Universitat de Girona
Escola Politècnica Superior

Treball final de grau

Estudi: Grau en Disseny i Desenvolupament de Videojocs

Títol: Desenvolupament d'un generador de ciutats amb carrers interconnectats i un videojoc basats en *Wave Function Collapse*

Document: Memòria

Alumne: Oriol Perarnau Arnau

Tutor: Gustavo Patow

Departament: Informàtica, Matemàtica Aplicada i Estadística

Àrea: Llenguatges i sistemes informàtics

Convocatòria (mes/any): Juny 2021

Agraïments

A Gustavo Patow, tutor del projecte, per l'ajuda i optimisme que m'ha proporcionat durant el desenvolupament.

A Aleix Risco Mas per fer-me qüestionar els aspectes bàsics del videojoc per aconseguir millorar el disseny.

A Rúben Moreno Romero per a la creació de la majoria d'efectes del so del videojoc.

A la meva família i amics per a l'ajuda amb les proves i suport durant el confinament.

Índex

1 Introducció, motivacions, propòsit i objectius del projecte

1.1 Introducció

Des de la seva aparició, els videojocs han anat evolucionant i millorant amb la tecnologia d'avantguarda, aconseguint, per exemple, un temps de joc més elevat o millors gràfics. Una altra millora és la possibilitat de crear millor contingut procedimental. El contingut procedimental és aquell generat automàticament per la màquina seguint uns patrons o regles per aconseguir que el material generat sigui coherent. En el món dels videojocs és molt útil aquest contingut, ja que augmenta la re-jugabilitat del joc de manera gairebé infinita sense la necessitat d'un dissenyador. Per contra, es perd la capacitat de modificar detalls específics del contingut generat. Per tant, s'ha de vigilar què aquest contingut generat funcioni correctament, augmentant el cost del desenvolupament.

Tot i això, aquesta capacitat de re-jugabilitat infinita és la base de molts videojocs, entre ells cal destacar la generació de nivells dels gèneres *Roguelike*[1], *Roguelite*[2] i *Casual*[3]. Però el contingut procedimental no només serveix per a la generació de nivells. Per exemple el futur videojoc *Road 96* de l'estudi *DigixArt* proposa un videojoc amb una narrativa procedimental o per altra banda el famós *Spore* de la companyia *Electronic Arts* utilitza la generació en els personatges.

En existir diferents tipus de contingut procedimental, no hi ha un algorisme que serveixi per a tots els continguts. Un d'aquests algorismes per generar nivells és el *Wave Function Collapse (WFC),* el qual està inspirat en una idea de la mecànica quàntica i utilitza un conjunt de regles i superposició de diferents estats per a la generació de contingut. El *WFC* va aparèixer en el món de la informàtica en l'any 2016, però no va ser fins a la seva utilització en el videojoc *Bad North* de l'estudi *Plausible Concept* a l'any 2018, i amb la subseqüent presentació d'un dels seus creadors on va explicar el com l'utilitzaven, que no es va fer popular.

---

[1] *Roguelike*: Videojocs centrats en la mort permanent del personatge i en tornar a començar una nova partida cada cop.

[2] *Roguelite*: Variació del *Roguelike* amb millores permanents entre partida i partida

[3] *Casual*: Videojocs amb regles simples i poc complexes.

Per tot això, s'ha proposat per aquest projecte aconseguir fer un generador de ciutats utilitzant l'algorisme *WFC* i un videojoc aprofitant els nivells generats amb la mecànica principal de carrers interconnectats[4].

1.2 Objectius

1.2.1 Estudiar i entendre el funcionament de l'algorisme *WFC*

Per poder crear una ciutat utilitzant l'algorisme *WFC* primer s'ha de saber utilitzar i conèixer el funcionament per entendre les possibles limitacions que aquest pot portar en el disseny del videojoc i en la ciutat generada.

1.2.2 Crear un generador de ciutats

Crear un generador de ciutats capaç de generar una quantitat gairebé infinita de variacions. Aquest haurà de seguir regles físiques, com per exemple que una carretera no porti a un edifici, o que les portes d'aquests estiguin orientades al carrer.

1.2.3 Interconnectar els carrers de la ciutat generada

Per aconseguir la mecànica principal del videojoc s'haurà d'aconseguir interconnectar carrers, trobant la manera idònia i eficient.

1.2.4 Afegir mecàniques de videojoc a la ciutat

Afegir mecàniques de videojoc a la ciutat per crear un videojoc complet. Aquest haurà de tenir un principi i un final i ha d'ensenyar al jugador a jugar sense necessitat d'una ajuda externa.

1.3 Quadre d'autovaloració

Estètica	5%
Narrativa	5%
Mecàniques	40%
Tecnologia	50%

Taula 1. Quadre d'autovaloració

[4] Carrers interconnectats: En aquest projecte es definirà com a interconnectats quan dos carrers tot i no estar físicament un al costat de l'altre, tinguin una connexió que permeti la transició entre ells de manera contínua i invisible.

Com es pot veure en el quadre anterior (Taula 1), tot i tenir com a objectiu de projecte fer un videojoc complet, els pesos del projecte no estan repartits de manera balancejada.

Actualment en el món dels videojocs l'estètica és la millor manera de vendre un producte, generalment un videojoc que es veu bé vendrà més que un que tingui una millor idea. Per això s'ha decidit utilitzar art ja creat i per tant gairebé no se'n demana la valoració. A part, el no haver de produir l'art permet dedicar més temps a l'objectiu principal del projecte.

Respecte a la narrativa, com que la majoria de recursos del treball aniran destinats a la implementació del generador i de les mecàniques, s'ha decidit crear un videojoc casual, els quals no requereixen narrativa molt complexa. Tot i això, es necessari donar un mínim de sentit al videojoc, i per tant, se'n demana una avaluació mínima.

Les mecàniques del joc han de ser interessants i divertides, no serviria de res crear un generador de ciutats molt complex, si després no hi juga el jugador. Per això, un gran pes dels recursos del projecte aniran destinats a què aquest sigui divertit.

Finalment la tecnologia engloba la generació de la ciutat i la programació del videojoc, i essent aquesta la base en la qual es sustenta el projecte, és necessària que estigui ben feta, i per tant, utilitzarà la major part de recursos.

2 Estudi de viabilitat

Els videojocs requereixen una gran quantitat de recursos per a la seva creació, ja que requereixen moltes hores de desenvolupament i quan s'acaben de desenvolupar es necessiten més recursos per màrqueting, posar el joc a la venda, etc. Per conseqüència, abans de treure un projecte s'ha d'estar segur que aquest serà capaç de tornar tota la inversió, ja que crear videojocs és un negoci i, encara que una empresa tingui molta passió per un projecte, no acostumarà a fer mai una gran inversió sense un possible resultat positiu.

Per aconseguir saber si un videojoc serà rendible se n'ha de fer un estudi de viabilitat, mirant de manera detallada quin *software* i *hardware* s'utilitzaran per al projecte, quants treballadors hi treballaran i quin és l'estat de l'art en el mercat. Tot i que aquest projecte tingui un caràcter educatiu al ser un treball de final de grau, l'estudi serà fet com el que faria un estudi professional.

2.1 *Software*

Per aquest projecte s'utilitzarà diferents *softwares* per aconseguir-ne la creació. A continuació es llisten els més destacats i el perquè del seu ús.

2.1.1 *Unity*

Unity és el motor de videojocs que s'ha emprat. La primera versió d'aquest data del juny de 2005. És un motor fàcil d'aprendre el qual ha obert les portes al món dels videojocs a un gran públic. Utilitza el C# com a llenguatge i és gratuït[5]. S'ha decidit utilitzar per a la facilitat d'ús i per la gran quantitat de tutorials que existeixen sobre el funcionament.

Figura 2.1 Logotip de Unity

[5] És gratuït a condició que la companyia generi menys de 100.000$ a l'any.

2.1.2 *Visual Studio Community*

Editor de codi de *Microsoft* utilitzat per defecte a *Unity*. És una de les diferents versions del *Visual Studio*. Entre d'altres també hi trobem el *Visual Studio Professional, Visual Studio Enterprise* o *Visual Studio Code*.

Figura 2.2. Logotip de Visual Studio

2.1.3 *Blender*

En el capítol anterior s'ha explicat la decisió d'aconseguir art ja creat d'internet, tot i això no sempre es poden trobar tots els models que es requereixen, sigui per què són massa específics o per falta de pressupost. Per solucionar aquest problema, s'utilitzarà *Blender* per acabar de generar els models que manquen.

Blender és un programa opensource en el qual es pot modelar, renderitzar, animar i crear gràfics en 3D, tot i que en aquest projecte només serà emprat per a aquest últim propòsit.

Figura 2.3. Logotip de Blender

2.1.4 *Audacity*

Els videojocs necessiten efectes de so. Els efectes de so ens permeten donar vida i generar *feedback* pel jugador. Per aquest projecte no serà possible la creació, ja que requereix molta pràctica i esforç perquè quedin professionals i lliguin amb l'acció que representen.

Aleshores, els sons s'agafaran d'internet i, per retocar-los perquè s'integrin de manera natural, s'utilitzarà *Audacity*. *Audacity* és un programa opensource d'edició d'àudio.

Figura 2.4. Logotip d'Audacity

2.1.5 *Inkscape*

Els videojocs en 3D utilitzen una gran quantitat d'imatges, ja sigui petites icones, dibuixos per ensenyar els controls a l'usuari, el logotip del videojoc, etc. La gran majoria d'imatges es poden trobar a internet, però de la mateixa manera que amb l'art 3D, algunes imatges molt especifiques no serà possible trobar-les. Per tant, per generar l'art 2D del videojoc s'utilitzarà *Inkscape*. *Inkscape* és un programa *opensource* que permet fer dibuixos vectorials.

Figura 2.5. Logotip d'Inkscape

2.1.6 *Gimp*

En utilitzar imatges 2D en el videojoc serà necessari editar-les per solucionar possibles inconvenients com modificar saturació, retallar alguna imatge, etc. Com a editor d'imatges s'utilitzarà *Gimp*. *Gimp* és un programa opensource que permet editar i exportar imatges.

Figura 2.6. Logotip de Gimp

2.1.7 *Microsoft Word*

Finalment farà falta fer documents, com per exemple aquesta documentació. Com a editor de documents s'ha decidit emprar el programa *Microsoft Word*. *Microsoft Word* és un programa de pagament que permet editar documents.

Figura 2.7. Logotip de Microsoft Word

2.2 Recursos

2.2.1 Estació de treball

Per a la creació del videojoc es necessita una estació de treball capaç d'executar tot el *software* explicat a l'apartat anterior. En el cas d'aquest projecte s'utilitzarà un portàtil capaç de suportar un desenvolupament d'un joc en 3D.

- Processador *intel core* i-7
- 16 *Gigabytes* de *ram* dedicada
- Targeta gràfica *nvidia GeForce GTX* 1060

2.2.2 Recursos humans

Com s'ha explicat anteriorment, un videojoc utilitza diferents perfils de treballadors, en aquest projecte podem trobar els següents perfils i els sous estimats[6]:

- Programador: responsable de la programació del videojoc i el correcte funcionament.
    - Sou estimat:13 €/hora
- Dissenyador: responsable del disseny de mecàniques i com es relacionen amb el videojoc.
    - Sou estimat: 16 €/hora
- Artista: responsable de crear els models 3d i imatges necessaris per al videojoc
    - Sou estimat: 13 €/hora
- Dissenyador de so: responsable de crear els sons necessaris per al videojoc
    - Sou estimat: 13 €/hora
- *Tester*: responsable de provar les versions per trobar errors en aquestes.
    - Sou estimat: 8.65 €/hora

El projecte només serà desenvolupat per una sola persona agafant els diferents rols segons sigui convenient, amb l'excepció de *tester* pel qual es buscaran persones externes al projecte, per aconseguir més opinions.

[6] Sous estimats utilitzant la pàgina www.payscale.com

Es podria calcular el preu exacte dels treballadors utilitzant la planificació (vegeu Apartat 3), però en ser un projecte d'àmbit educatiu es considera que s'ha treballat gratuïtament.

2.3 Viabilitat

2.3.1 Cost del software

Software	Cost material nou
Unity	0 € , el projecte cau dintre el llindar gratuït del motor
Visual Studio Community	0 €, gratuït
Blender	0 €, *opensource*
Audacity	0 €, *opensource*
Inkscape	0 €, *opensource*
Gimp	0 €, *opensource*
Microsoft Word	135 €, s'utilitza la llicència d'estudiant per tant el cost real és 0 €
Total	0 €

Es pot observar que el cost real del *software* del projecte és de 0 €, gràcies al fet que tots aquests programes són gratuïts, *opensource* o ja se'n disposa de la llicència.

2.3.2 Cost dels recursos

Recurs	Cost
Estació de treball descrita	1300 €, ja es disposava amb anterioritat al projecte, pel que el cost real és 0 €
Treballador	0 €
Total	0 €

Es pot observar que el cost real dels recursos del projecte és de 0 €, gràcies a ja comptar amb el recurs amb anterioritat o es treballa gratuïtament per l'estil de projecte.

2.4 Estudi de mercat

Una pas important a l'hora de crear un videojoc és comprovar que no existeixi algun joc semblant i, en el cas que existeixi, poder assegurar que es podrà desenvolupar un que sigui millor o prou diferent per no acabar fent una còpia.

Per altra banda buscar la competència ajuda a veure quins videojocs semblants han aconseguit més vendes i veure quins són els punts claus que són necessaris per al projecte.

A continuació es poden veure alguns videojocs semblants a la idea d'aquest projecte.

2.4.1 Jocs semblants a l'objectiu

2.4.1.1 Crazy Taxi – 1999 – Sega

Videojoc en el qual el jugador pren el control d'un taxista el qual ha d'anar a buscar clients per la ciutat i portar-los a on aquests l'hi demanin. Es tracta d'un joc poc seriós i ràpid on s'ignoren les regles de tràfic amb l'objectiu d'arribar en el menor temps possible als punts demanats.

Figura 2.8. Captura del videojoc Crazy Taxi.

2.4.1.2 City Racing 3D – 2014 – 3DGames

Videojoc en el qual el jugador pot conduir una gran varietat de cotxes a dintre de circuits urbans a diferents ciutats del món. Joc de carreres sense objectes ni *power-up* centrat a arribar el primer a la meta.

Figura 2.9. Captura del videojoc City Racing 3D.

2.4.1.3 You suck at parking – 2021 – Happy Volcano

Videojoc en el qual el jugador ha d'aparcar un vehicle en un pàrquing. Amb la peculiaritat que el jugador no pot tirar marxa enrere ni frenar. El joc és poc seriós i de temàtica divertida i tranquil·la.

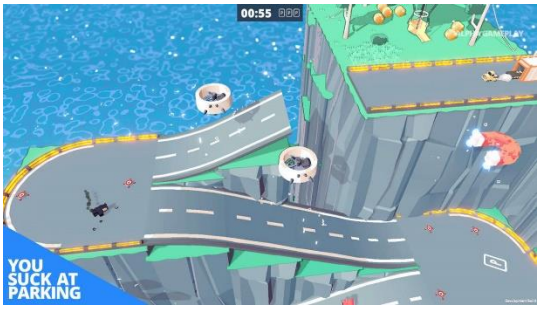

Figura 2.10. Captura del videojoc You Suck At Parking.

2.4.1.4 Absolute drift – 2015 – Funselektor Labs Inc.

Videojoc en el qual el jugador haurà de dominar "l'art de derrapar". en aquest joc el jugador controlarà un cotxe i haurà de completar un seguit de nivells i reptes en l'ordre que vulgui.

Figura 2.11. Captura del videojoc Absolute Drift

2.4.2 Quadre comparatiu

Observats els diferents videojocs que existeixen en el mercat, els compararem utilitzant la següent matriu de competència per acabar d'aclarir com es relaciona el projecte amb els competidors.

	Gràfics	Estètica	Caràcter	Gènere	Duració partida
Crazy taxi	3D	Retro	Caòtic	*Arcade*	Ràpida
City Racing 3D	3D	Realista	Seriós	Carreres	Ràpida
You suck at parking	3D	*Low poly*	Poc seriós i caòtic.	*Casual*	Ràpida
Absolute drift	3D	*Low poly*	Desenfadat i tranquil.	*Casual*	Lenta
Lost Ambulance	3D	*Low poly*	Poc seriós i frenètic.	*Casual*	Ràpida

- Gràfics: En quina dimensió són els gràfics del joc.
- Estètica: Fa referència a quin estil visual té joc.
- Caràcter: Quines són les sensacions que transmet al jugador.
- Gènere: Fa referència a la tipologia de joc.
- Duració de la partida: Fa referència a quina velocitat tenen les partides, essent una partida ràpida una que dura menys temps.

2.5 Públic objectiu i tipologia de jugador

Un punt important per a fer un videojoc coherent és entendre bé quin és el públic objectiu al qual va destinat. per altra banda també ens ajuda a dissenyar el joc pensant en aquest públic objectiu i per tant essent més eficaços.

En el subapartat anterior s'ha decidit que el videojoc serà de gènere casual, la particularitat d'aquests és que tenen molt poca limitació d'edat. Qualsevol jugador ha de poder començar a jugar sense problema i entendre el joc en pocs passos. Per culpa d'això el joc no tindria rang d'edat específic, però se'n pot determinar un mínim, essent aquest uns sis anys, ja que a partir d'aquesta edat els nens aprenen a llegir.

2.5.1 Taxonomia de Bartle

Una altra manera de classificar els possibles jugadors és utilitzant la taxonomia de Bartle, creada per Richard Bartle l'any 1996. En aquesta podem classificar els jugadors segons les seves motivacions.

Existeixen 4 tipus de jugadors:

- ***Achievers***: Busquen la diversió en completar el joc i en perfeccionar el seu mode de joc per aconseguir més punts. Interessats a actuar en el món.
- ***Explorers***: Busquen la diversió en explorar el contingut del joc, descobrir secrets del joc i contingut poc conegut d'aquest. Interessats a interactuar amb el món.
- ***Killers***: Busquen la diversió en lluitar contra altres jugadors. Interessats a actuar sobre altres jugadors.
- ***Socializers***: Busquen la diversió en la col·laboració amb altres jugadors. Interessats a actuar amb altres jugadors.

En el projecte no tenen sentit les dues últimes categories, ja que el videojoc no és en línia, i fer qualsevol sistema que permetés el multijugador sobrepassaria el temps disponible per a la producció d'aquest projecte.

També cal afegir que la tipologia *explorers* tampoc té sentit en un videojoc de gènere *casual*, ja que el joc està limitat a nivells i el jugador no hi té res a descobrir.

Per altra banda, la tipologia *achiever* defineix perfectament al jugador objectiu, ja que es busca un jugador que vulgui acabar el joc i vulgui perfeccionar el temps aconseguit per mitjà d'entendre i practicar les mecàniques del videojoc.

3 Planificació

Per al correcte desenvolupament d'un videojoc és necessària una bona planificació, ja que l'absència d'aquesta genera dificultats a l'hora de calcular el cost del videojoc. En la planificació han d'estar reflectides quines tasques són necessàries per arribar al final del desenvolupament.

3.1 Tasques

3.1.1 Tasques de programació

Les tasques de programació són les següents:

- **Generador de la ciutat**: centrada a aplicar l'algorisme *WFC* per a la generació del plànol d'una ciutat i, després, utilitzar-lo per afegir els models 3D d'edificis i carreteres per generar la ciutat completa.

- **Interconnexió de carrers**: centrada a interconnectar els carrers de la ciutat generada. Per aconseguir-ho s'haurà de trobar una solució eficient la qual sigui invisible per al jugador.

- **Moviment del jugador**: centrada en desenvolupar un moviment del jugador divertit. En aquest videojoc el jugador estarà en moviment pel mapa la major part de la partida. Aleshores, per a fer que el joc sigui divertit, s'haurà d'implementar un bon moviment i trobar formes de fer-lo més interessant.

- **Lògica del videojoc**: centrada en ajuntar els resultats de les tasques anteriors per a generar el joc final. A part, en aquesta tasca també s'afegirà la lògica per al correcte funcionament del joc, és a dir, codificar el principi i final de partida.

- **Implementació del sistema d'àudio**: centrada en generar un controlador capaç de tocar l'àudio del joc quan sigui necessari per donar *feedback* sobre els esdeveniments del videojoc.

- **Pantalles del videojoc**: centrada en implementar les diferents pantalles que són necessàries per al correcte funcionament d'un videojoc. Per aquest projecte definim les següents:
    - Pantalla de títol: mostra el títol del videojoc.

    - Menú principal: mostra i permet l'accés a les pantalles de selector de nivells i opcions i sortir del joc.
    - Selector de nivells: mostra i permet l'accés a tots els nivells disponibles per al jugador.
    - Opcions: permet modificar la configuració del videojoc.
    - Pantalla de victòria: mostra que el jugador ha aconseguit superar el nivell.
    - Pantalla de derrota: mostra que el jugador no ha aconseguit superar el nivell.
    - Pantalla de controls: mostra els controls en el jugador.
    - Menú de pausa: permet aturar el joc en mig d'una partida.

- **Poliment:** una de les parts més importants d'un videojoc actualment és que tan polit és. Definim polit com la propietat de què es vegi professional, és a dir, afegir *feedback* per a cada acció del videojoc, fer que els controls siguin millors, etc. Un exemple clar és quan un jugador agafa una moneda en un joc. Aquesta acció podria ser tan simple com augmentar un comptador i fer desaparèixer la moneda, però generarà un major estímul en el jugador si afegim un petit so i una petita animació quan l'agafem. Aleshores aquesta tasca es basa a revisar tot el *feedback* del joc perquè sigui bo, afegir nou *feedback* quan aquest sigui necessari i millorar els controls.

3.1.2 Tasques d'art i so

- **Creació d'art faltant:** centrada en generar els models 3D i imatges que no s'han pogut trobar a internet, sigui perquè són massa específics o per falta de pressupost.

- **Creació dels efectes de so faltants:** centrada en generar els efectes de so que no s'han pogut trobar a internet, sigui perquè són massa específics o per falta de pressupost.

3.1.3 Tasques miscel·lànies

- **Documentació:** centrada en generar aquest document.

- **Testeig:** centrada en fer testeig sobre el joc per trobar errors en les diferents versions.

- **Narrativa:** centrada en generar la història i l'explicació del funcionament del joc.

3.2 Metodologia de treball

Per gestionar el projecte s'ha decidit crear diferents *deadlines* per a cada paquet, tot i estar treballant una sola persona. Els *deadlines* són necessaris per a no descontrolar el temps que es dedica a cada tasca, poder acabar el joc de manera satisfactòria i no tenir un temps de desenvolupament superior a l'esperat.

A part, s'ha decidit aplicar la metodologia *Kanban*, una variació de la metodologia *Agile*[7], la qual se centra en tres grans columnes: per fer, fent-se i fet. Aquestes columnes representen l'estat de cada tasca. És una bona metodologia per entendre concretament quines tasques s'han de fer i en quin estat estan.

