Debug* de l'editor per controlar els valors de les diferents variables i els diversos objectes que hi ha al joc.

A continuació es mostraran diverses mascotes amb les modificacions que es poden arribar a fer a *Otis* (Veure Figures 192 i 193).

Figura 192: 6 *Otis* totalment diferents

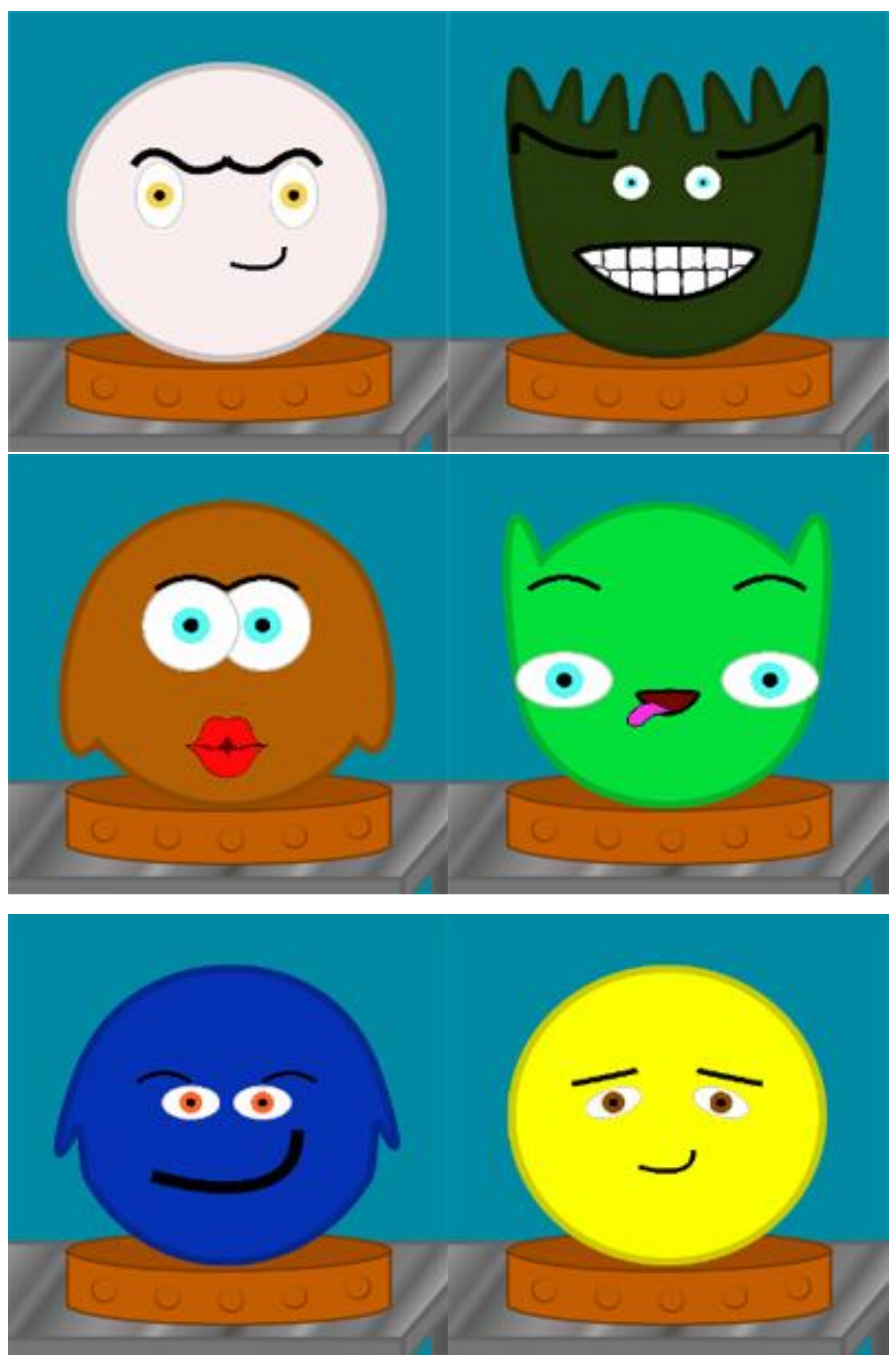


Figura 193: 6 *Otis* totalment diferents

8. Conclusions

En aquest apartat s'exposaran les conclusions extretes després d'haver desenvolupat el projecte d'*Otis*. Pel que fa a l'objectiu principal, el qual era acabar desenvolupant un videojoc sencer, s'ha complert de manera satisfactòria. És evident que, amb el temps que s'ha disposat i els recursos tècnics i humans amb els que s'ha treballat, el producte final era impossible que acabés sent competència directa dels màxims referents d'aquest sector, ja que s'està parlant de projectes realitzats amb grans pressupostos, fets per expert en la industria i amb més d'uns mesos per implementar. Tot i això, personalment opino que el joc té moltíssim potencial, sobretot pel disseny que hi ha al darrere, del qual és del que em sento més orgullós.