3.3 Diagrama de Gantt

En el següent diagrama (vegeu Figura 3.1) es pot observar com s'han distribuït les tasques segons el temps disponible per aquest projecte. A efectes pràctics, es considerarà que el projecte va començar l'octubre de 2020 i suposarà el final el juny de 2021.

En ser un sol integrant, les tasques estan ordenades en sèrie (tasques blaves) i no en paral·lel com seria en un estudi professional amb més treballadors. Tot i això, hi ha tasques que requereixen d'altres i són secundàries (verdes), per exemple les tasques de generar l'art faltant, ja que aquest art s'haurà creat per la necessitat durant el desenvolupament.

[7] *Agile*: Metodologia basada en el disseny iteratiu i incremental, és a dir, on els requisits i solucions evolucionen amb el desenvolupament.

Finalment s'ha decidit partir el temps de desenvolupament en paquets de mig mes, ja que el temps disponible per desenvolupament és llarg, i partir-ho en dies o setmanes hauria estat massa granulat.

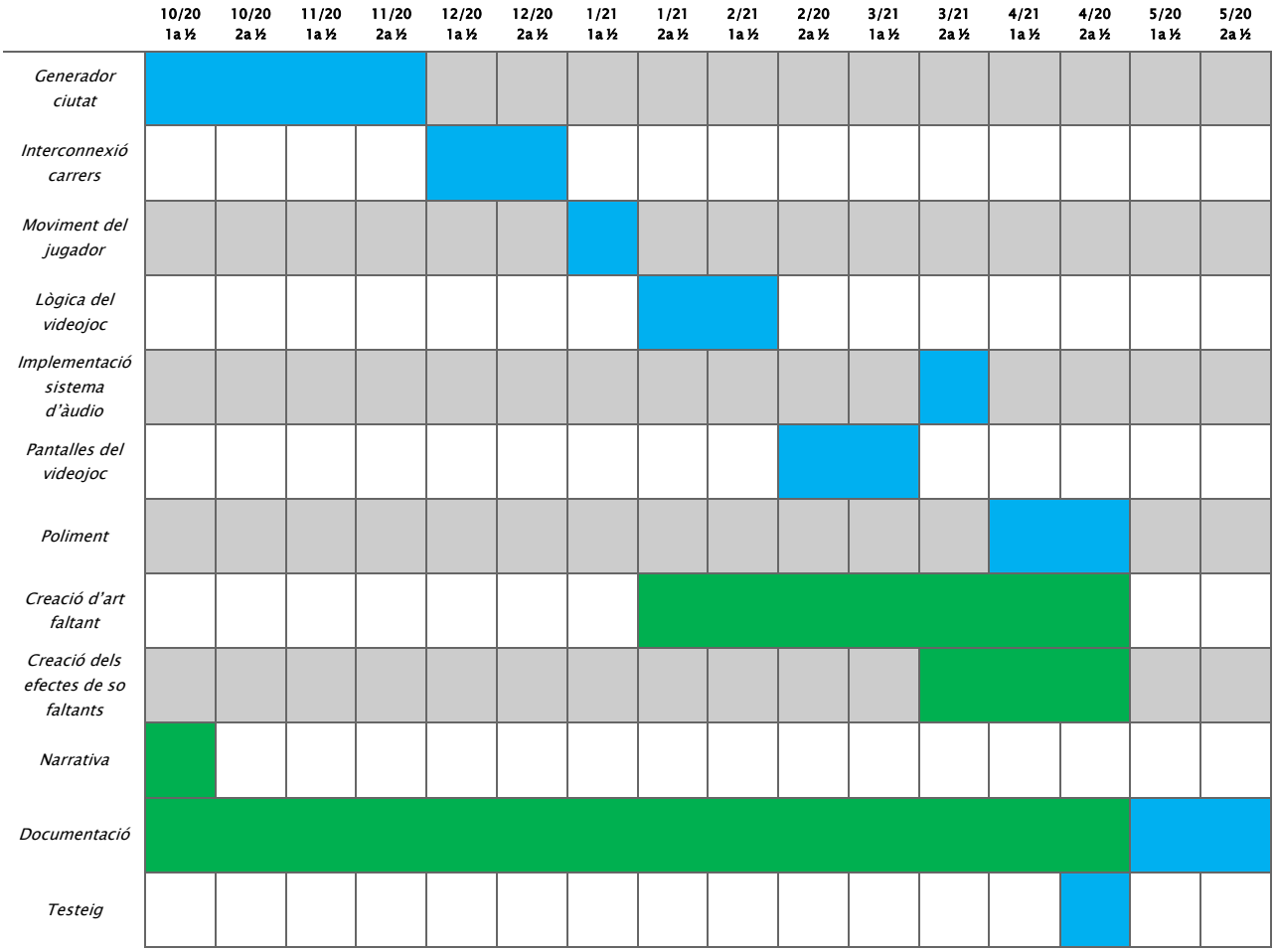

Figura 3.1. Diagrama de Gantt

4 Comercialització del videojoc

L'objectiu d'aquest projecte no ha estat mai aconseguir diners amb el videojoc, sí que és veritat que s'ha estructurat aquest projecte de tal manera que fos rendible econòmicament, però no s'ha cercat mai posar-lo a la venda.

Això és per culpa que la motivació principal d'aquest projecte no ha estat fer un videojoc el qual sigui supervendes, sinó la curiositat sobre el *WFC* i com trobar la manera d'adaptar-lo a un videojoc propi.

Per això el videojoc serà publicat a *itch.io*[8] de manera gratuïta perquè tothom que vulgui el pugui provar. Tot i això, en el cas que es volgués fer un videojoc comercial, sortiria al mercat amb un preu de 3 €, ja que el contingut actual del videojoc és molt reduït. En el cas d'afegir més sistemes, com per exemple personalització amb diferents vehicles, més modes de joc, etc, es podria augmentar el preu a 9 €.

[8] *Itch.io*: Pàgina web en la qual pots comprar i descarregar videojocs.

5 Marc de treball

Com que en l'apartat d'Estudi de mercat (vegeu Apartat 2.4) ja s'han mostrat els diferents videojocs semblants i com es diferencien del projecte, en aquest apartat s'explicarà amb detall com funciona l'algorisme de *WFC,* el paquet[9] de *Tessera* i el concepte d'interconnexió de carrers.

5.1 *WFC: Wave Function Collapse*

Com s'ha explicat anteriorment, el *WFC* està inspirat en la mecànica quàntica, per ser precisos, està inspirat en el principi de superposició. La superposició quàntica succeeix quan un objecte té més d'un estat i, quan aquest és observat, col·lapsa i passa a tenir-ne un.

Per explicar com funciona el *WFC* i com s'inspira en la superposició quàntica, s'utilitzarà el següent exemple.

5.1.1 Sudoku

Imaginem un Sudoku de quatre per quatre amb blocs de quatre[10] completament en blanc (vegeu Figura 5.1), cada casella d'aquest té la possibilitat de ser qualsevol dels quatre números a la vegada. Per tant es podria dir que cada casella té quatre estats superposats, un per a cada número (vegeu Figura 5.2).

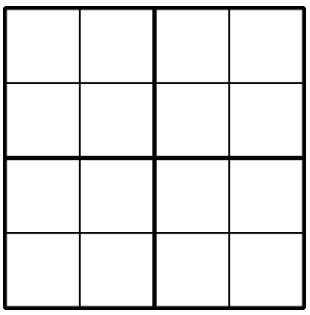

Figura 5.1. Sudoku en blanc

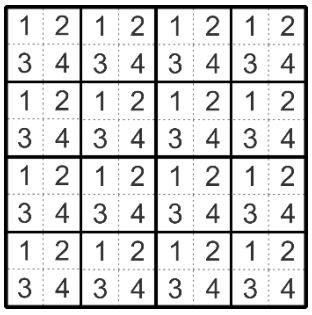

Figura 5.2. Sudoku amb estats

[9] Paquet: conjunt de codi, imatges, models, etc.

[10] Mides Sudoku: Per aquest exemple s'ha decidit utilitzar mides relativament petites per simplificar el procés.

Ara decidim agafar una casella aleatòria, per exemple el 2 i col·lapsar-la, és a dir, s'agafa un dels estats possibles (vegeu Figura 5.3). Aleshores, si es considera que la cel·la és correcta, per les regles del Sudoku[11], s'elimina la possibilitat de tenir l'estat 2 a les caselles que comparteixen fila, columna o bloc (vegeu Figura 5.4). A aquesta acció d'eliminar els estats impossibles s'anomena propagació.

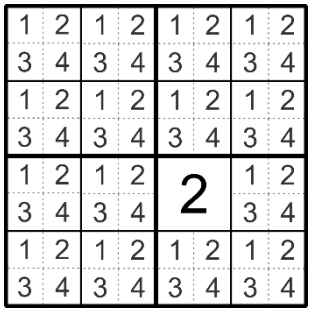

Figura 5.3. Sudoku amb casella col·lapsada

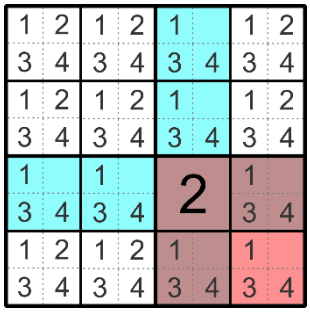

Figura 5.4. Visualització de la propagació d'informació

A continuació es busca la casella amb menys estats possibles, es col·lapsa a un estat aleatori i es propaga la informació a les altres caselles (vegeu Figura 5.5). Si aquest procés el fem en una iteració, acabem col·lapsant tot el Sudoku i donant una solució (vegeu figura 4.6).

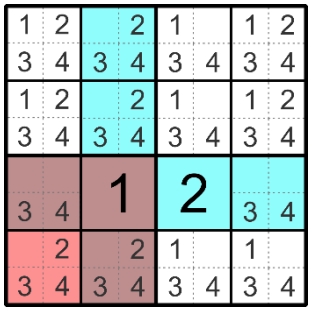

Figura 5.5. Visualització de la propagació 2

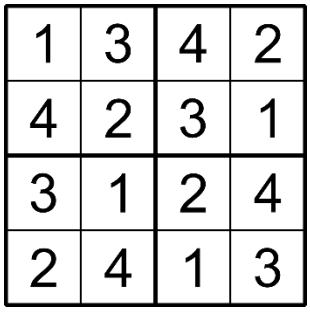

Figura 5.6. Sudoku complet

5.1.2 Funcionament del *WFC*

Es pot definir el funcionament del *WFC* de la següent manera. En primer lloc, es parteix d'una graella buida (en l'exemple anterior és la graella de caselles) en la qual se li poden posar diferents objectes, els quals formen

[11] Regles Sudoku: No hi poden haver repetits en el mateix bloc, en la columna o en la fila.

part d'una paleta (per exemple, els números en el Sudoku). Aquests objectes tenen unes regles sobre com poden ser col·locats (regles del Sudoku).

Aleshores s'itera sobre la graella de la següent manera:

1. Mentre no tenim solució completa.
    1.1. Seleccionar la cel·la amb menys estats superposats.
    1.2. Col·lapsar la cel·la amb un estat aleatori.
    1.3. Propagar informació.

Tot i això és possible que no arribi a una solució correcta encara que aquesta existeixi, ja que es pot donar la situació que una casella tingui zero estats superposats per culpa de la propagació de la informació. Quan això passa, l'algorisme de *WFC* té incorporat un *backtracking* el qual li permet anar des-col·lapsant cel·les i escollir un altre estat.

D'aquesta manera l'algorisme és el següent:

1. Crear pila de cel·les (PC)
2. Mentre no solució completa
    2.1. Selecciona la cel·la (C) amb menys estats superposats.
    2.2. Si C té 0 estats possibles:
        2.2.1. Des-empilar la cel·la (C') de PC
        2.2.2. Des-col·lapsar C'
        2.2.3. Eliminar de C' últim estat col·lapsat
        2.2.4. Propagar informació
    2.3. Si no:
        2.3.1. Col·lapsar C amb un estat aleatori
        2.3.2. Empilar C a PC
        2.3.3. Propagar informació

5.1.3 Exemples d'ús del *WFC*

5.1.3.1 Generació d'una ciutat infinita

En el següent exemple es pot veure com s'ha utilitzat l'algorisme de *WFC* per a generar una ciutat infinita. En primer lloc la paleta (vegeu Figura 5.7) que ha utilitzat el sistema i seguidament el resultat obtingut a la Figura 5.8.

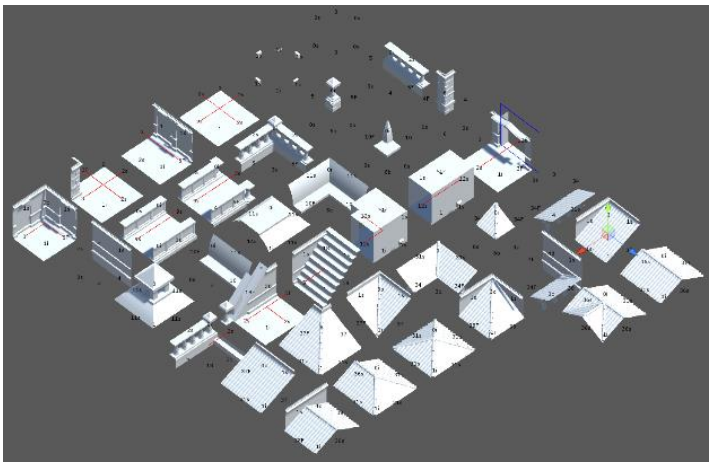

Figura 5.7. Estats possibles del sistema.

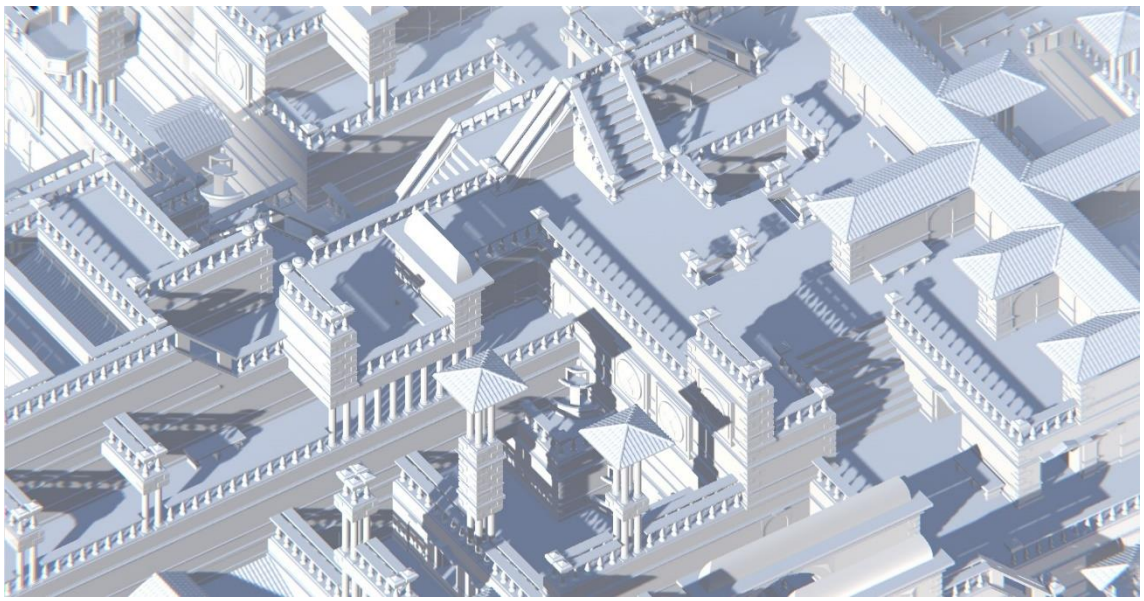

Figura 5.8. Resultat de la generació de la ciutat.

En l'exemple el jugador pot moure's en primera persona per la ciutat infinita, per aconseguir que sembli infinita, la ciutat es va generant a mesura que es mou el jugador.

5.1.3.2 Utilització en un videojoc comercial

El videojoc *Bad North* utilitza l'algorisme *WFC* per a generar les illes que utilitza com a nivells. A continuació es poden observar diferents resultats de la generació a la Figura 5.9. En aquest videojoc el jugador controla unes tropes per defensar cases d'uns enemics que venen en barca del mar.

Figura 5.9. Nivells procedimentals de Bad North

5.2 *Tessera*

Tessera és un paquet que porta implementat l'algorisme *WFC* en 3D, tot i que només s'utilitza en 2D en el projecte. Permet definir la paleta com entitats d'*Unity* i les regles utilitzant colors i una matriu per definir connexions.

Tot i permetre utilitzar entitats de qualsevol forma, l'algorisme les encapsula en contenidors amb forma de prisma rectangular anomenades *tiles*. Per tant la unió entre objectes ha de ser utilitzant bordes rectes (vegeu Figura 5.10).

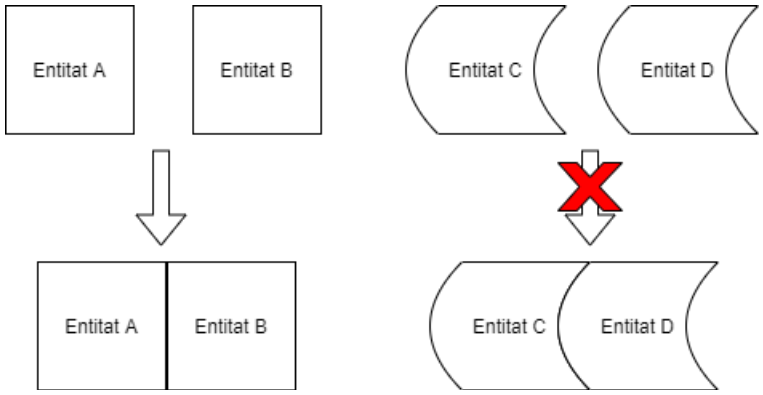

Figura 5.10. Funcionament Tessera.

5.2.1 Exemple funcionament *Tessera*

A continuació es mostrarà el funcionament del *Tessera* fora del context del videojoc del projecte i per entendre millor el funcionament.

En primer lloc es defineix una paleta d'objectes per popular el sistema (vegeu Figura 5.11). Per definir cada *tile* de la paleta s'utilitza el script *TesseraTile, com es mostra a la Figura 5.12.*

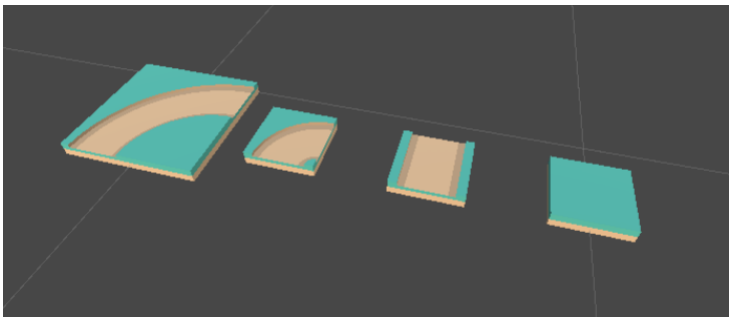

Figura 5.11. Paleta del sistema.

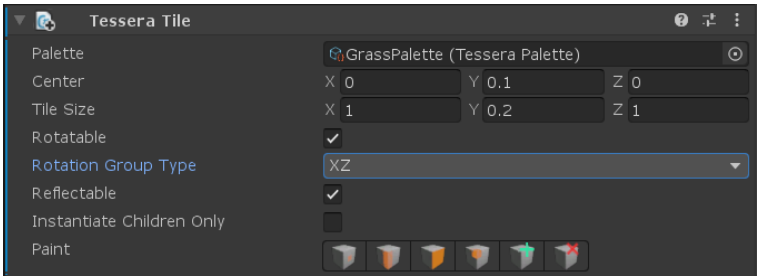

Figura 5.12. Configuració del TesseraTile

Amb aquest script es pot definir:

- ***Palette***: Referència a la paleta que utilitzarà l'objecte.
- ***Center***.: Posició local del centre de l'objecte.
- ***Tile Size***: Mida de la *tile*.
- ***Rotable***: Si es pot rotar la *tile*.
- ***Rotation Group Type***: Permet escollir en quins eixos pot rotar.
- ***Reflectable***: Si es pot reflectir la *tile*.
- ***Instantiate Children Only***: Si en comptés d'utilitzar tota la *tile*, s'utilitzen els seus fills[12].
- ***Paint***: Permet establir els colors de les connexions.

Amb l'opció de *Paint* és pinten totes les *tiles* segons les regles, que vindran definides en una matriu dins la paleta (vegeu Figures 7.13 i 7.14).

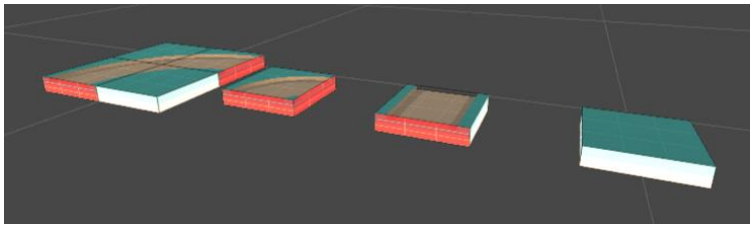

Figura 5.13. Tiles pintades.

[12] Fills: A *Unity* s'anomena fill a una entitat la qual està emparentada amb una altra. A l'estar emparentades qualsevol transformació que s'apliqui sobre el pare s'aplicarà sobre els fills.

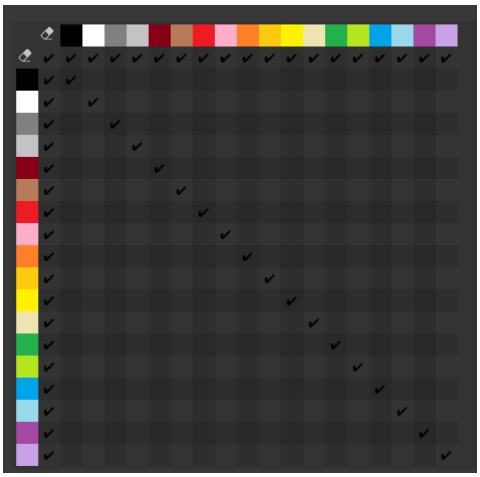

Figura 5.14. Matriu de connexions.

En aquest exemple s'ha definit que cada color només pot anar amb el seu propi color, per tant les cares de color vermell (de camí) només podran anar amb cares de color vermell.

Aleshores s'utilitza un altre objecte de l'escena anomenat *TesseraGenerator* per generar el resultat (vegeu Figura 5.15). En aquest generador es pot configurar:

- ***Center:*** centre del generador.
- ***Size:*** mides del generador.
- ***TileSize:*** mides de les *tiles* del generador.
- ***Backtrack:*** si activa l'algorisme de *backtracking*.
- ***Retries:*** quants cops intentarà generar l'escenari com a màxim.
- ***Sky box:*** permet afegir un cel a l'escena.
- ***Tiles:*** quines *tiles* utilitza per a la generació.