Deixant de banda el resultat final per un moment, ha estat tot un repte desenvolupar un videojoc des de 0, ja que ha estat la primera vegada que m'encarregava de realitzar un projecte d'aquestes característiques jo sol. Penso que s'han establert unes pautes molt clares i que s'ha seguit amb precisió la planificació realitzada a inicis de setembre, veient-se això reflectit en com ha anat avançant el desenvolupament del joc. Personalment em vaig posar l'objectiu de dissenyar un sistema que em facilités la feina en el moment d'implementar i ampliar el contingut, dedicant-li molt temps a idear mecàniques que podria tenir el joc i adaptant el disseny segons aquestes funcionalitats. D'aquesta manera, analitzant el codi i l'estructura del joc, es pot estar molt satisfet amb el resultat final, ja que s'han complert amb escreix les expectatives que hi havia a cada part del desenvolupament.

A nivell de disseny, aquest projecte m'ha demostrat que es tracta d'una fase importantíssima en la creació d'un videojoc, ja que si es tenen les coses clares des d'un principi, es gaudeixen molt més els moments de la implementació, tan a nivell artístic com de programació. En cap moment del desenvolupament m'he sentit incòmode amb el que estava fent, perquè tenia molt clar en tot moment com s'havien d'implementar totes i cada una de les parts d'*Otis*, i tot gràcies a la bona feina que es va fer a inicis del projecte amb el disseny del joc.

Sobre la programació, el resultat reflecteix el que s'ha comentat sobre el disseny. El fet de tenir establerta l'estructura del videojoc des del principi, va facilitar molt la feina de programar. A més, el fet de que sigui la meva part preferida en el desenvolupament d'un videojoc, ha ajudat a que no se'm fes difícil agafar l'ordinador i posar-me a programar en qualsevol moment i qualsevol aspecte del joc. Amb el resultat final estic molt content, sobretot amb la part de modificació de la mascota, ja que des de l'inici del projecte vaig comunicar-li al meu tutor que volia que aquesta mecànica destaqués, i que al projecte final el jugador tingués un gran ventall d'opcions per crear una mascota i fer-se-la pròpia.

Finalment, a nivell artístic també puc estar satisfet amb el resultat final. Des del principi que sabia que l'art no és el meu punt fort, i és per això que es va dissenyar tot el joc tenint això en compte, sense haver-me d'obligar a crear elements artístics complexos, ja que l'únic que hagués suposat és la pèrdua de motivació en el projecte. De totes maneres, artísticament el joc ha quedat molt coherent, que és el que més em

preocupava al cap i a la fi, i això em demostra que per molt que els elements artístics del joc fossin generalment simples, el resultat pot seguir sent més que decent i acabar obtenint un producte que sigui agradable a la vista.

Pel que fa a aprenentatge a través d'aquest projecte, gràfica i artísticament és on més he après, ja que només el fet d'haver-me encarregat de tota la part artística del joc, m'ha obligat a buscar-me la vida i obtenir un bon resultat amb els recursos que tenia. També he après molt sobre el motor de joc utilitzat: *Unity*. Gràcies a haver desenvolupat aquest projecte conec molt més les diferents funcionalitats d'aquest motor, i tot i no ser un expert del tema, actualment domino molt el desenvolupament 2D amb *Unity*. A nivell de programació no diria que ha estat un dels punts on hagi après o millorat més, però m'ha reforçat molt la idea de voler-me dedicar a aquest sector en un futur, ja que durant tota la implementació no he perdut la motivació per programar el joc, és més, m'ha ajudat a evadir-me de moltes situacions en les que m'he trobat durant aquests mesos de desenvolupament i ha estat una via d'escapatòria en molts moments.

És per tot això que puc dir que àmbits generals estic molt satisfet de la feina que s'ha realitzat, perquè tot i les diferents circumstàncies que m'he anat trobant mentre desenvolupava *Otis*, he aconseguit arribar a l'objectiu d'implementar des de zero un videojoc acabat i amb un potencial enorme.

9. Treball futur

Tot i que en l'apartat anterior es comenta que el resultat final és molt bo, és evident que pel fet de ser un únic desenvolupador i tenir un temps bastant limitat han quedat moltes coses que es podrien haver fet. També cal tenir en compte que aquest treball es va planificar en una situació on personalment disposava de bastant temps lliure, el qual a partir dels últims mesos va disminuir de forma extrema perquè vaig començar a treballar a jornada completa mentre seguia acabant el grau. A continuació es presentarà un llistat d'aspectes que es podrien implementar en un futur:

- **Sons i música**. És un aspecte que es va deixar pel final del desenvolupament, ja que la idea era tenir totes les funcionalitats i mecàniques llestes per saber exactament quin sons s'havien d'implementar. Es va decidir deixar-ho pel futur.