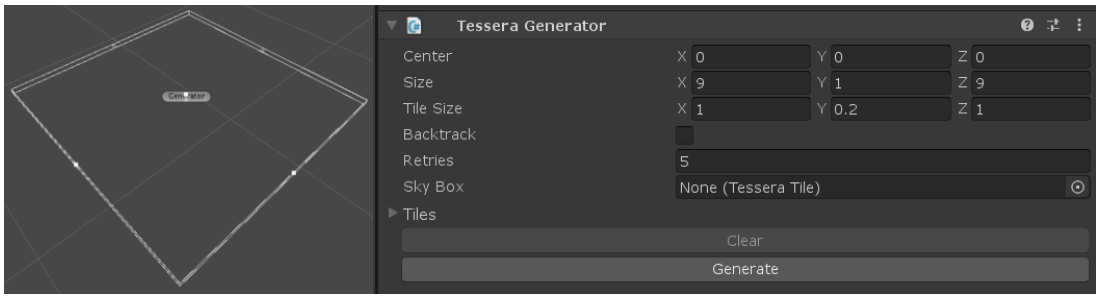

Figura 5.15. Configuració del generador.

Finalment, donant-li al botó de generar, es poden obtenir resultats com els següents. Mostrats a la Figura 5.15.

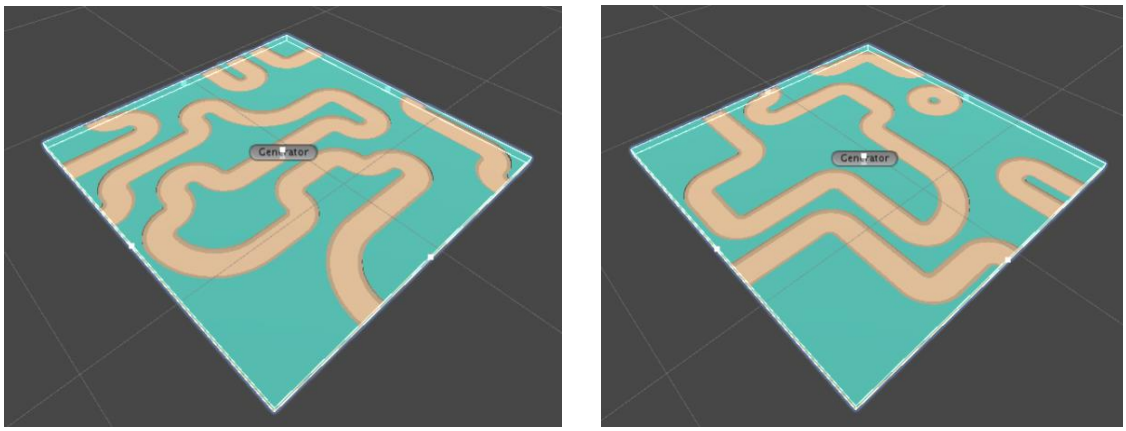

Figura 5.15. Diferents exemples de generacions.

5.2.2 *Tiles* preexistents

Aquest paquet també resulta molt útil, ja que si en la posició del generador ja existeix alguna *tile* creada, aquest la pren en compte a l'hora de generar el resultat (vegeu Figura 5.16).

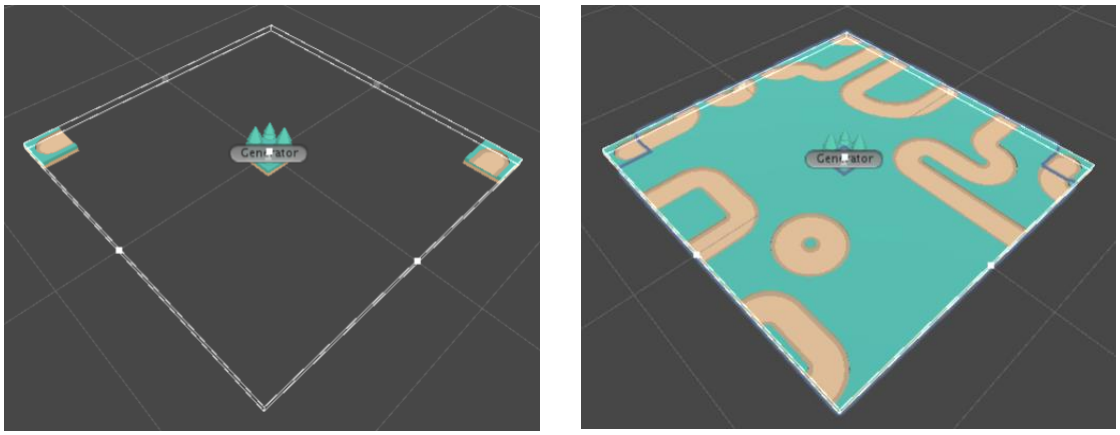

Figura 5.16. Generació amb tiles preexistents, observar com les tiles preexistents continuen en el seu lloc.

5.3 Interconnexió de carrers

Interconnectar carrers significa crear una transició contínua i invisible entre dos carrers separats en l'espai, generant aleshores una translació major a la qual el jugador pensa que ha fet en realitat (vegeu Figura 5.17).

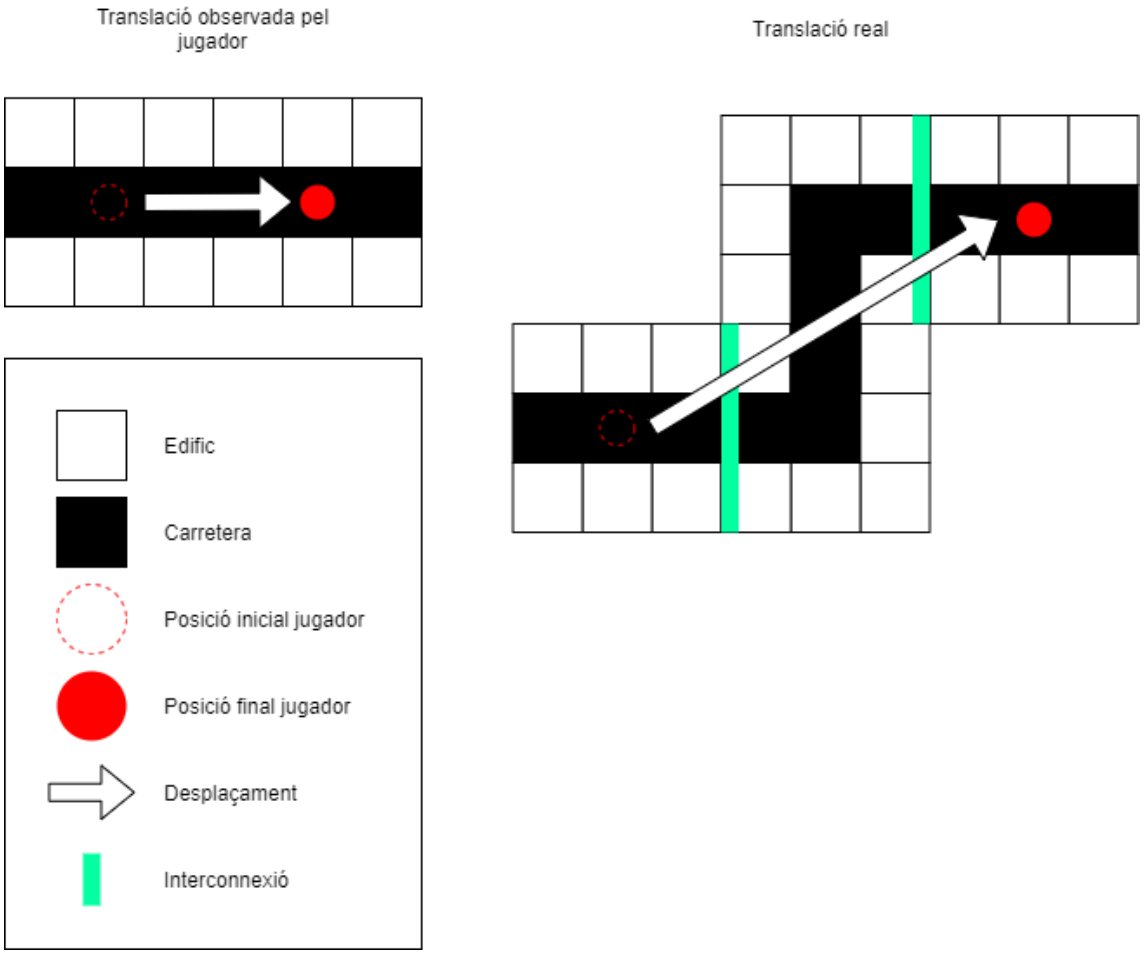

Figura 5.17. Esquema de la interconnexió.

Per aconseguir la interconnexió de carrers s'ha decidit utilitzar portals. La principal raó ha estat per interès i perquè l'altre opció disponible era anar movent els edificis de la ciutat mentre es movia el jugador, la qual cosa podia generar problemes en determinades ocasions, per exemple quan el jugador es troba davant de dos carrers interconnectats cap a la mateixa direcció.

5.3.1 Portals

Els portals permeten portar entitats entre ells fent una transició invisible i continua entre els espais. Per donar-se aquest fet no hi poden haver moviments bruscos i s'ha de poder veure l'altre costat del portal.

Figura 5.18. Captura del videojoc Portal mostrant el funcionament dels portals.

En els videojocs acostumen a tenir un marc per mostrar la zona que afecten (vegeu Figura 5.18), però en el projecte s'ha decidit eliminar-lo, ja que no es vol que el jugador sàpiga a on estan aquestes interconnexions, i si el jugador pogués veure un marc flotant en mig de la ciutat es trencaria la il·lusió.

Tot hi això es mantenen les dues funcionalitats principals, la translació de tota entitat que la travessa i la capacitat de veure el que hi ha a l'altre costat del portal.

5.3.1.1 Translació d'entitats

Per aconseguir una translació d'entitats invisible és necessari que l'objecte és mogui de portal a portal conservant la posició relativa respecte del primer portal però ara respecte el segon. Per exemple, si una entitat no entra pel centre si no per la dreta del portal, aquesta haurà de sortir per la dreta en el següent portal (vegeu Figura 5.19).

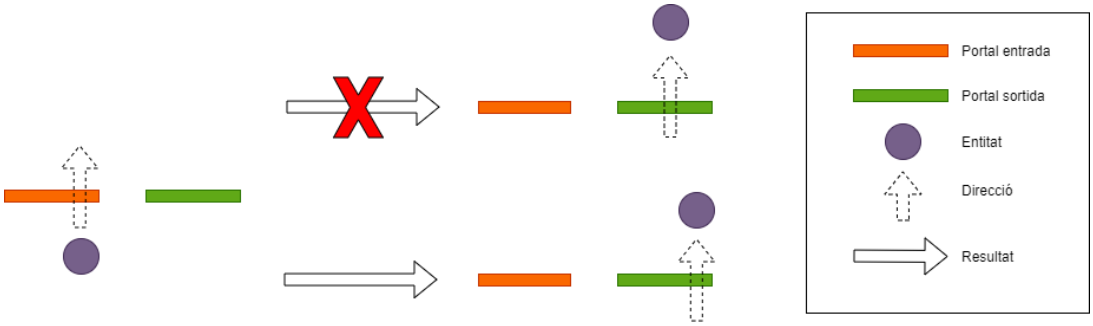

Figura 5.19. Funcionament translació portals I.

A part també és necessari tenir en compte la rotació relativa respecte el portal. No val només en modificar la posició de l'objecte, perquè si no es

té en compte al fer la translació es podria veure un canvi sobtat en la rotació de l'objecte (vegeu Figura 5.20).

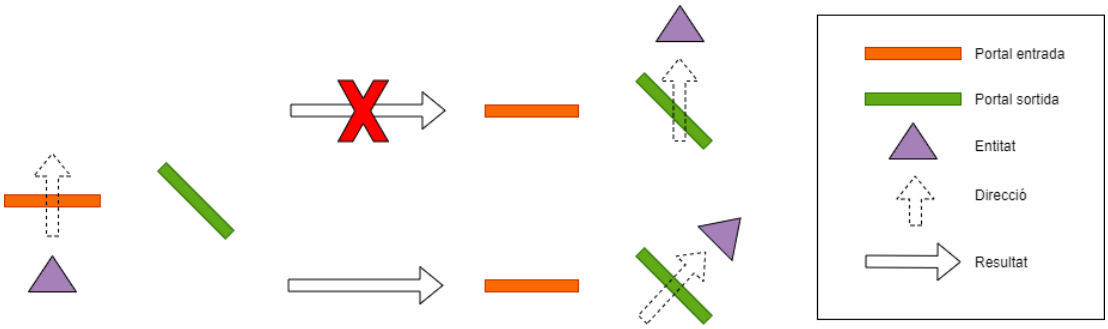

Figura 5.20. Funcionament translació portals II.

Per tant els passos per aconseguir la transició d'entitats entre portals són els següents:

1. Aconseguir la posició i rotació de l'entitat respecte el món.
2. Transformar la posició i rotació de global a local respecte el portal d'entrada.
3. Transformar la posició i rotació de local a global respecte el portal de sortida.

A l'utilitzar el portal de sortida en el tercer pas s'aconsegueix mapejar la posició relativa al portal d'entrada en el de sortida, aconseguint l'efecte cercat.

5.3.1.2 Veure a través del portal

Perquè la il·lusió funcioni, el que es veu en el portal d'entrada ha de ser la vista del que es veu al portal de sortida, tenint en compte la posició i rotació de l'observador.

És a dir, s'ha d'utilitzar les posicions relatives. Per aconseguir l'efecte s'utilitzarà una càmera extra, la qual renderitzarà l'altre costat del portal des de la posició relativa de l'observador a l'altre portal. (vegeu Figura 5.21).

Si no es fa així, es trencaria la il·lusió en el moment que l'observador es poses en moviment, ja que si, per exemple, la càmera extra no agafés la posició relativa, al moure's l'observador la imatge del portal seria sempre la mateixa i semblaria un quadre (vegeu Figura 5.22).

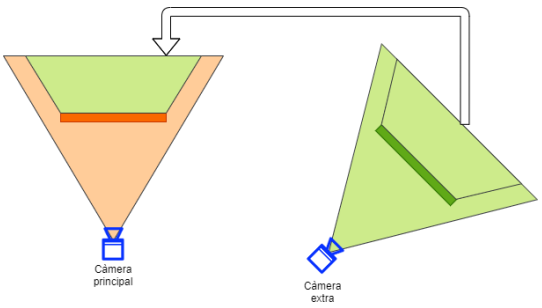

Figura 5.21. Esquema de càmeres portals.

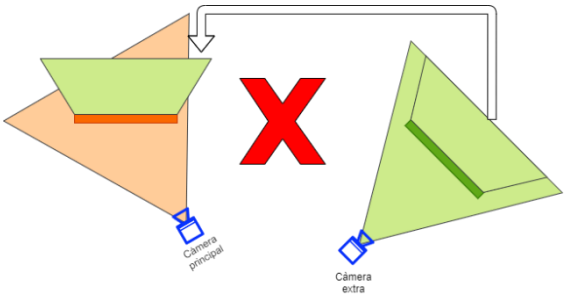

Figura 5.22. Esquema de càmeres portals sense orientar.

Per tant el procés utilitza les mateixes transformacions que la funcionalitat de translació d'entitats, però en comptés d'aplicar la transformació a l'observador, li aplica a una càmera extra. A continuació es poden veure els passos.

1. Aconseguir la posició i rotació de l'entitat respecte el món.
2. Transformar la posició i rotació global a local respecte el portal d'entrada.
3. Transformar la posició i rotació local a global respecte el portal de sortida.
4. Col·locar la càmera extra a la posició i rotació transformada.
5. Renderitzar càmera extra i guardar la textura.
6. Renderitzar càmera principal i aplicar la textura guardada en el portal.

6 Disseny del videojoc

De la mateixa manera que la planificació d'un videojoc és important pel correcte desenvolupament, tenir una estructura firma del disseny també ho és. Tot i això, els videojocs poden evolucionar durant el desenvolupament i noves idees poden aparèixer mentre es crea. Aquestes idees poden millorar el contingut, però de la mateixa manera poden empitjorar el resultat. No tota idea nova pot ser afegida al videojoc, encara que sembli una bona idea, ja que es pateix el risc de perdre l'essència del joc original i tornar-se una idea vaga. A part, afegir noves idees/mecàniques en el videojoc produeix un augment de temps, i la planificació del projecte passa a ser difusa, ja que s'hauria de planificar per idees que encara no existeixen. Per tot això, en aquest projecte s'ha decidit no permetre afegir idees durant el desenvolupament, per tenir un disseny fix i rígid.

6.1 Objectiu del videojoc en el projecte

L'objectiu del videojoc en aquest projecte és utilitzar al màxim la ciutat generada a partir del *WFC* i la propietat d'interconnexió descrita a l'apartat 5.3. Per això el jugador tindrà com a nivell la ciutat generada i s'haurà de moure per aquesta per aconseguir arribar a un punt objectiu des d'un punt inicial.

6.2 Mecàniques

Parlem de mecàniques en els videojocs quan es vol definir què pot fer el jugador en el món del videojoc. Les mecàniques són el que diferencia els videojocs de les altres formes d'entreteniment digital, i en són l'essència de la diversió. Per tant, com millors i més interessants siguin les mecàniques, més divertit serà el joc.

Les mecàniques del projecte han estat creades per adaptar-se sobre la idea principal del projecte, és a dir, la necessitat d'utilitzar una ciutat amb carrers interconnectats com a nivells, de la millor manera possible.

6.2.1 Espai

El videojoc ha de ser capaç de extreure el màxim profit a l'espai, el jugador està jugant en un espai complex i s'ha d'aprofitar. Però, a part de

moure'l d'un punt a un altre de la ciutat per fer perdre al jugador, no és gaire interessant.

Per afegir interès a l'espai se li dóna al jugador un objectiu, un punt d'inici i un final, l'objectiu és anar del punt A al B, i gràcies al fet que la ciutat és generada cada vegada i està interconnectada serà un objectiu prou difícil d'assolir.

6.2.1.1 Problema

Però existeix un problema, com fer entendre al jugador que a la ciutat no tots els carrers porten a on tocaria? A cada partida canvia el nivell, fent que el jugador no pugui aprendre's els carrers i, si la interconnexió és invisible, no podrà entendre que passa.

És important que el jugador vegi que alguna cosa va diferent en el videojoc, ja que si una mecànica existeix, però el jugador no la coneix ni l'utilitza, és el mateix que dir que la mecànica no existeix. Per tant en el projecte es busca que el jugador noti que els carrers no estan connectats de manera normal, però mantenint aquestes interconnexions invisibles.

Una solució és afegir referències a la ciutat, punts emblemàtics els quals permetin al jugador identificar que ja ha passat per aquella zona. Tot i això, omplir el mapa de referències no fa que el jugador sempre noti que la ciutat està interconnectada, ja que podria ser que mai passés per aquestes referències o simplement no els hi donés importància.

6.2.1.2 Solució

La solució que s'ha trobat és modificar la interconnexió de carrers, per fer que tots els carrers interconnectats portin al mateix lloc: el punt de sortida.

El punt de sortida és una referència que sempre veu el jugador a la partida, ja que sempre comença el nivell en aquest punt. Aleshores si el jugador sempre se'l retorna al punt de sortida, notarà segur que passa alguna cosa amb els carrers de la ciutat.

6.2.1.3 Descripció mecànica

En aquest projecte s'aprofita l'espai del joc fent que tots els carrers interconnectats de la ciutat portin a la sortida per fer veure al jugador de manera clara que la ciutat està interconnectada.

6.2.1.4 Nivells creats a mà: Tutorial

En donar-li al jugador un objectiu (anar del punt A al B) se li ha d'explicar a aquest per què el pugui assolir. Per fer-ho existeixen diverses opcions, per exemple escriure un manual o fer uns nivells de tutorial. S'ha decidit utilitzar nivells de tutorial per evitar que el jugador li faci mandra llegir un text fora del joc o molt extens.

Aquests seran nivells reduïts al mínim, els quals requereixen un gran control per part del dissenyador, serà necessari que siguin fets a mà, ja que si s'utilitzés una generació procedimental podria donar-se el cas que el jugador no entengués el que ha de fer a causa de la possible complexitat del nivell.

6.2.1.4.1 Nivell 1.

El nivell 1 permet al jugador entendre quin és l'objectiu principal i com aconseguir-lo. El jugador només té l'opció de moure's cap endavant i girar a la dreta per arribar al punt final. Tot i això si aquest decidís donar la volta i anar enredera, també trobaria el final (vegeu Figura 6.1).

Figura 6.1. Disseny del nivell de tutorial 1.

6.2.1.4.2 Nivell 2

El nivell 2 permet al jugador entendre que la ciutat està interconnectada. Per fer-ho se l'obligarà a passar per un carrer interconnectat segur. El problema és que el jugador podria passar directament pel camí correcte i no utilitzar mai un carrer interconnectat.

La manera de solucionar-ho és crear només dos possibles camins pel jugador, els quals estan interconnectats al principi, però quan el jugador passa a través d'un d'ells, l'altre es desactiva. D'aquesta manera s'obliga el jugador a utilitzar una interconnexió i pot adonar-se del funcionament de la ciutat. Per fer-ho simple s'han afegit dos punts objectius, dels quals el jugador només ha d'arribar a un.

Per tant el jugador entrarà en el nivell i si decideix agafar el gir a la dreta tornarà al principi i la interconnexió de l'esquerra s'eliminarà i podrà accedir a l'objectiu de l'esquerra, en canvi, si decideix girar primer a l'esquerra la interconnexió de la dreta s'eliminarà i podrà accedir a l'objectiu de la dreta (vegeu Figura 6.2).

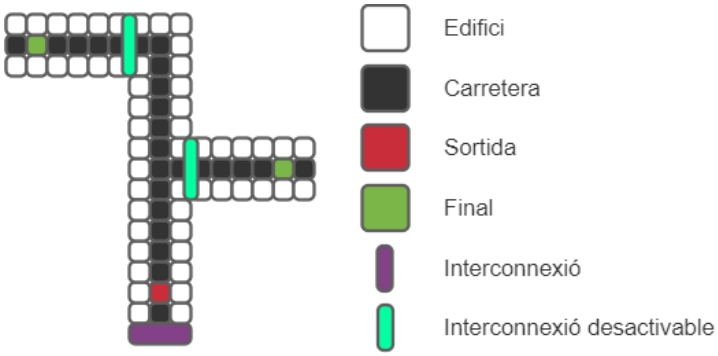

Figura 6.2. Disseny del nivell de tutorial 2.

6.2.2 Moviment del jugador

En l'apartat anterior s'ha explicat l'objectiu del jugador de travessar la ciutat per anar de punt A a punt B. Per a poder aconseguir aquest objectiu, ha de tenir alguna manera de moure el personatge fins al punt B.

La manera de fer-ho serà utilitzant un vehicle, ja que en una ciutat generalment només hi ha vehicles i persones, i per culpa de la dificultat d'animar en 3D persones de manera fluïda, s'ha decidit utilitzar un vehicle.

6.2.2.1 Control bàsic del personatge

El vehicle es controla de la mateixa manera que un vehicle a la vida real, és a dir, no es pot desplaçar cap als laterals directament, sinó que ha d'anar endavant i girar per canviar de direcció.

Una modificació que se li ha fet és no tenir frens ni permetre tirar enredera. Aquesta decisió apareix de la voluntat de fer tornar al jugador al punt de sortida. Si aquest sempre que veiés un carrer interconnectat frenés, fes marxa enrere i continués per un altre camí, mai passaria per una interconnexió.

Finalment s'ha decidit limitar l'angle de gir del vehicle, ja que es vol incentivar al jugador a utilitzar la mecànica de derrapar.

6.2.2.2 Derrapar

El jugador passa la major part del temps conduint per la ciutat, però simplement conduir no és gaire divertit, no té una dificultat real i es fa avorrit ràpidament. Per solucionar-ho, se li permet derrapar. D'aquesta manera la quantitat de moviments que pot fer augmenta i al ser més difícil que conduir, permet que el jugador se senti motivat.

Per acabar-ho d'arrodonir, se li afegeix una petita acceleració al vehicle quan aquest acaba de derrapar, sent aquesta acceleració més llarga i ràpida quan més temps hagi estat el jugador derrapant, recompensant al jugador per haver fet una acció més difícil.

6.2.3 Contrarellotge

El jugador té un objectiu i una manera de complir-lo, però li falta una limitació, ja que no existeix cap dificultat en el videojoc. Ara mateix no s'incentiva al jugador a anar ràpid al punt final. Al no existir dificultat fa que el joc sigui avorrit, ja que no existeix cap repte pel jugador.

El repte que s'ha decidit implementar és un contrarellotge, on el jugador té un temps límit per arribar al punt final. D'aquesta manera no es pot permetre passar sempre pels mateixos llocs i existeix una motivació a anar de pressa. Tanmateix, el joc és massa aleatori per a decidir un temps fix; pot ser que el jugador trobi el camí correcte a la primera i li sobri temps, però també existeix la possibilitat que s'equivoqui moltes vegades i se senti frustrat per què ha tingut mala sort.

La solució és lligar el temps que té l jugador a les seves accions. Si el jugador juga bé augmentarà el temps disponible. D'aquesta manera es

relaciona l'habilitat del jugador amb la dificultat del joc, i per tant si el jugador perd un nivell és culpa seva i no de la mala sort.

Hi ha dues accions que pot fer el jugador per aconseguir més temps, derrapar i agafar monedes.

6.2.3.1 Derrapar

El jugador ja està incentivat de base a derrapar, ja que derrapar dóna l'acceleració que li permet anar més de pressa, però si a més a més afegeix més temps per arribar a l'objectiu passa a ser un moviment molt útil pel jugador.

A part, en requerir més habilitat a l'hora d'executar-lo, si el jugador l'efectua de manera correcta, rebre temps és una petita recompensa que el motiva.

6.2.3.2 Monedes

Ara mateix la ciutat està buida, només existeix el vehicle del jugador, i per tant és avorrida. Per això s'ha decidit afegir monedes en el videojoc. Aquestes tenen dues funcions, la primera afegir més vida a la ciutat, ja que uns objectes mínimament animats fan que hi hagi més objectes en moviment per la ciutat; i la segona permet al jugador interactuar més amb el joc.

Tot i això no es poden afegir monedes sense donar-li una utilitat. Si aquestes no serveixen de res, el jugador no està motivat a agafar-les i passen a ser una molèstia.

La utilitat de les monedes en aquest videojoc és aconseguir més temps, una manera simple pel jugador per sumar una petita quantitat de temps i motivar-lo a agafar-les.

6.2.3.3 Sistema de combo

Finalment per acabar de motivar el jugador a jugar bé, és a dir sense colpejar els edificis de la ciutat i anar de pressa cap a l'objectiu, s'ha afegit un sistema de combo, el qual permet multiplicar la quantitat de segons guanyats per cada acció.

Aquest combo va augmentant fins a un límit a mesura que el jugador fa accions correctes i es perd quan el jugador xoca contra una paret o passa massa temps sense fer una acció. D'aquesta manera es motiva al jugador a no xocar i continuar sempre en moviment, ja que té una penalització si no ho fa.

6.3 Funcionament del videojoc

Ajuntant totes les mecàniques de l'apartat anterior es pot concretar com es jugarà i com seran les partides. En resum, en aquest videojoc el jugador portarà un vehicle per la ciutat d'un punt A a un punt B, tindrà un temps determinat el qual podrà augmentar derrapant i agafant monedes i, finalment les interconnexions faran que torni al principi per generar confusió.

A continuació és pot observar el diagrama de flux d'una partida i com funciona (vegeu Figura 6.3 i Figura 6.4).

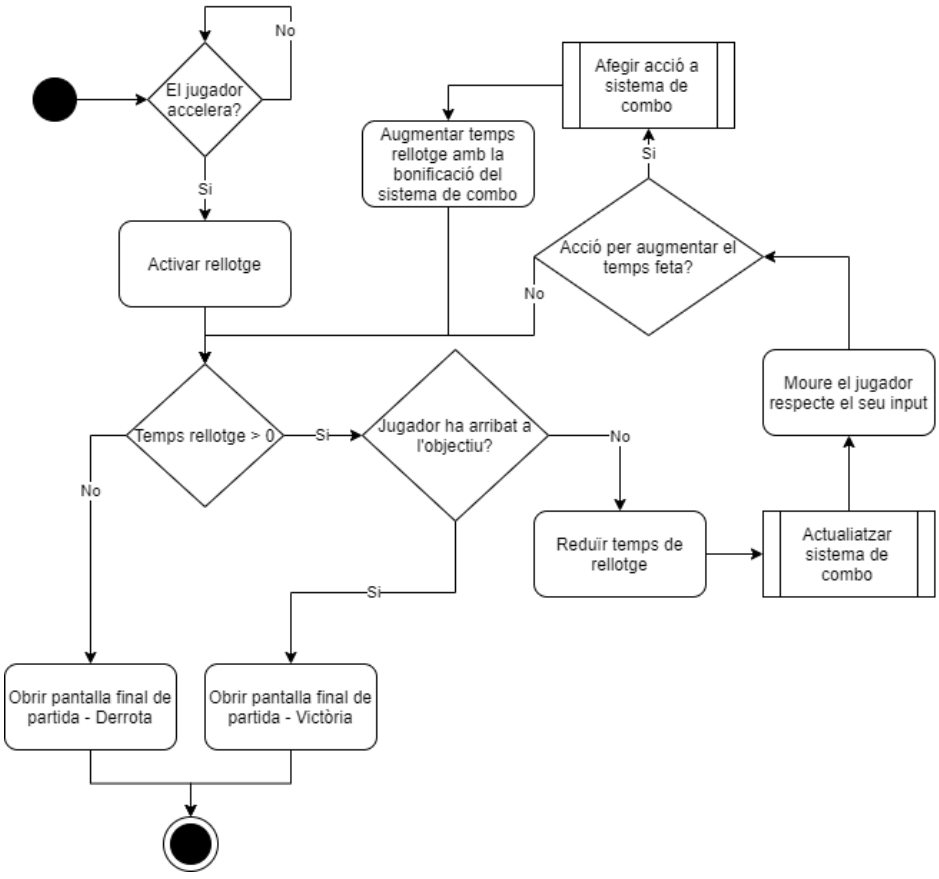

Figura 6.3. Diagrama de flux.

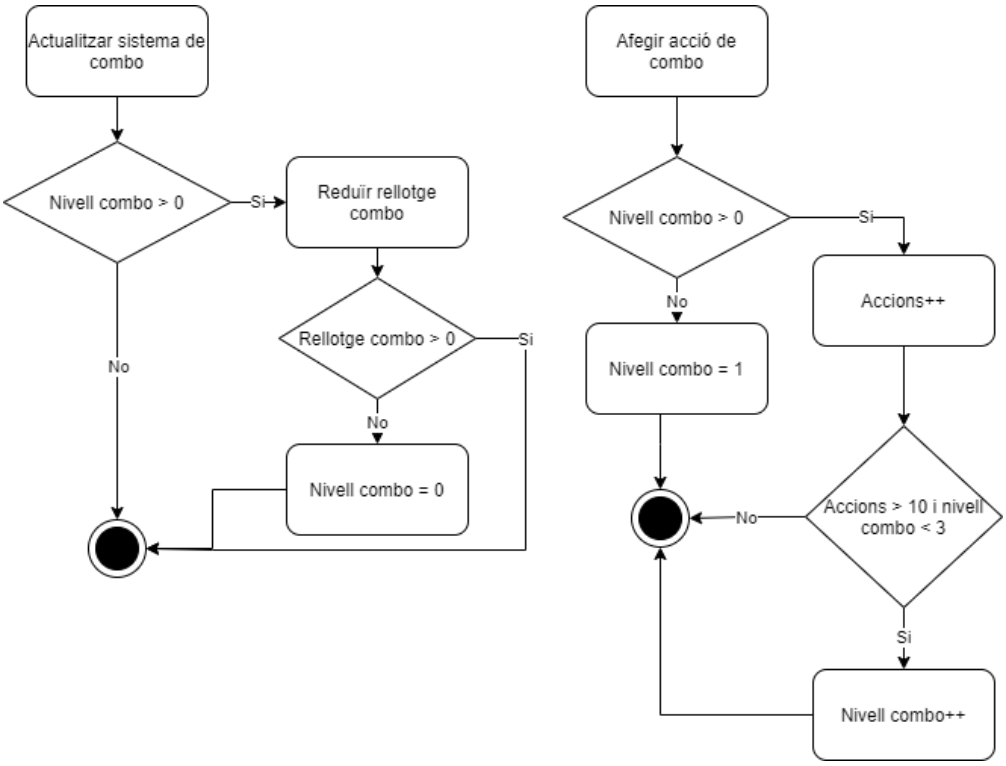

Figura 6.4. Continuació diagrama de flux.

6.4 Evitar frustració

Per culpa de la naturalesa dels nivells hi ha una alta probabilitat que el jugador senti frustració amb el joc en equivocar-se triant sempre el mateix camí. Això és perquè la ciutat és molt regular, i per tant, no hi ha referències amb les quals el jugador es pugui orientar i no té manera de saber per on ha passat anteriorment.

Una possible solució és afegir una gran quantitat d'edificis variats per donar vida a la ciutat i fer que el jugador pugui identificar si ja ha passat o no. El problema és que aquests estan fora del punt de vista principal del jugador, que està mirant sempre la carretera i només veuria els edificis que té de cares. Per tant és molt probable que no s'hi fixés, a més del fet que augmenta el nombre de models d'edificis necessaris.

Per a solucionar el problema s'ha decidit mostrar al jugador per on ha passat. En fer-ho el jugador pot guiar-se per escollir el camí correcte i s'elimina la probabilitat que s'equivoqui sempre de camí per mala sort. Per mostrar-li el camí s'ha modificat un aspecte del joc i s'ha inclòs un de nou: les monedes ja no desapareixen en ser agafades i el vehicle deixa marques de rodes.

6.4.1 Monedes

En els videojocs normalment quan s'agafa una moneda aquesta s'elimina perquè ja no hi ha gens d'interès perquè hi continuï. En canvi, en el projecte, quan el jugador n'agafi, aquesta canvia de color a un blau transparent per mostrar al jugador que ja ha passat per allà. Per tant el jugador pot seguir les monedes que no estan agafades per explorar parts de la ciutat per les quals encara no hi ha passat.

Figura 6.5. Moneda sense agafar i agafada.

6.4.2 Marques de les rodes

Fer que el vehicle deixi un rastre permet que el jugador pugui veure clarament per on ha passat i per mostrar-li que sempre passa pel mateix lloc de sortida. Això és important per què, si no, el jugador podria pensar que ha passat per una altra sortida diferent de la que ha començat. En tenir les seves pròpies marques, veu clarament que és el mateix punt.

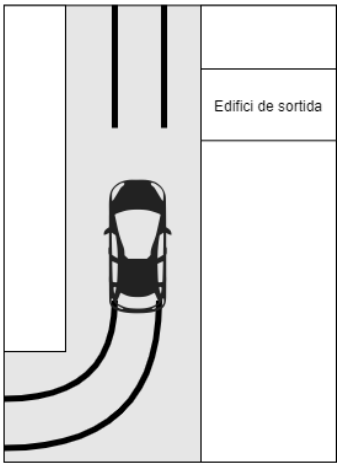

Figura 6.6. Esquema de les marques de les rodes.

6.5 Accions del jugador

Els videojocs es basen en la interacció amb el jugador, per tant és important definir com ho fa. A continuació es defineixen les accions que pot fer el jugador en el joc, les quals estan diferenciades per accions de menú i accions de partida.

6.5.1 Accions de menú.

Definim com accions de menú les accions que pot fer el jugador quan està en una pantalla fora de la partida, com per exemple el menú principal o el de pausa.

Els menús del videojoc se centren a utilitzar botons amb diferents resultats depenent de l'opció que simbolitzen. A continuació s'explica la funcionalitat compartida per a cada botó.

- **Seleccionar**: permet marcar el botó com a pròxim a utilitzar.
- **Utilitzar**: permet executar la funcionalitat pròpia de cada botó.
- **Canviar**: permet seleccionar un altre botó.

Les funcionalitats pròpies que poden tenir els botons són:

- **Canviar de pantalla**
- **Començar la partida**
- **Anar al següent nivell**
- **Sortir del joc**
- **Modificar el volum de la música**
- **Modificar el volum dels efectes de so**
- **Mostrar/amagar els FPS del joc**
- **Mostrar el temps en segons o mil·lisegons**

6.5.2 Accions de partida

Definim com accions de partida les accions que el jugador pot executar dintre el nivell mentre juga la partida. Les accions que pot fer el jugador durant la partida són:

- **Accelerar**: permet augmentar la velocitat frontal del vehicle.
- **Girar**: permet modificar la rotació del vehicle.
- **Derrapar**: permet executar la mecànica de derrapar.
- **Aturar/Continuar el joc**: permet parar el joc o continuar la partida.

6.6 Reptes i objectius del jugador

A l'apartat de mecàniques s'ha definit per primer cop l'objectiu del jugador (vegeu Apartat 6.1). A continuació es mostren quins reptes ha de complir el jugador per poder aconseguir l'objectiu (vegeu Figura 6.7)

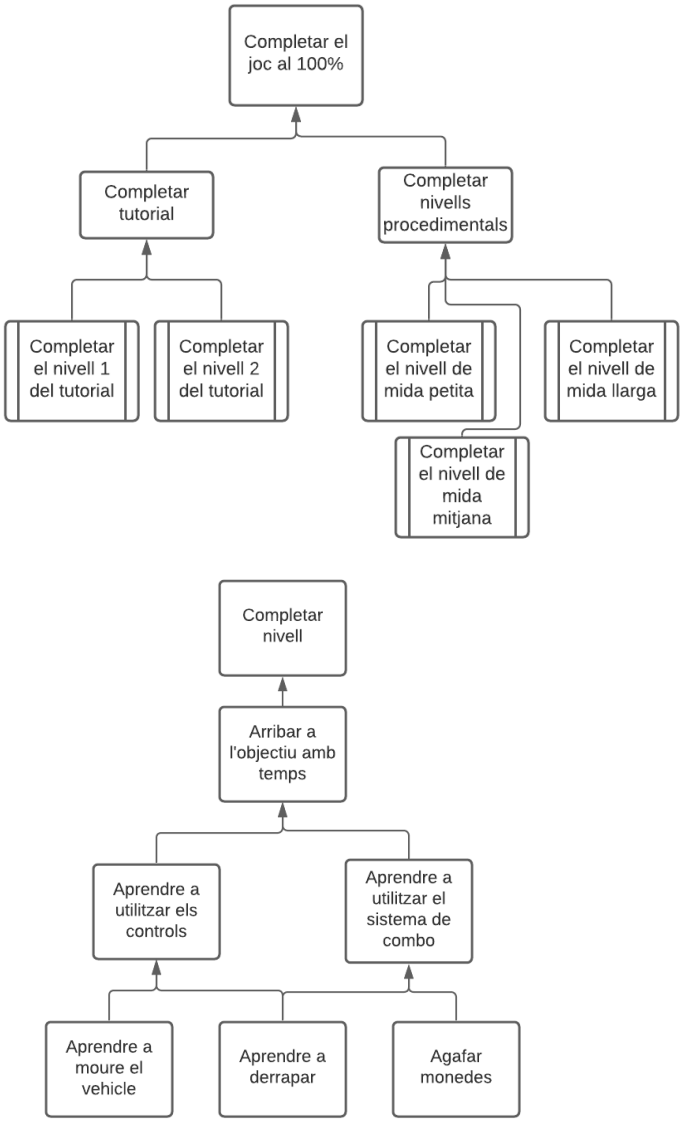

Figura 6.7. Diagrama de reptes per completar un nivell.

Es considera l'objectiu del joc completar-lo al cent per cent, al complet. Això és així per culpa que el joc no limita a quins nivells es pot accedir. El jugador des del principi pot entrar a qualsevol nivell i els pot fer en qualsevol ordre. A part, com que no existeix una pantalla final dient que ha completat el joc, és difícil definir un objectiu real. Això passa com a la majoria de jocs arcade o casual, els quals no tenen un final com a tal.

Aquests jocs tenen com a objectiu alenar al jugador a millorar la seva pròpia puntuació.

Per tot això, en el projecte s'ha afegit una pantalla de puntuació final, la qual avalua al jugador respecte que tan bé ha jugat, que es pot veure en el següent apartat.

6.7 Avaluació del jugador

Com s'ha explicat a l'apartat anterior, se li ha de mostrar al jugador quan ha jugat bé i quan no. D'aquesta manera el jugador pot tenir com a objectiu del joc millorar, ja que si no se li defineix un objectiu clar, aquest no tindrà motivació per jugar.

Per avaluar al jugador es recolliran dades de la partida mentre juga i s'utilitzaran per aconseguir valorar la puntuació final. Les dades que s'avaluaran són:

- **Temps restant sobre temps inicial**: proporció de temps del jugador entre els segons amb els quals ha arribat al final i els segons que tenia quan ha començat la partida per primer cop.
- **Col·lisions**: cops que el jugador ha xocat amb una paret.

Per altra banda també se li mostrarà al jugador amb quantes monedes ha arribat i quantes vegades ha completat correctament l'acció de derrapar. Aquestes accions no s'utilitzen per avaluar, ja que aquestes accions afecten directament al temps restant del jugador. Per tant ja estan afectant el resultat final de manera indirecta.

Finalment, en comptes d'un sistema que avaluï numèricament al jugador, s'ha decidit utilitzar un sistema que posa una nota en format de lletres: F, C, B, A i S; ordenades de pitjor a millor. Això és a causa del fet que, per un jugador, és molt més fàcil entendre que tan bona és la seva puntuació. Per exemple, si a un jugador se li posa una puntuació de 164, no pot saber si és una bona puntuació o no, i no pot calcular el màxim de puntuació d'un nivell ja que el càlcul és gairebé impossible perquè els nivells són generats de manera procedimental i per tant, canviants, i no es pot saber fàcilment quina és la millor manera de conduir per la ciutat. Això, sumat a què és només per la puntuació final, es considera que és

un cost massa gran per al desenvolupament i per tant s'utilitza el sistema de lletres.

6.8 Interfícies del joc

A continuació es mostraran el disseny per a cada pantalla del videojoc amb la informació que es vol mostrar al jugador.

6.8.1 Pantalla de recomanació de comandament

Aquest projecte està pensat per ser jugat amb comandament, perquè el control és més precís. Tot i això, existeixen jugadors en el mercat els quals no disposen d'un comandament per jugar, per això el joc permet la utilització del comandament o del teclat, a gust del jugador. Tot i això, és important mostrar al jugador com es considera que funciona millor el joc. Per això, al principi del joc es mostrarà la pantalla de la Figura 6.8 recomanant al jugador utilitzar el comandament.

Figura 6.8. Disseny de la pantalla de recomanació de comandament.

6.8.2 Pantalla de títol

La pantalla de títol és la primera cosa que veuran els jugadors del joc (és veritat que durant l'inici s'haurà mostrat el logotip d'*Unity*[13] i la recomanació de controls, però això és informació secundària). En la pantalla de títol el jugador veu per primer cop el logotip del joc (vegeu Figura 6.9).

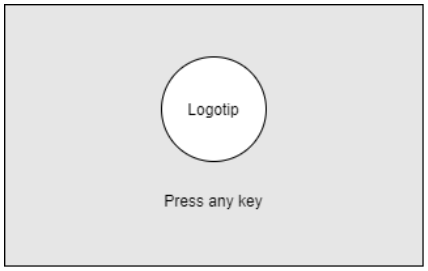

Figura 6.9. Disseny de la pantalla de títol.

[13] *Unity* obliga a posar el seu logotip a la versió gratuïta.

6.8.3 Menú principal

El menú principal del joc és el que mostra al jugador què pot fer en el joc. Des d'aquest es pot anar al selector de nivells, al menú d'opcions o sortir del joc (vegeu Figura 6.10).

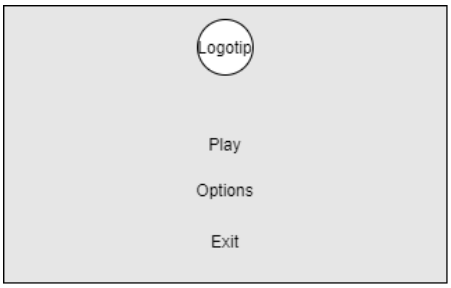

Figura 6.10. Disseny del menú principal.

6.8.4 Selector de nivells

Des del Selector de nivells el jugador pot decidir quin nivell jugar. Ara mateix estan limitats a: dos nivells de tutorial i tres nivells procedimentals, en els quals només es varia la mida de la ciutat.

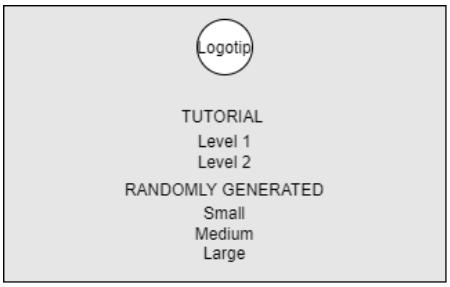

Figura 6.11. Disseny del selector de nivells.

6.8.5 Menú d'opcions

Des del menú d'opcions el jugador pot modificar la configuració del joc: es pot modificar el volum de la música, el volum dels efectes de so, si es vol mostrar o no els *FPS* mentre juga, i mostrar els mil·lisegons en el temps restant que el jugador té per arribar al final.

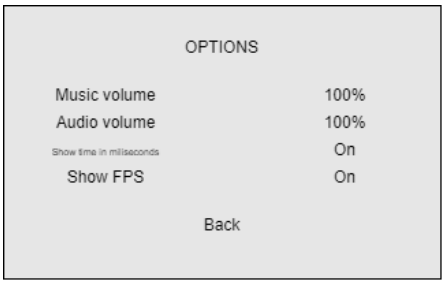

Figura 6.12. Disseny del menú d'opcions.