- **Afegir més contingut per la modificació de la mascota**. Tal i com està dissenyat i implementat el joc, afegir contingut al videojoc és molt senzill, però aquest contingut primerament s'ha de crear. Tampoc es va poder arribar a crear tants *assets* com es volia, tot i que en un futur pròxim l'únic que s'ha de fer és crear les formes de les diferents parts que es volen sumar al joc, i afegir-les a les carpetes corresponents, sense haver de tocar res de codi.

- **Afegir més animacions.** Es tracta d'un aspecte similar a l'anterior, la suma d'animacions al joc és senzilla per la forma en la que aquest està implementat, però no s'ha tingut el temps que es desitjava per idear i desenvolupar totes les animacions que es volia per a la mascota, com poden ser més moviments dels ulls o més animacions en situacions concretes.

- **Implementar més minijocs**. Des de l'inici es va decidir que s'implementarien 4 minijocs, i seguidament es veuria si hi havia opció a fer-ne algun més, però per temes logístics no ha estat possible. La implementació dels 4 minijocs que es van dissenyar des del principi no va ser excessivament difícil, però per tornar-ne a dissenyar i implementar-ne de nous, és necessari bastant més temps.

- **Generar arxiu .apk**. També es va posar com a objectiu opcional arribar a publicar el joc a la *PlayStore*, després de generar l'arxiu .apk del joc, però per fer això s'ha de tenir un videojoc més complert, i això es tindrà quan s'afegeixin els aspectes que s'han esmentat en aquesta llista anteriorment.

10. Bibliografia

Unity – Manual: Experiencia de Juego en 2D

https://docs.unity3d.com/es/530/Manual/Overview2D.html

Unity – Manual: Rigidbody 2D

https://docs.unity3d.com/es/530/Manual/class-Rigidbody2D.html

Unity – Manual: UnityEvents

https://docs.unity3d.com/es/530/Manual/UnityEvents.html

Unity – Manual: Animation Clips

https://docs.unity3d.com/es/530/Manual/AnimationSection.html

Unity – Documentation: PlayerPrefs

https://docs.unity3d.com/ScriptReference/PlayerPrefs.html

Game Development – In Unity, how do I correctly implement the singleton patern?

https://gamedev.stackexchange.com/questions/116009/in-unity-how-do-i-correctly-implement-the-singleton-pattern

Game Development – What is best for 2D movement: velocity or AddForce?

https://gamedev.stackexchange.com/questions/163850/what-is-best-for-2d-movement-velocity-or-addforce

Adobe Color – Color Wheel, a color palette generator

https://color.adobe.com/create/color-wheel

Coolors – The super fast color paletes generator

https://coolors.co/

PabloMakes – Color Picker in Unity

https://www.youtube.com/watch?v=rKhFYxUNL6A

Brackeys – Settings Menu in Unity

https://www.youtube.com/watch?v=YOaYQrN1oYQ

Brackeys – 100 Unity Tips

https://www.youtube.com/watch?v=thA3zv0IoUM

Unity – Answers: move 2d character affected by physics with velocity and/or add force

https://answers.unity.com/questions/1726362/move-2d-character-affected-by-physics-with-velocit.html

Unity – Forum: What determines canvas render order?

https://forum.unity.com/threads/what-determines-canvas-render-order.266682/

Unity – Documentation: Resources.Load

https://docs.unity3d.com/ScriptReference/Resources.Load.html

11. Annexos

11.1. Taula Aliments

Imatge	Nom Català	Nom Castellà	Nom Anglès	Preu	%Gana	Categoria
	poma	manzana	apple	2	2	Fruita
	plàtan	plátano	banana	4	5	Fruita
	cireres	cerezas	cherries	4	5	Fruita
	kiwi	kiwi	kiwi	6	8	Fruita
	taronja	naranja	orange	6	8	Fruita
	pera	pera	pear	10	15	Fruita
	maduixa	fresa	strawberry	10	15	Fruita
	síndria	sandía	watermelon	15	25	Fruita
	remolatxa	remolacha	beetroot	2	2	Verdura
	bròquil	brócoli	broccoli	4	5	Verdura
	pastanaga	zanahoria	carrot	4	5	Verdura
	blat de moro	maíz	corn	6	8	Verdura
	enciam	lechuga	lettuce	6	8	Verdura