6.8.6 *HUD*

HUD (*Head Up Display*), és refereix a la informació de feedback que donem al jugador mentre juga, i com li donem. En el projecte s'ha decidit mostrar al jugador el temps restant que té per arribar al punt final, quantes monedes té, mostrar els *FPS* (si s'han activat en el menú d'opcions) i el sistema de combo, el qual només es mostrarà per pantalla si existeix algun combo actiu en aquell moment.

Figura 6.13. Disseny del HUD sense combo actiu.

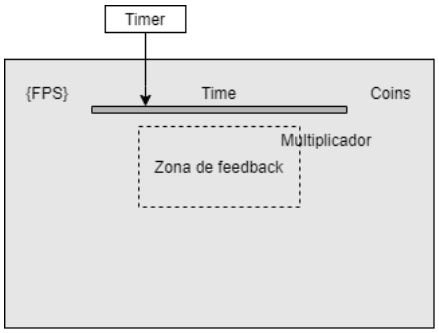

Figura 6.14. Disseny del HUD amb combo actiu.

6.8.7 Menú de pausa

En el menú de pausa el jugador podrà continuar el joc, reiniciar el nivell i tornar al menú principal.

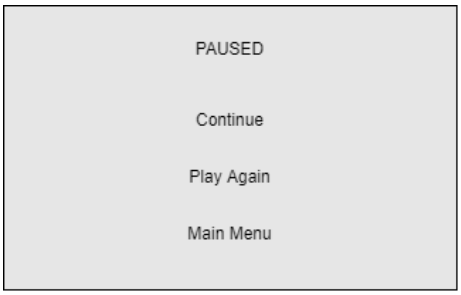

Figura 6.15. Disseny del menú de pausa.

6.8.8 Pantalla final de partida

En aquesta pantalla es mostrarà el resultat de l'avaluació del jugador. Per tant, haurà de ser una pantalla clara i fàcil d'entendre. Des d'aquesta pantalla el jugador podrà tornar al menú principal, tornar a jugar el nivell o passar al següent nivell.

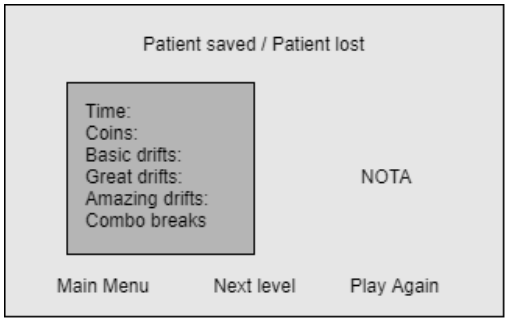

Figura 6.16. Disseny de pantalla final de partida.

6.9 Narrativa

La narrativa d'aquest videojoc no és un dels punts importants. Al ser un videojoc *casual*, no existeix la necessitat que sigui molt complexa i profunda. Tot i això, la narrativa és l'encarregada de donar la motivació del jugador per anar del punt A al punt B de la ciutat. A part, també ha de ser capaç d'explicar per què la ciutat va canviant i està interconnectada.

La narrativa del videojoc és la següent: **el jugador és el conductor novell d'una ambulància, el qual acaba de recollir un pacient i l'ha de portar fins a l'hospital, però es perd pel mapa perquè encara no coneix la ciutat.**

Aquesta narrativa és simple perquè utilitza elements que qualsevol persona podrà entendre, és clara i dóna certa urgència a arribar al punt final, que és el que es buscava des d'un principi.

Tot i això, no és perfecte, ja que no és capaç d'explicar el perquè derrapar per la ciutat dóna més velocitat o per què el temps per arribar a la ciutat augmenta quan el jugador agafa monedes, ni perquè les monedes hi són, etc. Però, pel contrari no cal que sigui perfecta. Mirem per exemple el famós videojoc *Super Mario Bros* de *Nintendo*. La història d'aquest videojoc se centra en el segrest de la princesa del regne Xampinyó i el jugador l'ha d'anar a rescatar i res més, no explica per què el protagonista té vides, perquè hi ha monedes pel mapa, per què quan el jugador perd, reviu, etc.

La narrativa en aquest estil de videojocs no busca explicar per què passa tot en el món del videojoc, només busca donar un pretext per donar al jugador l'objectiu, i per això la narrativa explicada anteriorment és funcional, i per tant, vàlida.

6.10 Nom del videojoc

Un cop explicada la narrativa, es necessita triar un nom pel videojoc, que ha d'estar relacionat amb la narrativa i amb les mecàniques del joc. A part, no pot ser un nom molt semblant a un altre videojoc, ja que es podrien confondre. Amb totes aquestes consideracions, el nom final escollit és el de ***Lost Ambulance***.

6.11 Estil artístic del joc

Per acabar falta trobar un estil artístic el qual lligui amb les mecàniques del joc, sigui fàcil de produir, i que alhora es vegi professional, hagi de tenir en compte la narrativa i hagi de transmetre el caràcter del joc.

6.11.1 Requeriment: Sense ombres

Tot i això, l'art del videojoc en concret ha de tenir una particularitat: no hi poden haver ombres. Això és a causa que poden existir dos carrers interconnectats amb diferents orientacions, i per tant les ombres tindrien diferents direccions depenent d'en quina posició estigui el jugador. Com que es busca que la interconnexió sigui invisible, la manera més fàcil de solucionar-ho és eliminar el problema, és a dir, eliminar les ombres.

6.11.2 Requeriment: Sense núvols

L'art del videojoc no pot tenir núvols, això és a causa que en travessar portals aquests es podrien veure tallats. Aleshores, de la mateixa manera que amb el problema de les ombres, la solució més fàcil és eliminar els núvols directament.

6.11.3 Referències estètiques

A continuació es poden observar diferents videojocs els quals tenen un estil artístic semblant a l'estil objectiu d'aquest projecte.

Figura 6.17. Captura de Katamary Damacy

Figura 6.18. Captura de You Suck At Parking

Figura 6.19. Captura de Art of Rally

Figura 6.20. Captura de Nitroneers

6.12 Disseny de personatges

En el videojoc només hi ha un personatge. Això és a causa que, per l'estil del joc i les mecàniques, només es requereix el personatge que porta el jugador, una ambulància.

6.12.1 Ambulància

L'ambulància seguirà l'estil artístic explicat a l'Apartat 6.11 per a aconseguir una coherència entre tots els elements del videojoc. És a dir, seguirà un estil *low poly,* el qual es defineix en tenir una quantitat reduïda de polígons. A continuació es pot observar, a la Figura 6.21, una referència pel disseny de l'ambulància i el resultat final.

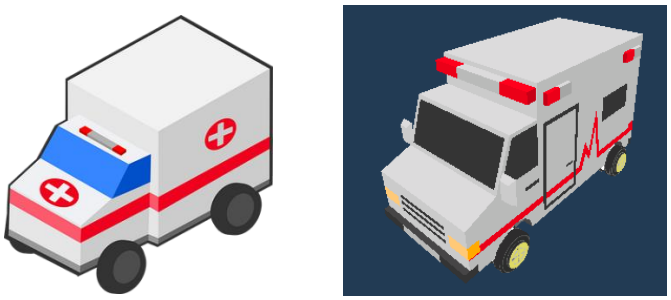

Figura 6.21. Referència estètica i disseny final ambulància.

6.13 Disseny d'objectes

En el videojoc només hi ha un objecte amb el qual pot interactuar el jugador, les monedes.

6.13.1 Monedes

Pel disseny de les monedes del videojoc s'ha decidit utilitzar la forma bàsica de les monedes de la majoria de videojocs, és a dir, un prisma circular amb poca altura i, per aconseguir que tingués més sentit amb la funcionalitat, tindran unes marques com si fossin les agulles d'un rellotge, ja que les monedes en el videojoc aconsegueixen més temps. Aquest disseny seguirà l'estil artístic explicat a l'apartat 6.11. A continuació es pot observar, a la Figura 6.22, una referència pel disseny de l'ambulància i el resultat final.

Figura 6.22. Referència estètica i disseny final monedes.

6.14 Llista d'elements a desenvolupar

Els elements a desenvolupar del projecte es poden dividir en diferents apartats: elements d'interfícies, personatges, objectes i música i efectes de so.

6.14.1 Elements d'interfícies

Els elements d'interfícies requerits pel projecte són:

- **Logotip del joc:** signe gràfic identificador del videojoc.
- **Esquema de controls**: imatge per mostrar al jugador els controls del joc

6.14.2 Personatges

Els personatges requerits pel projecte són:

- **Ambulància:** personatge principal del videojoc.

6.14.3 Objectes

Els objectes requerits pel projecte són:

- **Monedes:** col·leccionable del joc.

6.14.4 Música i efectes de so

La música i els efectes de so requerits pel projecte són:

- **Música:** banda sonora del videojoc.
- **Accelerar:** efecte de so per quan l'ambulància acaba de derrapar.
- **Moneda:** efecte de so per quan s'agafa una moneda.
- **Seleccionar un botó:** efecte de so per quan se selecciona un botó.
- **Utilitzar un botó:** efecte de so per quan s'utilitza un botó.
- **Victòria:** efecte de so per quan el jugador guanya la partida.
- **Derrota:** efecte de so per quan el jugador perd la partida.
- **Col·lisió:** efecte de so per quan l'ambulància col·lideix amb una paret.

7 Implementació

En aquest apartat s'explicarà com s'han implementat els elements més interessants del videojoc.

7.1 Art del videojoc

En l'apartat 6.9 s'ha explicat l'estil d'art que es buscava pel videojoc i buscant per internet, es va trobar la pàgina web https://syntystore.com, la qual ven un paquet d'art que es diu *Simple Town* el qual encaixa perfectament pel videojoc.

Figura 7.1. Paquet d'art Simple Town I.

Figura 7.2. Paquet d'art Simple Town II.

A part el paquet d'art conté una ambulància que es pot utilitzar com a personatge principal. L'única cosa que li falta és un model per les monedes del videojoc.

7.1.1 Art de les monedes

Seguint el disseny explicat a l'apartat 6.13.1, el model resultant del disseny de les monedes es pot observar a la Figura 7.3.

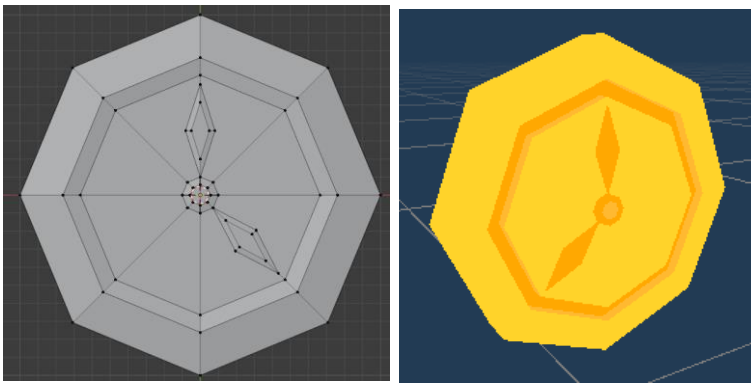

Figura 7.3. Model de les monedes amb i sense textura.

7.2 Generació de la ciutat

La generació del plànol de la ciutat serà feta a partir de l'algorisme *WFC.* Per aconseguir-ho s'ha decidit utilitzar un paquet de l'*Asset Store*[14] de *Unity* anomenat *Tessera* (explicat a l'apartat 5.2). S'ha decidit utilitzar-lo per simplificar la feina i aconseguir dedicar més temps a què el resultat final fos professional.

7.2.1 *Tiles* compostes

En el projecte el paquet de *Tessera* no utilitza *tiles* d'un sol objecte com les ensenyades en l'exemple, si no que utilitza *tiles* formades de 9 *subtiles*. Pel *Tessera* aquestes *subtiles* són invisibles ja que ell només treballa amb la *tile* composta, però en transformar de plànol a models s'utilitzen les *subtiles*. S'ha decidit utilitzar aquest sistema per què permet tenir més varietat.

Es pot aconseguir varietat ja que si s'utilitzen *tiles* simples s'ha de generar variacions una a una, per exemple per fer què les tiles amb forma de L siguin diferents entre si, s'ha de crear un conjunt de models per les tiles en forma L i després escollir-ne un aleatori, per tant, cada variació s'ha de generar a mà.

[14] Asset Store: biblioteca de paquets lliures i comercials creats tant per *Unity Technologies* com per membres de la comunitat.

En canvi, utilitzant un conjunt de models per cada *subtile*, es pot generar escollint un conjunt aleatori de models per a cada *subtile*. Generant així més varietat ràpidament.

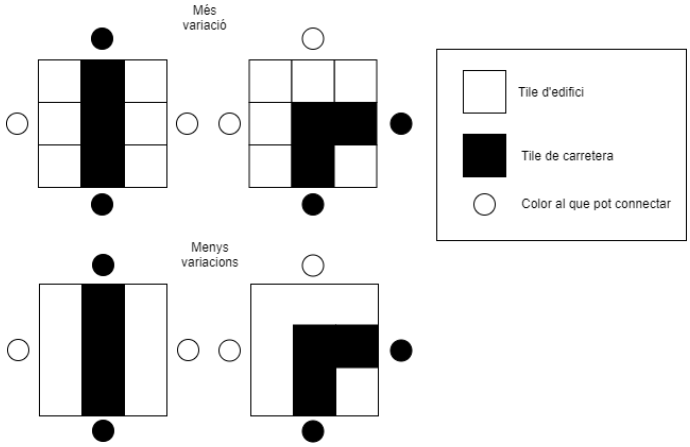

Figura 7.4. Simplificació a l'utilitzar tiles compostes.

7.2.2 Generació de la ciutat – ComplexGenerator.cs

A continuació s'explicarà com es genera la ciutat pas a pas en el projecte aprofitant les tiles compostes, el paquet de *téssera* i el script ComplexGenerator.cs, en el qual existeix tota la lògica de la generació.

7.2.2.1 Paleta

La paleta utilitzada en aquest projecte és la següent:

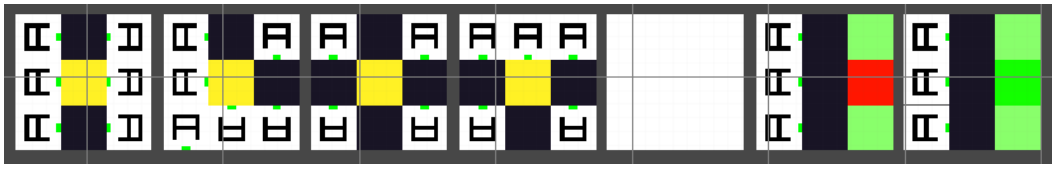

Figura 7.5. Paleta utilitzada en la generació de la ciutat del projecte.

Totes les *tiles* estan formades per *subtiles* excepte la cinquena *tile*. Aquesta és tota blanca i simbolitza que no hi va res, per tant, quan es passi de plànol a models, s'ignora.

Les *tiles* compostes estan formades per les següents *subtiles*:

- ***Subtile* d'edifici**: s'utilitza per marcar que en aquesta posició i va un edifici i amb quina orientació, sent el punt verd on hi haurà la porta.

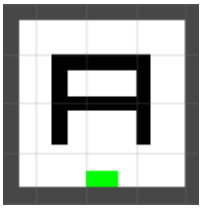

Figura 7.6. Subtile d'edifici.

- ***Subtile* de carretera**: s'utilitza per marcar que en aquesta posició i orientació hi va una peça de carretera.

Figura 7.7. Subtile de carretera.

- ***Subtile* de carretera d'or**: s'utilitza per marcar que en aquesta posició i orientació hi va una peça de carretera amb monedes.

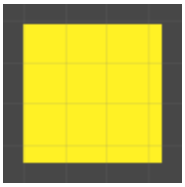

Figura 7.8. Subtile de carretera d'or.

- ***Subtile* d'inici**: s'utilitza per marcar que en aquesta posició i orientació hi va l'edifici de sortida.

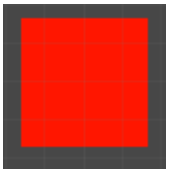

Figura 7.9. Subtile d'inici.

- ***Subtile* de final**: s'utilitza per marcar que en aquesta posició i orientació hi va l'edifici objectiu del jugador.

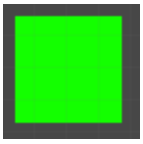

Figura 7.10. Subtile de final.

- ***Subtile* de decoració**: s'utilitza per marcar que en aquesta posició i orientació hi va la peça que es col·loca al costat de la sortida i el final per decorar.

Figura 7.11. Subtile de decoració.

7.2.2.2 Regles/Colors

Els colors es poden reduir a dos, un per indicar que hi ha connexió cap aquella direcció i l'altre per indicar que no. Per això, la matriu del generador és la següent:

Figura 7.12. Matriu de colors del projecte.

7.2.2.3 Generació de vores

Per començar la generació, el primer que es fa és generar unes vores al costat del generador. Això permet tancar la ciutat, ja que, si no es creessin, la ciutat podria quedar amb carrers que porten al no-res quan es generés (vegeu Figura 7.13). En aquest pas només es posen les vores (vegeu Figura 7.14).

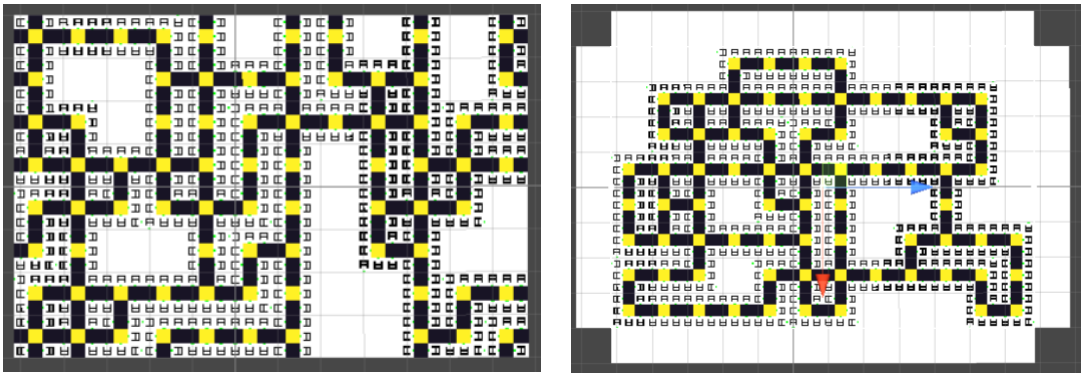

Figura 7.13. Exemple de generació sense i amb vores.

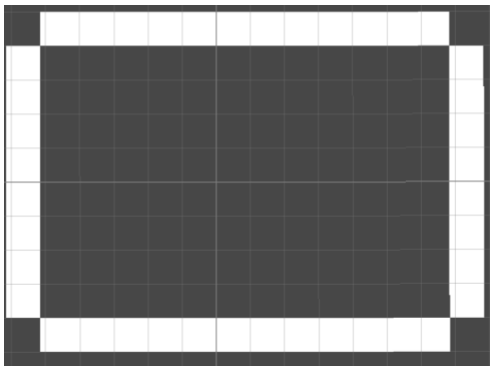

Figura 7.14. Resultat del primer pas de la generació.

En el codi es troba a la funció *SpawnBorders,* la qual genera aquestes dues columnes i files de *tiles*.

7.2.2.4 Generació d'inici i final

A continuació es col·loquen les *tiles* d'inici i final. Aquestes existeixen en la paleta, però el generador no les pot utilitzar. Això és a causa que només es busca un punt d'inici i un de final, per tant, si el generador les pogués col·locar lliurement, en posaria més d'una.

La col·locació d'aquests punts va lligada a les mides del generador, aquest sempre les col·locarà cap a les vores de la distància més llarga entre l'amplada i l'altura (vegeu Figura 7.15). El punt de sortida sempre el col·locarà tocant una de les vores més curtes, per evitar que el jugador pugui travessar el portal principal en contra de la direcció esperada (vegeu Apartat 7.3).

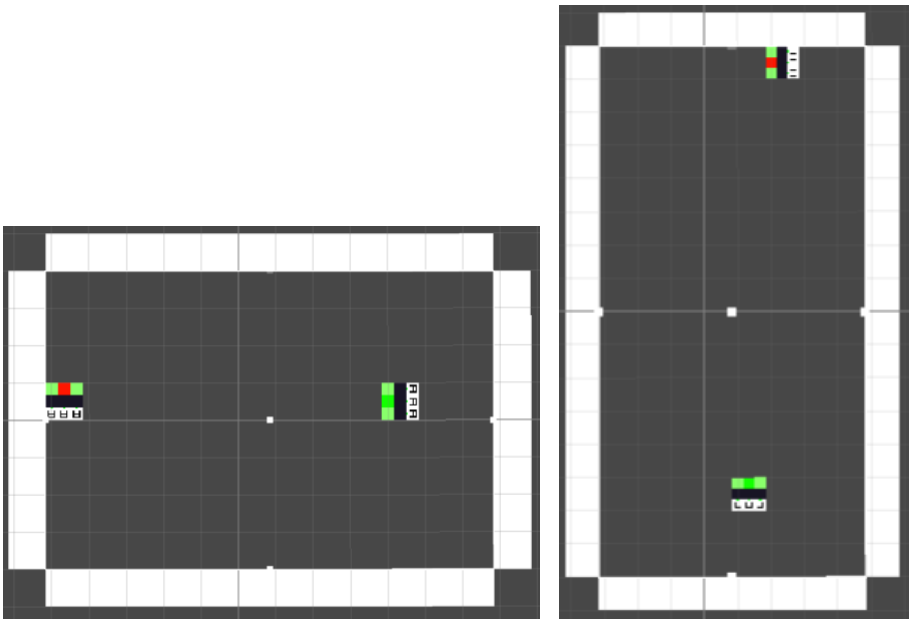

Figura 7.15. Col·locació del punt d'inici i final amb diferents mides de generació.

En el codi és crida la funció *SpawnStartAndFinish,* la qual calcula una posició aleatòria pel punt d'inici i final, una rotació vàlida i els instancia en el generador amb la rotació i la posició calculades.

7.2.2.5 Generar el plànol de la ciutat

Arribats a aquest moment ja es pot utilitzar la generació del *Tessera*, el qual s'adapta a les *tiles* que ja estan col·locades en el mapa. Per cridar a a generació del *Tessera* s'utilitza la funció *Generate.* A la Figura 7.16 es pot veure el resultat d'una generació.

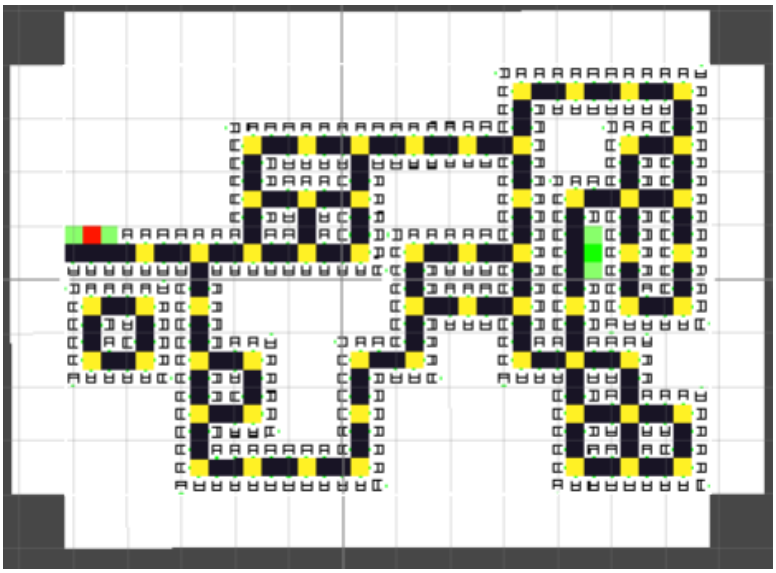

Figura 7.16. Resultat de la generació del Tessera.

7.2.2.6 Comprovar que el nivell és vàlid

En ser un nivell procedimental pot ser que el nivell sigui impossible de completar, que pot passar si no existeix un camí entre la posició d'inici i la final. Per assegurar que això no passi, primer s'ha de comprovar que hi ha com a mínim un camí i per això s'haurà d'aplicar un algorisme de *pathfinding*[15].

Malauradament, no es pot aplicar l'algorisme de *pathfinding* directament perquè no existeix cap estructura que guardi les dades de la ciutat generada, el *Tessera* no permet accedir a la matriu que ha generat aleshores s'ha de buscar una manera d'obtenir aquesta informació.

Aquesta informació es guarda en una classe auxiliar anomenada *TileData,* en la qual podem trobar tota la informació necessària sobre una *tile*. Aquesta informació és la següent.

- ***Type***: guarda quin tipus de *tile* composta és (vegeu Figura 7.5).
- ***Tiles***: guarda quines altres *tiles* té al costat i si són transitables entre elles, és a dir, es pot anar d'una a l'altre.
- ***RotationY***: guarda la rotació en l'eix Y per poder orientar els edificis més endavant.
- ***Position***: guarda en quina posició està del mapa.

[15] Algorisme de *pathfinding*: Algorisme que intenta trobar un camí entre un punt A i un B.

El *ComplexGenerator* té una matriu de *TileData* anomenada *tileMatrix,* la qual omple utilitzant la funció *CompileMatrix*, on es passa per a cada *tile* i s'utilitza el nom per saber de quin tipus és (vegeu Figura 7.17).

```
foreach (Transform child in generator.transform)
{

    index = WorldToMatrix(child.localPosition);
    TileData.tileType type;
    switch (child.name[0])
    {
        case 'i':
            type = TileData.tileType.i;
            break;
        case 'l':
            type = TileData.tileType.l;
            break;
        case 't':
            type = TileData.tileType.t;
            break;
        case 'p':
            type = TileData.tileType.p;
            break;
        default:
            type = TileData.tileType.air;
            break;
    }

    tileMatrix[index.x, index.y] = new TileData(index, type, (int)child.eulerAngles.y);

}
```

Figura 7.17. Omplir matriu.

Després s'itera sobre cada *tile* creada per comprovar si es pot connectar amb les *tiles* veïnes i en crea les connexions (vegeu Figura 7.18).

```
Vector2Int[] explore = { new Vector2Int(0, 1), new Vector2Int(1, 0), new Vector2Int(0, -1), new Vector2Int(-1, 0) };

//Create connections
for (int i = 0; i < tileMatrix.GetLength(0); i++)
{
    for (int j = 0; j < tileMatrix.GetLength(1); j++)
    {
        int ind = 0;
        foreach (Vector2Int dir in explore) //Per a cada direcció
        {
            if (!OutOfBounds(i+dir.x, j+dir.y)) //Si no surt dels límits
            {
                tileMatrix[i, j].SetConnectionIndex(ind, //Crear la connexió
                    ref tileMatrix[i + dir.x,j + dir.y]);
            }

            ind++;
        }
    }
}
```

Figura 7.18. Creació de les connexions.

Per observar s'ha afegit un tros de codi per veure les connexions en l'editor (vegeu Figura 7.19), les línies de color vermell mostren si hi ha una connexió entre dos tiles.

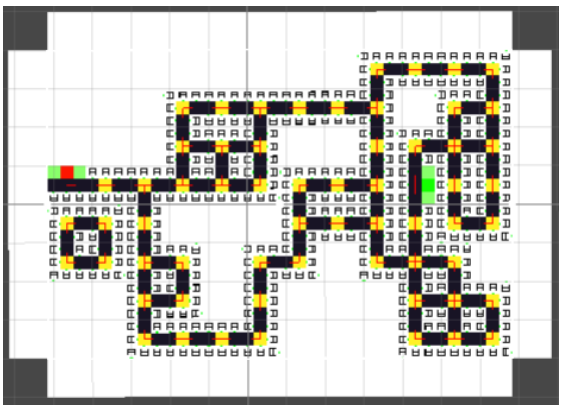

Figura 7.19. Resultat del CompileMatrix.

Finalment, per saber si hi ha connexió entre els dos punts, s'utilitza un algorisme propi, el qual busca el primer camí possible utilitzant la unió entre un algorisme voraç i un *backtracking*. Busca aprofitar la velocitat del voraç i la capacitat de tirar enrere del *backtracking* en el cas de no trobar un camí en el primer intent.

El camí trobat no té per què ser el camí òptim, però només s'està buscant que hi hagi una connexió, per tant és irrellevant si és òptim o no.

L'algorisme creat segueix el següent esquema.

1. Crear una llista ordenada (per la distància de la posició al punt final) de posicions per explorar (LE).
2. Crear una matriu de posicions ja visitades (CE).
3. Crear un booleà que guardi si s'ha trobat camí inicialitzat a fals (T)
4. Afegir posició inicial a LE.
5. Mentre no s'ha trobat camí i LE no buida.
   5.1. Extreure la primera posició de LE (PA).
   5.2. Si PA igual final:
      5.2.1. T = true
   5.3. Si no:
      5.3.1. Marcar PA a CE com ja visitada.
      5.3.2. Guardar a LE tots els veïns de PA no visitats.

Aleshores, utilitzant T es pot saber si s'ha trobat un camí possible. A continuació es pot veure el resultat de l'algorisme (vegeu Figura 7.20).

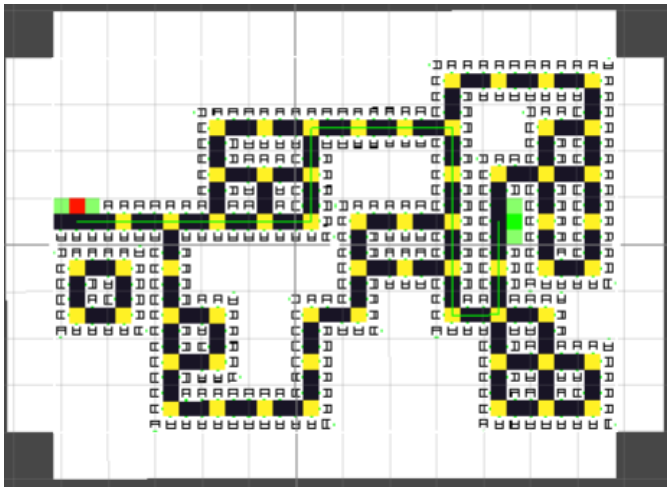

Figura 7.20. Camí trobat per l'algorisme.

Si al fer el càlcul es descobreix que no existeix un camí possible, es neteja la generació i es torna a començar en el següent *frame*. S'ha d'esperar un *frame* perquè el *Tessera* té un *bug*; si intenta fer més d'una generació en el mateix *frame* falla. S'entén que el problema és que no és capaç d'actualitzar les dades respecte a l'última generació i entra en un estat en el qual no pot trobar mai una solució.

Per altra banda, reiniciar tota la generació obre la possibilitat a un bucle infinit de generacions, però és força improbable, ja que la probabilitat de tornar a generar és de 1/15 aproximadament. Per tant la probabilitat de generar dos cops és d'1/225, de tres és d'1/3375, de quatre és d'1/50625, etc.

7.2.2.7 Generar la interconnexió de carrers - Instanciar portals

El següent pas de la generació de la ciutat és crear les interconnexions. Per fer-ho s'ha de popular el mapa amb portals però evitant posar-los en el camí trobat en l'apartat anterior. Això és a causa que podria donar-se el cas que es posés un portal en mig de l'únic camí per arribar a l'objectiu i el jugador no pogués arribar mai al final. Els portals es col·locaran entre dues *tiles* que tinguin connexió entre si. Per escollir les *tiles*, el que es fa primer és aconseguir una *tile* (que no estigui en el camí) i es col·loca un portal entre aquesta i una aleatòria que tingui connectada.

Per escollir la primera *tile* s'ha decidit no utilitzar l'aleatorietat directament. Això és a causa que pot donar-se la possibilitat que tots els portals es col·loquessin per la mateixa zona i deixessin una part de la ciutat buida. Per evitar-ho s'ha decidit utilitzar una unió entre l'algorisme de *Best candidate* i *Poisson Disc Sampling.* Aquests dos serveixen per generar un conjunt de punts aleatoris distribuïts.

- ***Best Candidate:*** l'algorisme genera un conjunt de punts de prova i després mira, per a cada un, quin és el que està més separat dels punts ja generats. El punt més separat de tots és el que retorna.
- ***Poisson Disc Sampling:*** en aquest es defineix una mínima distància r. Aleshores, s'escull un punt generat anteriorment i es genera un nou punt a una distància d'entre r i 2r. Si aquest punt és vàlid, es retorna, si no es torna a generar fins a una quantitat determinada

de vegades. Si un punt arriba a aquesta quantitat, ja no se l'escull per generar punts aleatoris.

S'ha decidit utilitzar un híbrid entre els dos per aconseguir solucionar els problemes que cada un d'ells per separat provocaven en aquest projecte. En primer lloc, en el projecte no s'utilitzen punts aleatoris, sinó un conjunt de *tiles* aleatòries. Per tant, l'algorisme de *Poisson Disc Sampling* no funciona, perquè calcular quines tiles estan a una distància d'entre r i 2r tindria un cost massa elevat. A part, la generació parteix d'un punt i va creixent, per la qual cosa podria donar-se el cas que tots els portals es generessin en la mateixa zona. Finalment, l'algorisme s'atura quan ja no pot escollir més punts per començar a generar, no quan s'ha arribat a un màxim de punts.

En segon lloc, l'algorisme de *Best Candidate* ficaria portals a les cantonades i a les vores del mapa, ja que els primers portals generats buscarien la posició més apartada.

De la unió d'aquests dos apareix l'algorisme d'aquest projecte, el qual se centra a buscar un punt que com a mínim estigui a una distància r dels altres, però si després de diversos intents no en troba cap, redueix aquesta distància per ser més permissiu. D'aquesta manera es té un control sobre el nombre de portals, sobre com queden distribuïts i mai es col·loquen directament en les vores de la ciutat.

Un altre punt a destacar es que quan un portal és genera, deixen de considerar-se vàlides qualsevol *tile* que vagi en línia recta fins al portal. Això és per culpa que, malauradament, no s'ha aconseguit crear recursivitat en els portals. Aleshores, per no veure un portal tot negre en mig del mapa[16] s'ha decidit eliminar la possibilitat que passi.

A la Figura 7.21 es pot observar la implementació de l'algorisme per escollir posicions aleatòries.

[16] Els portals creats per aquest projecte, quan s'aprecien des d'un altre portal (recursivitat), surten pintats de negre.

```
public Vector2Int GenerateSample(ref List<Vector2Int> availablePositions){
    int tries = 0; //Proves per a la generació actual
    bool generated = false; //S'ha generat el punt

    Vector2Int pos = new Vector2Int(); //Posicio a generar

    while(!generated)
    {
        pos = availablePositions[Random.Range(0, availablePositions.Count)]; //Posició aleatòria

        if (PointIsValid(pos)) //Si el punt és vàlid
        {
            positions.Add(pos); //S'afegeix el punt a la llista de punts generats per mantenir un registre
            generated = true;
        }
        else //Si no és vàlid
        {
            tries++; //Augmentar el número de proves
            if (tries > k) //Si el número de proves supera el màxim
            {
                tries = 0; //Reiniciar el número de proves
                radius *= decreaseValue; //Reduir el radi
            }
        }
    }

    return pos;
}
public bool PointIsValid(Vector2Int point)
{
    foreach (Vector2Int p in positions)
    {
        if ((p - point).magnitude < radius)
            return false;
    }

    return true;
}
```

Figura 7.21. Elecció de tiles aleatòries.

Resultat d'instanciar portals:

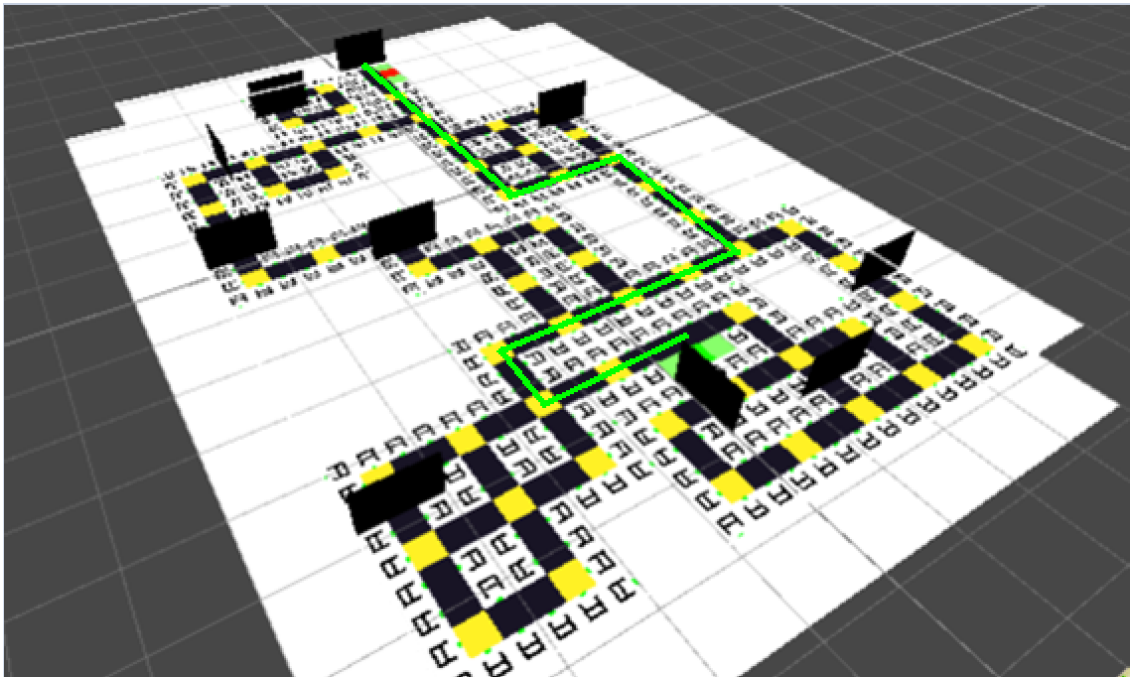

Figura 7.22. Resultat de la generació de portals.

7.2.2.8 Substitució de tiles per models

A continuació es transforma cada *subtile* en el corresponent conjunt de models. Per fer-ho, s'itera sobre cada *subtile* del mapa, es llegeix el nom de la *tile* i s'instancia el conjunt de models necessaris assignats a cada *subtile* (vegeu Figura 7.23).

```
public void BuildCity()
{
    ClearChildren(borderTilesContainer); //Eliminar les vores

    if (startRef != null) //Fer que la sortida sigui processat
        startRef.transform.parent = generator.transform;

    if (finishRef != null) //Fer que el punt final sigui processat
        finishRef.transform.parent = generator.transform;

    foreach (Transform child in generator.transform) //Per a cada tile
    {
        if (child.name == "air") { continue; } //Si és una tile blanca ignorar-la

        //Ignorar en aquest apartat
        chunk

        foreach (Transform subchild in child) //Per a cada subtile
        {
            GameObject spawn = finalTileReferences.GetPrefab(subchild.name); //Aconseguir el conjunt de models assignat
            if (spawn == null) { continue; } // Si el conjunt és null continuar

            GameObject s = Instantiate(spawn, chunk.transform); //Instanciar el conjunt

            //Ignorar en aquest apartat
            //s.GetComponent<ChunkObject>()?.Awake();

            s.transform.position = subchild.position; //Assignar la posició
            s.transform.eulerAngles = new Vector3(0, subchild.eulerAngles.y, 0); //Assignar la rotació
        }
    }

    generator.Clear(); //Netejar el plànol
}
```

Figura 7.23. Funció per construir la ciutat.

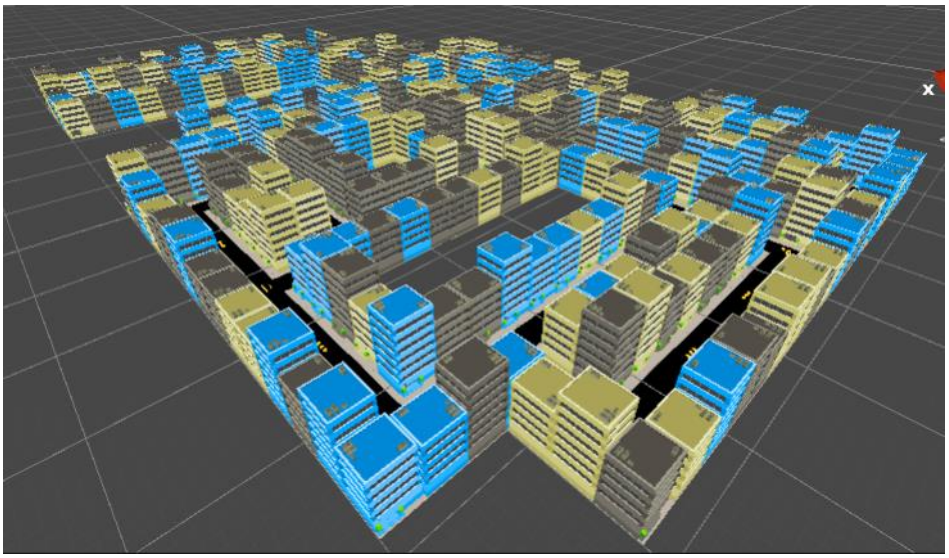

Figura 7.24. Resultat de la substitució de tiles.

D'aquesta manera la ciutat ja estaria generada. El temps per generar el nivell, si es genera correctament a la primera (existeix un camí entre punt A i punt B), és de 250ms de mitjana, i per cada subseqüent generació en cas d'error, se li afegeix 500ms. Per tant, en el cas que el jugador se li generés el mapa 2 cops, aquest tardaria 750ms de mitjana.

7.3 Interconnexió de carrers – portals

Per aconseguir implementar la interconnexió de carrers s'utilitzen dues classes diferents: *PortalTraveller* i *Portal.* La primera és un component que s'afegeix als objectes del joc per permetre que viatgin entre portals, i a la segona trobem tot el codi relacionat amb la visualització de l'altre costat del portal i la translació dels *PortalTraveller*.

7.3.1 *PortalTraveller*

La necessitat d'aquesta classe apareix per dos motius. En primer lloc és l'encarregada de crear un clon gràfic a l'altre costat del portal. En segon lloc permet la notificació a altres entitats quan viatja entre portals.

7.3.1.1 Clon gràfic

El clon gràfic és una còpia de l'apartat gràfic del *PortalTraveller*, el qual es col·loca a l'altre costat del portal mentre aquest l'està travessant, en la posició relativa del jugador per aconseguir crear una transició suau.

Si no existís aquesta còpia, quan una part de l'objecte estigues travessant el portal, es deixaria de veure a l'altre costat i es veuria un tall. En utilitzar el clon es completa l'objecte per l'altre costat (vegeu Figura 7.25).

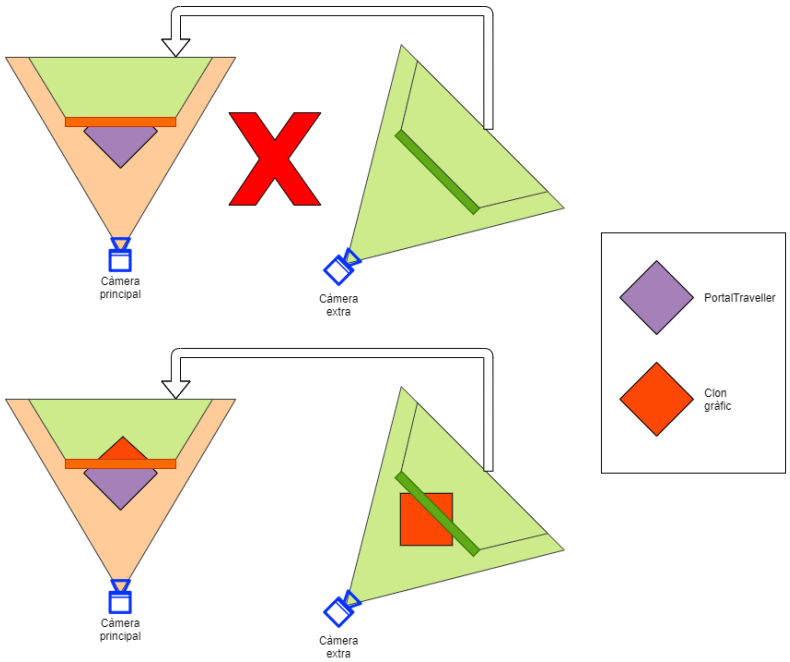

Figura 7.25. Diagrama esquema clon gràfic.

Per utilitzar aquest clon correctament és necessari copiar només la part gràfica del *PortalTraveller,* ja que si es copiés també la part lògica, el clon faria accions dins del joc. Per exemple, es podria moure cap a una direcció errònia perquè llegiria l'input del jugador.

Per aconseguir copiar la part gràfica s'ha de crear l'entitat de tal manera que la seva part gràfica i lògica siguin independents. Aleshores, en el codi s'utilitza una referència a la part gràfica (vegeu Figura 7.26).

```
public class PortalTraveller : MonoBehaviour {

    public GameObject graphicsObject; //Referència a la part gràfica
    public Optional<Transform> graphicsCloneParent; //A quin objecte instanciar la còpia
    12 referencias
    public GameObject graphicsClone { get; set; } //Referència al clon

    //Si està tocant un portal guarda la distància a aquest en l'últim frame
    3 referencias
    public Vector3 previousOffsetFromPortal { get; set; }
```

Figura 7.26. Variables PortalTraveller.