	pebrot	pimiento	pepper	10	15	Verdura
	patata	patata	potatoe	10	15	Verdura
	tomàquet	tomate	tomatoe	15	25	Verdura
	cafè	café	coffee	2	2	Begudes
	aigua	agua	water	4	5	Begudes
	xocolata desfeta	chocolate derretido	melted chocolate	4	5	Begudes
	suc	zumo	juice	6	8	Begudes
	llimonada	limonada	lemonade	6	8	Begudes
	refresc	refresco	soft drink	10	15	Begudes
	beguda energètica	bebida energética	energy drink	10	15	Begudes
	llet	leche	milk	15	25	Begudes
	cereals	cereales	cereals	2	2	Esmorzar
	galetes	galletas	cookies	4	5	Esmorzar
	croissant	croissant	croissant	4	5	Esmorzar

	magdalena	madalena	cupcake	6	8	Esmorzar
	donut	donut	donut	6	8	Esmorzar
	ou ferrat	huevo frito	fried egg	10	15	Esmorzar
	entrepà	bocadillo	sandwich	10	15	Esmorzar
	torrades	tosatadas	toasts	15	25	Esmorzar
	aletes de pollastre	alitas de pollo	chicken wings	2	2	Menjar Ràpid
	patates fregides	patatas fritas	fries	4	5	Menjar Ràpid
	hamburguesa	hamburguesa	burger	4	5	Menjar Ràpid
	frankfurt	frankfurt	hot dog	6	8	Menjar Ràpid
	kebab	kebab	kebab	6	8	Menjar Ràpid
	pizza	pizza	pizza	10	15	Menjar Ràpid
	sushi	sushi	sushi	10	15	Menjar Ràpid
	taco	taco	taco	15	25	Menjar Ràpid

	piruleta	piruleta	lollipop	2	2	Dolços
	regalèssia	regaliz	licorice	4	5	Dolços
	osets de caramel	ositos de caramelo	gummy bears	4	5	Dolços
	bombons	bombones	chocolate bonbons	6	8	Dolços
	barreta de xocolata	barrita de chocolate	chocolate bar	6	8	Dolços
	caramels	caramelos	candy	10	15	Dolços
	bastó de caramel	bastón de caramelo	candy cane	10	15	Dolços
	pastís	pastel	cake	15	25	Dolços

Taula 5: Aliments amb els seus valors corresponents

11.2. Taula complements

Imatge	Nom Català	Nom Castellà	Nom Anglès	Preu	Nivell Desbloqueig	Categoria
	gorra	gorra	cap	10	3	Cap
	cresta	cresta	crest	15	6	Cap
	boina	boina	beret	20	9	Cap
	cua de cavall	cola de caballo	pony tail	25	12	Cap
	orelles de ren	orejas de reno	reindeer ears	30	15	Cap
	gorra de llana	gorro de lana	wool cap	35	18	Cap
	barret	sombrero	hat	40	21	Cap
	barret de bruixa	Sombrero de bruja	witch hat	45	24	Cap
	ulleres	gafas	glasses	8	1	Ulls
	ulleres de moda	gafas de moda	fashion glasse	13	4	Ulls
	ulleres de sol	gafas de sol	sun glasses	18	7	Ulls
	ulleres de cors	gafas de corazones	love glasses	23	10	Ulls
	monocle	monóculo	monocle	28	13	Ulls
	ulleres 3D	gafas 3D	3D glasses	33	16	Ulls
	ulleres de natació	gafas de natació	swimming glasses	38	19	Ulls
	ulleres de snow	gafas de snow	snow glasses	43	22	Ulls
	polo	polo	polo	9	2	Cos
	vestit	vestido	dress	14	5	Cos
	mascareta	mascarilla	mask	19	8	Cos
	pijama	pijama	pijama	24	11	Cos
	davantal	delantal	apron	29	14	Cos
	bufanda	bufanda	scarf	34	17	Cos

	vestit elegant	traje	suit	39	20	Cos
	superheroi	superhéroe	superhero	44	23	Cos
	cua	cola	tail	50	25	Esquena
	capa	capa	cape	60	27	Esquena
	katanes	katanas	katanas	70	29	Esquena
	papallona	mariposa	butterfly	80	31	Esquena
	ales d'àngel	alas de ángel	angel wings	90	33	Esquena
	cabell de lleó	pelo de león	lion hair	100	35	Esquena
	tentacles	tentáculos	tentacles	110	37	Esquena
	medusa	medusa	medusa	120	39	Esquena

Taula 6: Complements amb els seus valors corresponents

11.3. Vídeo

A continuació es mostra un enllaç a *YouTube* amb un vídeo on es poden veure totes les mecàniques d'*Otis* i l'ampli ventall d'opcions que hi ha per modificar la mascota del joc.

https://www.youtube.com/watch?v=f7mK4ZGMg8c