Aquest clon no és necessari fins que l'objecte no a travessa un portal, per tant quan toca un portal, executa el codi de la Figura 7.27:

```
public virtual void EnterPortalThreshold () {
    // Cridat quan es toca un portal

    if (graphicsObject == null) { return; } //Si no existeix part gràfica return

    if (graphicsClone == null) { //Si no tenim cap clon
        graphicsClone = Instantiate (graphicsObject); //Crear clon

        if (graphicsCloneParent.Enabled){ //Si s'utilitza un objecte com a parent
            graphicsClone.transform.parent = graphicsCloneParent.Value; //Posar l'objecte
        }
        else //Si no Copiar el parent de l'objecte original
        {
            graphicsClone.transform.parent = graphicsObject.transform.parent;
        }

        graphicsClone.transform.localScale = graphicsObject.transform.localScale;

    } else { //Si ja existia el clon
        graphicsClone.SetActive (true); //Activar-lo
    }
}
```

Figura 7.27. Funció EnterPortalThreshold del PortalTraveller.

Per altra banda, quan l'objecte es desplaça i deixa de tocar el portal, el clon es desactiva (vegeu Figura 7.28).

```
public virtual void ExitPortalThreshold () {
    //Cridat quan es deixa de tocar un portal, excepte durant la translació
    if (graphicsObject == null) { return; } //Si no existeix clon tornar

    graphicsClone.SetActive (false); //Desactivar clon
}
```

Figura 7.28. Funció ExitPortalThreshold del PortalTraveller.

7.3.1.2 Event de translació

La segona funcionalitat d'aquesta classe és permetre a altres entitats del joc saber quan aquest ha viatjat entre portals. Per fer-ho s'utilitzen els *delegates* d'*Unity*, els quals permeten guardar referències a funcions de la mateixa o altres entitats, per cridar-les quan sigui necessari.

Per utilitzar *delegates* és necessari declarar la signatura per saber quin tipus de funcions es poden subscriure (vegeu Figura 7.29).

```
public delegate void OnPreTeleport(); //Signatura de la funció a cridar
public OnPreTeleport onPreTeleport; //Notificació a on subscriures
public delegate void OnTeleport(Vector3 lastPos, Vector3 newPos, Quaternion lastRot, Quaternion newRot, Portal from, Portal to);
public OnTeleport onTeleport;
```

Figura 7.29. Declaració de delegates.

Finalment quan el *PortalTraveller* viatja entre portals es crida el següent codi (vegeu Figura 7.30), el càlcul de la nova posició i rotació és calculat en el portal que a travessa l'objecte.

```
public virtual void Teleport (Portal fromPortal, Portal toPortal, Vector3 pos, Quaternion rot) {

    onPreTeleport?.Invoke(); //Cridar les funcions guardades en el delegate OnPreTeleport

    Vector3 lastPosition = transform.position; //Guardar la posició anterior
    Quaternion lastRotation = transform.rotation; //Guardar la rotació anterior
    transform.position = pos; //Modificar la posició de l'objecte
    transform.rotation = rot; //Modificar la rotació de l'objecte

    //Cridar les funcions guardades en el delegate OnTeleport
    onTeleport?.Invoke(lastPosition, pos, lastRotation, rot, fromPortal, toPortal);
}
```

Figura 7.30. Funció Teleport del PortalTraveller.

7.3.2 Portal

Com s'ha explicat anteriorment en l'apartat de portals (Apartat 5.3.1), es pot partir la funcionalitat d'aquests en dues propietats, translació

d'objectes i visualització de l'altra banda del portal. A continuació s'explicarà com s'ha aconseguit cada funcionalitat.

7.3.2.1 Translació d'objectes

Per aconseguir que la translació del *PortalTraveller* (viatger d'ara endavant) sigui invisible, s'ha d'utilitzar el clon gràfic (vegeu apartat 7.3.1.1).

Tota possible translació funciona de la següent manera: quan el viatger s'apropa suficientment al portal, el portal activa en el viatger la funció *EnterPortalThreshold* (explicada anteriorment a la Figura 7.27) i el guarda en una llista, amb la qual processa tots els viatgers actius. En aquest moment es crea el clon gràfic si no existia, o s'activa si existia, i el portal el col·loca en la posició correcta respecte a l'altre portal (vegeu Figura 7.31).

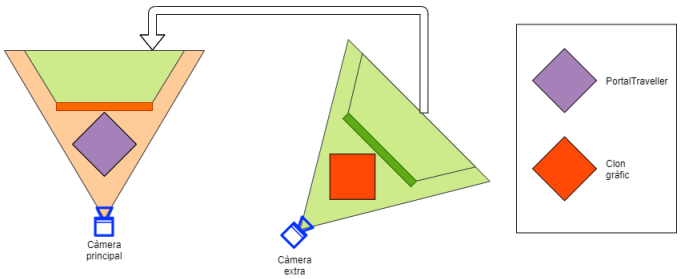

Figura 7.31. Esquema dintre el rang I.

Mentre el viatger no surti del rang i no travessi el portal, és a dir, el centre del viatger no passi d'una cara a l'altra del portal, s'anirà col·locant el clon a la posició relativa (vegeu Figura 7.32). Però si l'objecte no travessa el portal i surt del rang (gira cua per exemple), es treu de la llista de viatgers actius del portal i es crida la funció *ExitPortalThreshold* (explicada anteriorment a la Figura 7.28).

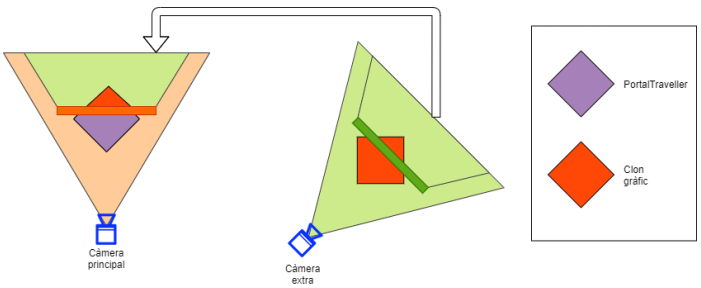

Figura 7.32. Esquema dintre el rang II.

Quan el viatger travessa el portal és quan s'invoca el mètode de *Teleport* (explicada anteriorment a la Figura 7.30). S'elimina el viatger de la llista de viatgers actius i s'introdueix a la llista de l'altre portal. Per tant el viatger serà processat des del portal de sortida.

L'altre portal executarà el mateix codi que el primer, és a dir, mentre el viatger estigui dintre el rang, es mantindrà a la llista i anirà col·locant el clon al portal d'entrada (vegeu Figura 7.33). De la mateixa manera que amb el portal d'entrada, quan el viatger surt del rang del portal de sortida, s'elimina de la llista d'actius i es crida la funció *ExitPortalThreshold* (vegeu Figura 7.34)

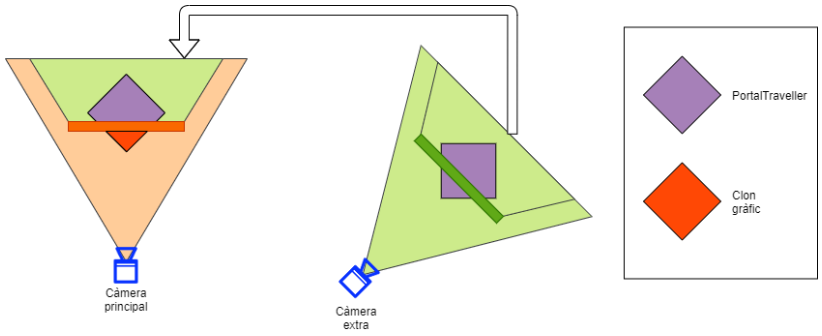

Figura 7.33. Esquema dintre el rang després de la translació.

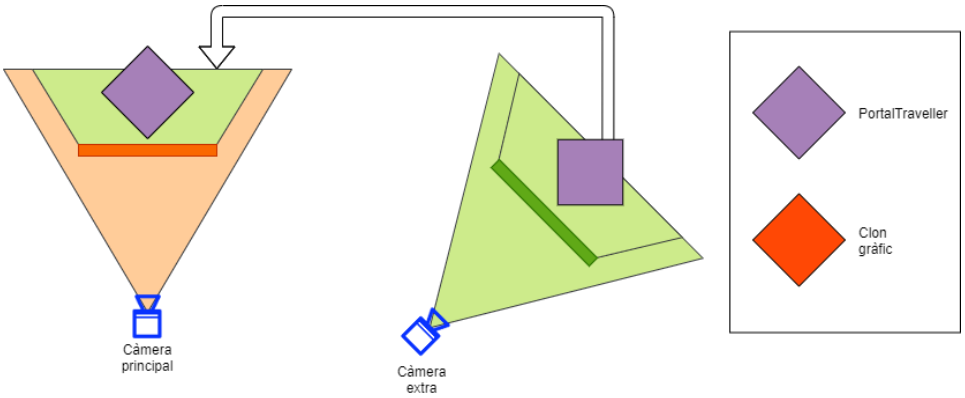

Figura 7.34. Esquema de sortir del rang del portal.

La funció *HandleTravellers,* la qual es mostra a continuació a la Figura 7.35, s'encarrega de gestionar els viatgers amb els portals.

```
void HandleTravellers () {
    if (linkedPortal == this) { return; } //Si el portal esta connectat a ell mateix return

    for (int i = 0; i < trackedTravellers.Count; i++) { //Per a cada traveller
        PortalTraveller traveller = trackedTravellers[i]; //Aconseguir el traveller
        Transform travellerT = traveller.transform; //Guardar el transform

        //Calcular la posició respecte l'altre portal
        var m = linkedPortal.transform.localToWorldMatrix * transform.worldToLocalMatrix * travellerT.localToWorldMatrix;

        Vector3 offsetFromPortal = travellerT.position - transform.position; //Calcular distància al portal
        int portalSide = System.Math.Sign (Vector3.Dot (offsetFromPortal, transform.forward)); //Calcular cara actual
        //Calcular cara del frame anterior
        int portalSideOld = System.Math.Sign (Vector3.Dot (traveller.previousOffsetFromPortal, transform.forward));

        //Si el viatger a canviat de cara (ha travessat el portal)
        if (portalSide != portalSideOld) {
            var positionOld = travellerT.position; //Posició abans de la translació
            var rotOld = travellerT.rotation;  //Rotació abans de la translació

            traveller.Teleport (this, linkedPortal, m.GetColumn (3), m.rotation); //Teletransportar el viatger

            if (traveller.graphicsObject != null) //Si el viatger té clon posar-lo a la posició antiga
                traveller.graphicsClone.transform.SetPositionAndRotation (positionOld, rotOld);

            //Col·locar el viatger a la llista de l'altre portal
            linkedPortal.OnTravellerEnterPortal (traveller);
            trackedTravellers.RemoveAt (i); //Extreure el viatger de la llista
            i--;

        } else { //Si el viatger no ha travessat el portal
            if (traveller.graphicsObject != null) //Si existeix
            {
                //Posicionar el clon
                traveller.graphicsClone.transform.SetPositionAndRotation(m.GetColumn(3), m.rotation);
                CopyLocalTransform(traveller.graphicsObject.transform, traveller.graphicsClone.transform);
            }

            //Guardar posició com l'anterior per calcular el canvi de cara
            traveller.previousOffsetFromPortal = offsetFromPortal;
        }
    }
}
```

Figura 7.35. Funció HandleTravellers de la classe Portal.

7.3.2.2 Visualització de l'altre costat.

Per aconseguir visualitzar l'altre costat del portal, s'han de tenir diferents aspectes en compte.

7.3.2.2.1 Renderitzar en l'ordre correcta

En primer lloc, les càmeres dels portals han de fer render abans que la càmera principal, si no, els portals es veurien sense imatge, ja que el render de la càmera principal és el que surt per pantalla. Per aconseguir-ho totes les càmeres utilitzades pels portals no fan render de manera automàtica, sinó que la càmera principal té una referència a totes les càmeres de portal i crida, a cada una d'elles, la funció de *Render* del portal i un *PostPortalRender* per eliminar possibles talls que puguin aparèixer al fer transició entre portals (vegeu Figura 7.36).

```
void OnPreCull () {
    for (int i = 0; i < portals.Length; i++) { //Per a cada portal
        if (portals[i].enabled) //Si el portal està actiu
        {
            portals[i].Render(); //Fer render del portal
        }
    }

    for (int i = 0; i < portals.Length; i++) { //Per a cada portal
        if (portals[i].enabled) //Si el portal està actiu
        {
            portals[i].PostPortalRender(); //Cridar la funció de post render
        }
    }

}
```

Figura 7.36. Renderitzar en l'ordre correcta.

7.3.2.2.2 Render del portal

Per generar la textura que es posarà en el portal s'utilitza el codi de la Figura 7.37, a més d'un *shader* per col·locar la imatge correctament.

```
public void Render () {

    // Si no existeix camera o el jugador no està prou aprop o el portal no es veu desde la càmera return
    if (playerCam == null || !CloseEnough() || !CameraUtility.VisibleFromCamera (screen, playerCam))
    {
        return;
    }

    //Ignorar en aquest apartat
    if (linkedPortal.oneWay && !locked && SideOfPortal(Camera.main.transform.position) == 1)...

    CreateViewTexture (); //Crear la textura que es col·locarà al portal

    //Calcular posicions de la camera del portal
    var localToWorldMatrix = playerCam.transform.localToWorldMatrix;
    portalCam.projectionMatrix = playerCam.projectionMatrix;
    localToWorldMatrix = linkedPortal.transform.localToWorldMatrix * transform.worldToLocalMatrix * localToWorldMatrix;
    Vector3 renderPosition = localToWorldMatrix.GetColumn (3);
    Quaternion renderRotation = localToWorldMatrix.rotation;

    //Col·locar la càmera a la posició correcta
    portalCam.transform.SetPositionAndRotation (renderPosition, renderRotation);

    // Amagar el portal observat per evitar que sigui renderitzat.
    linkedPortal.screen.shadowCastingMode = UnityEngine.Rendering.ShadowCastingMode.ShadowsOnly;
    screen.material.SetInt ("displayMask", 0);

    //Fer render de la càmera
    SetNearClipPlane (); //Configurar que apareix en la renderització
    portalCam.Render(); //Fer render de la càmera
    screen.material.SetInt ("displayMask", 1);

    // Mostrar el portal observat
    linkedPortal.screen.shadowCastingMode = UnityEngine.Rendering.ShadowCastingMode.Off;
}
```

Figura 7.37. Funció de Render de la classe Portal.

El *shader* transforma les coordenades del portal a coordenades de pantalla per així mostrar de la textura de la càmera només la part que sortiria per pantalla.

A continuació es pot veure una imatge del portal amb el *shader* modificat per mostrar el color de les coordenades (vegeu Figura 7.38). En aquest exemple el blau marí significa cantonada esquerra i el fúcsia significa cantonada dreta de la pantalla.

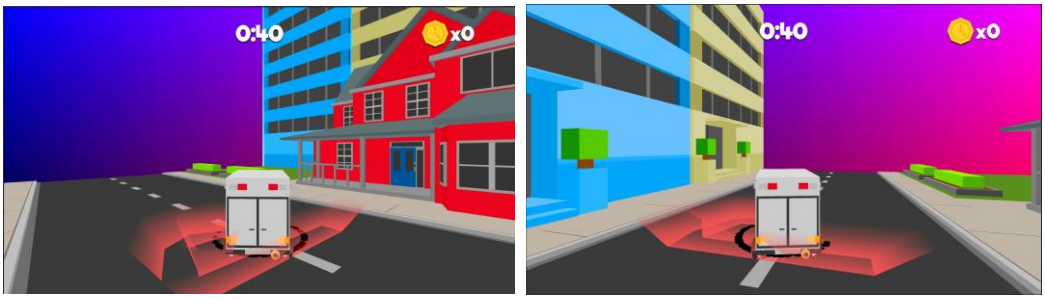

Figura 7.38. Shader del portal modificat.

En el *shader* se li introdueix com a textura principal el que renderitza la càmera del portal (vegeu Figura 7.39).

Figura 7.39. Render d'una càmera de portal.

Aleshores llegint la textura anterior amb les coordenades per pantalla obtenim el següent resultat (vegeu Figura 7.40). Com es pot observar el portal és invisible.

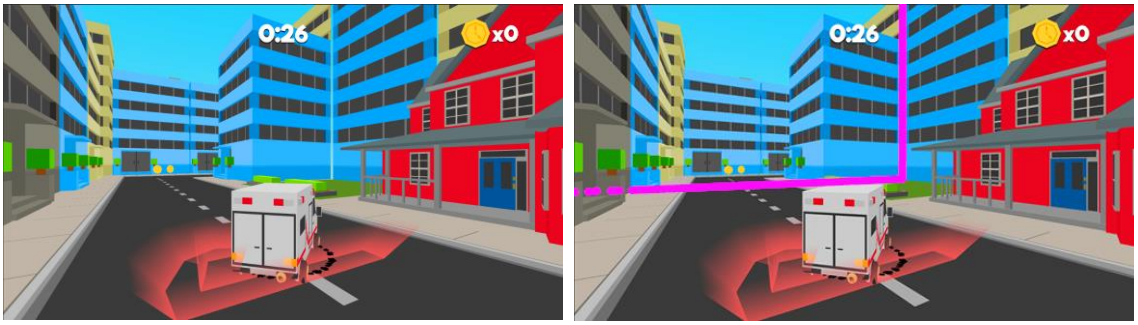

Figura 7.40. Resultat de la visualització sense marcar i marcant el portal.

El codi del *shader* és el següent:

```
fixed4 frag (v2f i) : SV_Target
{
    float2 uv = i.screenPos.xy / i.screenPos.w; //Aconseguir coordenades pantalla

    fixed4 portalCol = tex2D(_MainTex, uv); //Llegir textura càmera portal
    return portalCol * displayMask + _InactiveColour * (1-displayMask); //Desactivar/Activar el portal
}
```

Figura 7.41. Codi del shader de portals.

7.3.2.3 Portal principal

Amb el codi de translació que s'ha ensenyat (vegeu Apartat 7.3.2.1) si s'entra per un portal frontalment, es viatja fins a l'altre portal i se surt pel darrere. Això produeix un problema, ja que un dels punts de disseny del videojoc és que el jugador sempre torni a la sortida, però sempre per la mateixa cara del portal. Si el jugador surt per la cara equivocada, no veurà que ha passat per una interconnexió.

Per aconseguir que sempre surti per la mateixa cara del portal, la solució ha estat girar els portals quan aquests estan orientats de manera incorrecta. Per tant, si el jugador va cap a un portal i aquest el portarà a la tile del darrere del portal de sortida, el portal farà una rotació de 180 graus per orientar-se bé. A continuació, a la Figura 7.42, hi ha el codi per rotar el portal:

```
//Si el portal només permet una direcció, no s'està travessant i el jugador esta en la cara equivocada
if (linkedPortal.oneWay && !locked && SideOfPortal(Camera.main.transform.position) == 1)
{
    InvokeOnPortalRotated(); //Girar el portal
}
```

Figura 7.42. Orientar el portal correctament.

7.4 Personatge principal

Podem dividir la discussió sobre el personatge principal en tres parts: el controlador del vehicle, el qual gestiona el moviment; l'*AmbulanceTraveller*, una classe que hereta de *PortalTraveller* (vegeu Apartat 7.3.1) per aconseguir que viatgi entre portals; i una càmera en tercera persona modificada perquè funcioni amb els portals.

7.4.1 Moviment del personatge principal

Per aconseguir el moviment de l'ambulància s'ha decidit utilitzar un sistema simple, és a dir, sense utilitzar suspensions ni marxes. En primer lloc les suspensions no són necessàries, ja que la ciutat és plana; i en segon, les marxes només serien una capa afegida de decoració.

Aleshores, per implementar el moviment s'ha decidit utilitzar una solució simple però molt eficaç: Es divideix el vehicle en dues parts (vegeu Figura 7.43): una esfera i l'ambulància.

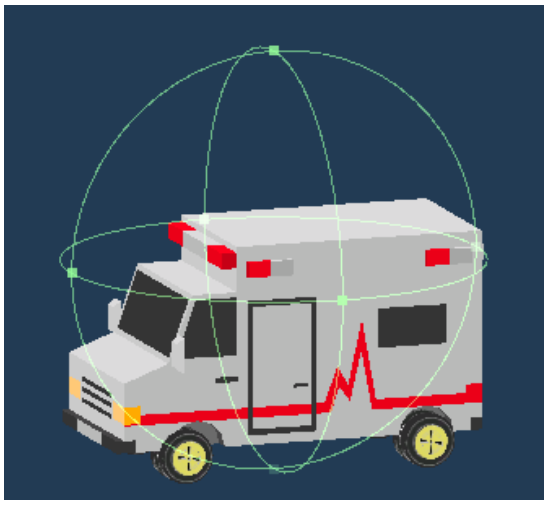

Figura 7.43. Esfera i l'ambulància del vehicle.

A l'esfera se li apliquen forces a cada *FixedFrame*[17] en la direcció de l'ambulància per aconseguir el moviment del vehicle (vegeu Figura 7.44). D'aquesta manera si es vol fer girar el vehicle, es fa rotar l'ambulància cap a la nova direcció i les forces aplicades a l'esfera faran que aquesta es redirigeixi.

```
private void FixedUpdate()
{
    if (!controllable) { return; } //Si l'ambulancia no es pot controlar (joc pausat) return

    if(!drifting) //Si el vehicle no està derrapant, s'aplica la força en la direcció right * -1 del transform (endavant en el model)
        sphereRigidbody.AddForce(-kartModel.transform.right * currentSpeed, ForceMode.Acceleration); //Ambulancia rotada 90 graus
    else //Si el vehicle està derrapant, s'aplica la força en la direcció forward del transform (lateral en el model)
        sphereRigidbody.AddForce(transform.forward * currentSpeed, ForceMode.Acceleration);

    //Aplicar gravetat
    sphereRigidbody.AddForce(Vector3.down * gravity, ForceMode.Acceleration);

    //Rotar l'objecte
    transform.eulerAngles = Vector3.Lerp(transform.eulerAngles, new Vector3(0, transform.eulerAngles.y + currentRotate, 0), Time.deltaTime * 5f);

    //Col·locar en posició l'ambulància
    RaycastHit hitNear;
    Physics.Raycast(transform.position + (transform.up * .1f)   , Vector3.down, out hitNear, 2.0f, layerMask);
    kartNormal.up = Vector3.Lerp(kartNormal.up, hitNear.normal, Time.deltaTime * 8.0f);

    //Girar l'ambulancia
    kartNormal.Rotate(0, transform.eulerAngles.y, 0);
}
```

Figura 7.44. FixedUpdate de l'ambulància.

Les forces a aplicar i la rotació són calculades entre els *frames* dels joc. Per calcular la velocitat es mira si el jugador està accelerant i per la rotació, si està desplaçant el *joystick* o prement les tecles adequades.

[17] FixedFrame: Frame utilitzat per la física, que es manté d'intervals constants durant tot el videojoc.

7.4.1.1 Derrapar

Per aconseguir que el personatge principal pugui derrapar es parteix l'acte en diferents parts:

- **Saltar**: per començar a derrapar primer el vehicle fa una petita elevació vertical en la qual pot inclinar-se més cap a una direcció que en el cas de no saltar.
- **Mantenir**: s'activa un cop ha fet el salt, i si el jugador manté premut el botó de derrapar, una variable interna anomenada *driftPower*, la qual augmenta segons el temps que estigui derrapant sense xocar contra cap paret.
- **Deixar anar**: quan el jugador deixa anar el botó de derrapar és transforma el valor del *driftPower* en un valor de l'1 al 3 el qual s'utilitza per calcular la duració de la millora de velocitat a l'acabar el derrapar.

7.4.2 Travessar portals - *AmbulanceTraveller*

Per viatjar entre portals s'utilitza la classe *AmbulanceTraveller*, una classe que modifica les funcions del *PortalTraveller* (a través d'herència) per aconseguir adaptar-se a l'estructura explicada a l'apartat 7.4.1. Per tant a l'hora d'aplicar la translació del vehicle, ha de fer viatjar l'esfera i l'ambulància (vegeu Figura 7.45).

```
private void OnTeleportEvent(Vector3 lastPos, Vector3 newPos, Quaternion lastRot, Quaternion rot, Portal fromPortal, Portal toPortal)
{
    //Ignorar en aquest apartat
    lastPortal = fromPortal;
    betweenPortals = true;

    //Moure les entitats: ambulància i esfera
    for (int i = 0; i < moveObjects.Length; i++) //Per cada entitat
    {
        Rigidbody rigidbody = moveObjects[i].GetComponent<Rigidbody>(); //Aconseguir rigid body
        if (rigidbody != null) //Si té rigid body
        {
            //Transformar les velocitats dels objectes per fer-les relatives al portal
            rigidbody.velocity = toPortal.transform.TransformVector(fromPortal.transform.InverseTransformVector(rigidbody.velocity));
            rigidbody.angularVelocity = toPortal.transform.TransformVector(fromPortal.transform.InverseTransformVector(rigidbody.angularVelocity));
        }

        moveObjects[i].position = transform.position + moveObjectOffset[i]; //Moure l'objecte
        moveObjects[i].rotation = rot; //Rotar l'objecte
    }
}
```

Figura 7.45. Modificació del AmbulanceTraveller.

S'aplica l'*AmbulanceTraveller* a un cub invisible, el qual té la funció de col·lidir només amb portals (vegeu Figura 7.46).

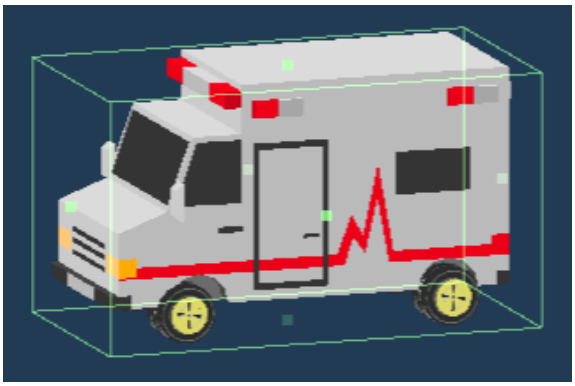

Figura 7.46. Cub per detectar portals.

7.4.3 Modificació de la càmera

Les càmeres de tercera persona s'acostumen a estructurar a dintre de l'entitat que segueixen, per tant, quan l'entitat que segueixen es mou, la càmera es mou amb ella (vegeu Figura 7.47).

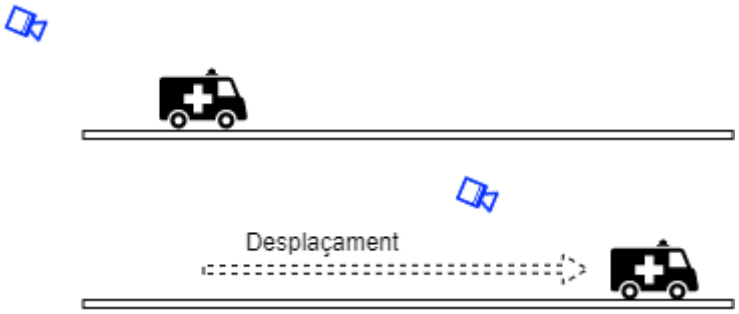

Figura 7.47. Estructura d'una càmera en tercera persona.

Malauradament, aquesta càmera no funciona amb els portals, ja que, si durant el desplaçament es viatja a través de dos portals, la càmera també es mouria amb l'entitat principal i la transició tindrà un tall (vegeu Figures 7.48 i 7.49).

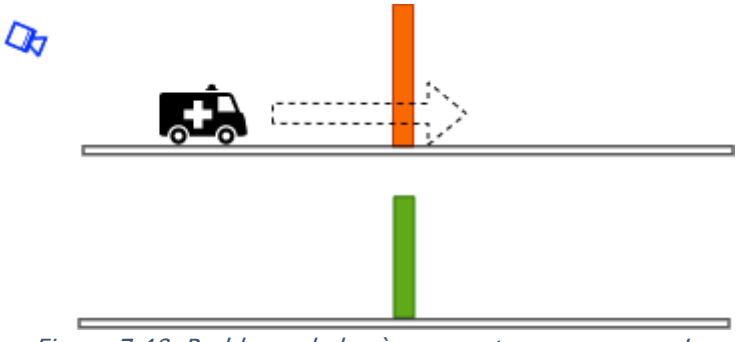

Figura 7.48. Problema de la càmera en tercera persona I.

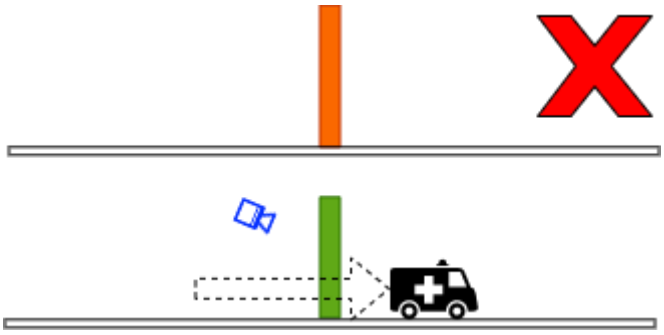

Figura 7.49. Problema de la càmera en tercera persona II.

Per solucionar-ho s'utilitza un objecte extra, *AmbulanceFollower*. Aquest objecte seguirà l'ambulància, però no viatjarà entre portals. La càmera estarà enfocant a aquest objecte i per tant, l'ambulància viatjarà entre portals sense moure la càmera. Quan la càmera passa per un portal es fa viatjar la càmera i es col·loca l'*AmbulanceFollower* a sobre l'ambulància. Per tant, la transició entre portals en realitat seguirà el següent esquema:

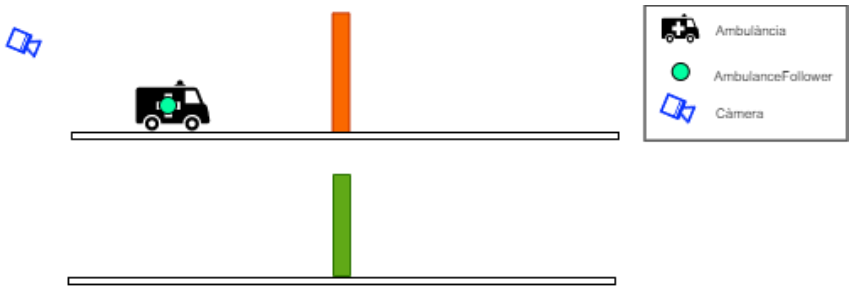

Figura 7.50. Transició entre portals. Abans de passar pel portal.

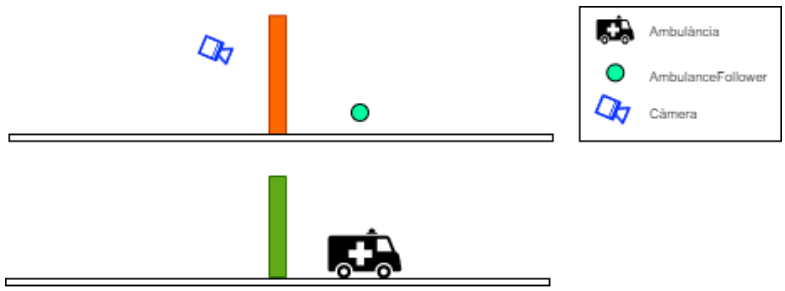

Figura 7.51. Transició entre portals. Càmera sense travessar el portal.

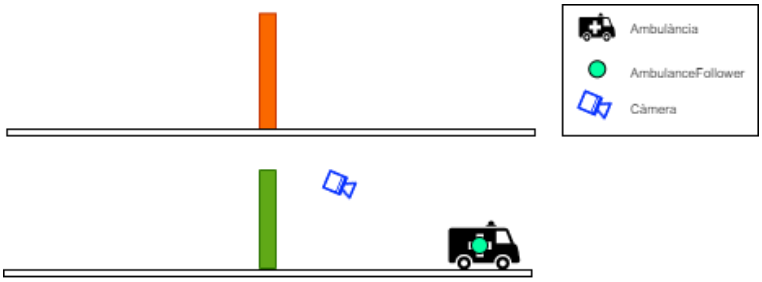

Figura 7.52. Transició entre portals. Després e passar pel portal

Per decidir la posició de l'*AmbulanceFollower,* s'utilitza la posició de l'ambulància depenent de si hi ha un portal entre la càmera i l'ambulància. Si no hi ha un portal, es copia la posició de l'ambulància directament (vegeu Figura 7.50). En cas que hi hagi un portal (vegeu Figura 7.51), es calcula la posició relativa de l'ambulància i es posa l'*AmbulanceFollower* en la posició correcta (vegeu Figura 7.53).

```
public void SetFollowerPosition()
{
    Vector3 pos; //Posicionar a col·locar
    Quaternion rot; //Rotació a col·locar
    if (betweenPortals) //Si existeix un portal entre la càmera i l'ambulància
    {
        //Transformació inversa - Calcular posició relativa
        Transform f = lastPortal.linkedPortal.transform;
        Transform t = lastPortal.transform;
        var m = t.localToWorldMatrix * f.worldToLocalMatrix * transform.localToWorldMatrix;
        pos =  m.GetColumn(3);
        rot =  m.rotation;
    }
    else //Si no - copiar posició i rotació ambulància
    {
        pos = transform.position;
        rot = transform.rotation;
    }

    //Posicionar el AmbulanceFollower
    follower.SetPosition(pos, rot);
}
```

Figura 7.53. Funció SetFollowerPosition de l'AmbulanceTraveller.

Per saber si hi ha un portal entre la càmera i l'ambulància, s'utilitza un booleà, *betweenPortals.* Aquest es posa en *true* quan l'ambulància passa per un portal i es posa en *false* quan la càmera passa pel portal (vegeu Figura 7.54). Aquest sistema falla en el cas que hi hagi més d'un portal entre la càmera i l'ambulància, però aquesta situació no es dóna en el projecte gràcies a l'ús de la distància mínima entre portals. Per tant, aquest error, es pot ignora.

```
private void OnTeleportEvent(Vector3 lastPos, Vector3 newPos, Quaternion lastRot, Quaternion rot, Portal fromPortal, Portal toPortal)
{
    //Set el betweenPortals
    lastPortal = fromPortal;
    betweenPortals = true;

    ...
}

1 referencia
public void ResetBetweenPortals()
{
    betweenPortals = false;
}
```

Figura 7.54. Codi per activar i desactivar el betweenPortals.

7.5 Lògica del videojoc - *GameManager*

La lògica del videojoc és l'encarregada de gestionar els estats de tots els components del joc, avaluar la informació dels components i la pròpia, i modificar l'estat del joc per aconseguir que funcioni. En aquest projecte, la classe encarregada de gestionar el joc és la *GameManager*. Aquesta segueix el patró *Singleton*[18], ja que es necessita una sola instància.

Tots els components del joc que necessiten informació sobre l'estat d'aquest criden al *GameManager*, per exemple quan la ciutat s'acaba de generar, *GameManager* envia a l'ambulància que pot llegir input del jugador per a ser controlada. El joc pot estar en els següents estats:

- ***LoadingCity:*** la ciutat encara s'està generant.
- ***Loaded:*** la ciutat ja s'ha generat, però el jugador encara no ha començat la partida.
- ***Playing:*** el jugador està en mig de la partida.
- ***Paused:*** el joc està pausat.
- ***EndScreen:*** la partida ha acabat i s'està mostrant el menú final.

El *GameManager* guarda en una variable en quin estat es troba el joc, i les altres classes que necessitin aquesta informació criden funcions auxiliars per saber l'estat del joc (vegeu Figura 7.55).

```
public class GameManager : MonoBehaviour
{
    private static GameManager Instance; //Patrò singleton

    14 referencias
    private enum GameState { //Possibles estats del joc
        loadingCity,
        loaded,
        playing,
        paused,
        endScreen
    }

    private GameState gameState; //Estat actual
    private GameState lastGameState; //Estat anterior, serveix per saber a quin
    //estat tornar en sortir del menú de pausa

    //Funcions auxiliars
    5 referencias
    public static bool playing { get { return Instance.gameState == GameState.playing; } }
    0 referencias
    public static bool loading { get { return Instance.gameState == GameState.loadingCity; } }
    2 referencias
    public static bool paused { get { return Instance.gameState == GameState.paused; } }
    0 referencias
    public static bool endScreen { get { return Instance.gameState == GameState.endScreen; } }
    3 referencias
    public static bool loaded { get { return Instance.gameState == GameState.loaded; } }
    1 referencia
    public static bool loadChunk { get { return playing || loaded; } }
    2 referencias
    public static bool canPause { get { return playing || loaded; } }
```

Figura 7.55. Sistema d'estats del GameManager.

[18] Patró Singleton: patró basat a assegurar que en cada moment només existeix una instància d'una classe.

7.5.1 Calcular resultat de partida victòria o derrota

Per a saber si el jugador ha superat l'objectiu del joc s'utilitza un rellotge intern, que es va reduint amb el temps i quan arriba a 0 el jugador perd. Ara bé, si el jugador envia la notificació que ha arribat al final abans que el temporitzador arribi a 0, guanya la partida (vegeu Figura 7.56).

```
void Update()
{
    //Començar el videojoc
    if (loaded && Control.AccelerateDown()) { gameState = GameState.playing; }

    //Si no s'està jugant return
    if (!playing) { return; }

    currentTime -= Time.deltaTime; //Reduir el temps del joc

    canvas?.SetTimer(currentTime); //Actualitzar la pantalla

    if (currentTime <= 0.0) //Si no queda temps
    {
        Lose(); //El jugador ha perdut
    }
}

1 referencia
public static void AmbulanceArrivedHospital()
{
    if (Instance.gameState == GameState.playing) //Si la partida encara segueix en curs
    {
        Instance.Win(); //El jugador ha guanyat
    }
}
```

Figura 7.56. Lògica de victòria i derrota.

7.5.2 Puntuació del jugador

Per gestionar la puntuació del jugador la qual es mostra al final de la partida s'utilitza una classe auxiliar: *ScoreSystem.* Aquesta classe serveix per extreure totes les funcions i dades del *GameManager* perquè sigui més entenedor.

El *ScoreSystem* guarda tota la informació que és necessària per a l'avaluació del jugador i amb la funció *CalculateResult* retorna quina és la qualificació del jugador (vegeu Figura 7.57).

```
public char CalculateResult()
{
    if (!victory) { return 'F'; } //Si no s'ha guanyat retorna F

    float timeValue = time / maxTime; //Calcular proporció del temps

    //Si no s'ha xocat contra la paret i ha arribat amb més temps que l'inicial retorna S.
    if (comboBreaks == 0 && timeValue > 1f) { return 'S'; }

    //Si ha xocat contra la paret un màxim de dos cops i encara li queda més del 75% del temps inicial retorna A.
    if (comboBreaks < 3 && timeValue > 0.75f) { return 'A'; }

    //Si ha xocat contra la paret un màxim de 4 cops i encara li queda més del 40% del temps inicial retorna A.
    if (comboBreaks < 5 && timeValue > 0.40f) { return 'B'; }

    //Si no s'ha complert res, el resultat és mitjà
    return 'C';
}
```

Figura 7.57. Càlcul de la puntuació del jugador.

7.5.3 Gestionar el resultat de partida

Un cop s'acaba la partida, el jugador pot haver guanyat o perdut. Per gestionar-ho internament s'utilitza el codi de la Figura 7.58.

```
void Win()
{
    EndGame(true); //Jugador guanya
}

1 referencia
void Lose()
{
    EndGame(false); //Jugador perd
}

2 referencias
void EndGame(bool victory)
{
    gameState = GameState.endScreen; //Entrar en l'estat d'endgame
    score.victory = victory; //Dir al sistema de puntuació si el jugador ha guanyat

    if (victory) //Si ha guanyat
    {
        AudioManager.PlayVictory(); //Fer sonar el so de victòria
        score.SetTimes(currentTime, maxTime); //Dir al sistema de puntuació el temps del jugador
    }
    else //Si no
    {
        AudioManager.PlayDefeat(); //Fer sonar el so de derrota
        score.SetTimes(0, maxTime); //Dir al sistema de puntuació el temps del jugador
    }

    canvas?.OpenEndScene(score); //Mostrar pantalla final

    onEndEvent?.Invoke(); //Enviar notificació de final de partida
}
```

Figura 7.58. Gestió del final de partida.

7.6 Sistema de combo – ComboPanel

Per implementar el sistema de combo s'ha creat la classe *ComboPanel*, la qual es pot partir en dues parts, la part lògica i la part gràfica. La part lògica s'encarrega de gestionar la bonificació i el control del combo actual del jugador; per altra banda, la part gràfica s'encarrega de mostrar la informació del combo per pantalla.

7.6.1 Part lògica

Per gestionar el combo s'utilitzen les variables mostrades a la Figura 7.59.

```
//Part lògica
public float comboTime = 1f; //Temps restant màxim per seguir el combo
private float currentComboTime; //Temps restant actual

public int maxLvlCombo = 3; //Nivell màxim de combo
private int currentLvlCombo; //Nivell actual de combo

public int actionsForLevelingUpCombo = 10; //Accions per pujar de nivell de combo
private int currentActions; //Accions actuals per pujar de nivell.
```

Figura 7.59. Variables de la part lògica del ComboPanel.

La variable de *currentCombotTime* s'utilitza com a rellotge intern. Quan aquest arriba a 0, el combo actual que tingues el jugador s'elimina (vegeu Figura 7.60).

```
private void Update()
{
    //Si no hi ha cap combo actiu o no estem en partida return
    if (currentLvlCombo == 0 || !GameManager.playing) { return; }

    //Part gràfica
    timerBar.localScale = new Vector3(currentComboTime / comboTime, 1, 1);

    //Reduïr el temps del rellotge
    currentComboTime -= Time.deltaTime;

    //Si el jugador està en mig d'un combo i es queda sense temps.
    if (currentLvlCombo != 0 && currentComboTime < 0)
    {
        BreakCombo(); //Trencar el combo
    }
}
```

Figura 7.60. Funcionament del rellotge intern del ComboPanel.

Quan el jugador fa una acció de combo, s'activa la funció *AddComboAction* i aleshores es reinicia el rellotge. Si no té cap combo actiu en comença. Si ja en tenia un, augmenta el número d'accions de combo i si aquestes superen el límit establert, s'augmenta el nivell de combo (vegeu Figura 7.61).

```
private void AddComboAction()
{
    currentActions++; //Augmentar accions del combo
    currentComboTime = comboTime; //Reiniciar el rellotge

    if (currentLvlCombo == 0) //Si el jugador no està en combo
    {
        LevelUpCombo(); //Augmentar el nivell del combo
    }
    //Si no i s'ha superat el límit
    else if (currentActions >= actionsForLevelingUpCombo) //
    {
        LevelUpCombo(); //Augmentar el nivell del combo
    }
}
```

Figura 7.61. Funció AddComboAction del ComboPanel.

7.6.2 Part gràfica

La part gràfica del sistema de combo la podem partir en dues parts: la part estàtica i la part dinàmica. A la part estàtica s'inclouen tots els elements que es mostren quan el jugador està executant un combo. Per altra banda, la part dinàmica inclou les notificacions que apareixen per pantalla.

7.6.2.1 Part estàtica

Aquest apartat inclou dos gràfics, una barra que s'utilitza com a rellotge i un text per mostrar el multiplicador actual del combo. La barra es va fent petita a mesura que la variable *currentCombotTime* (vegeu Apartat 7.6.1) s'acosta a 0. El multiplicador actual (vegeu el "x3" de la Figura 7.62) agafa el valor de la variable *currentComboLevel* (vegeu Apartat 7.6.1).

Figura 7.62. Part estàtica de la part gràfica,

7.6.2.2 Part dinàmica

Per donar *feedback* al jugador s'utilitzen unes notificacions les quals descriuen l'acció acabada de fer pel jugador. Les notificacions es pinten d'un color diferent per cada acció per fer que siguin més fàcils de llegir. Les notificacions que s'utilitzen en el projecte són les següents:

- **Moneda:** apareix quan el jugador agafa una moneda en el joc. La X es modifica pel valor de temps entregat.

Figura 7.63. Notificació de moneda.

- **Derrapatge:** apareix quan el jugador acaba de derrapar, depenent del nivell d'aquest (vegeu Figura 7.4.1.1), s'utilitza una notificació diferent. La X es modifica pel valor de temps entregat .

Figura 7.64. Notificació de derrapatge nivell 1.

Figura 7.65. Notificació de derrapatge nivell 2.

Figura 7.66. Notificació de derrapatge nivell 3.

- **Trencament del combo**: Apareix quan el jugador col·lideix amb una paret i finalitza el combo.

Figura 7.67. Notificació de trencament de combo.

En comptes de generar una notificació cada vegada que es necessita una, s'ha decidit utilitzar el sistema de *Pooling*, el sistema té una llista d'entitats disponibles i si no n'hi ha cap en crea una. D'aquesta manera s'aconsegueix més eficiència, ja que el joc no ha d'anar creant i destruint entitats (vegeu Figura 7.68).

```
private ComboNotification GetNotification() //Sistema de pooling
{
    //Buscar una notificació inactiva
    for (int i = 0; i < notificationPool.Count; i++)
    {
        if (!notificationPool[i].playing) //Si no s'està utilitzant
        {
            return notificationPool[i]; //Retornar-la
        }
    }

    //Si no n'hi ha cap disponible generar una de nova
    ComboNotification newNotification = Instantiate(comboNotificationPrefab, transform);
    return newNotification;
}
```

Figura 7.68. Sistema de pooling del ComboPanel.

Per fer aparèixer una notificació, es demana al sistema de *pooling* una notificació, que es configura i després es mostra per pantalla en una petita animació feta per codi, la qual la fa aparèixer amb un *fade in* i *fade out* mentre la desplaça cap amunt.

```
private void ShowNotification(string text, Color color)
{
    ComboNotification notif = GetNotification(); //Aconseguir notificació

    notif.SetText(text, color); //Configurar notificació
    notif.Play(); //Començar l'animació de la notificació
}
```

Figura 7.69. Codi per mostrar una notificació.

A continuació es pot veure en acció:

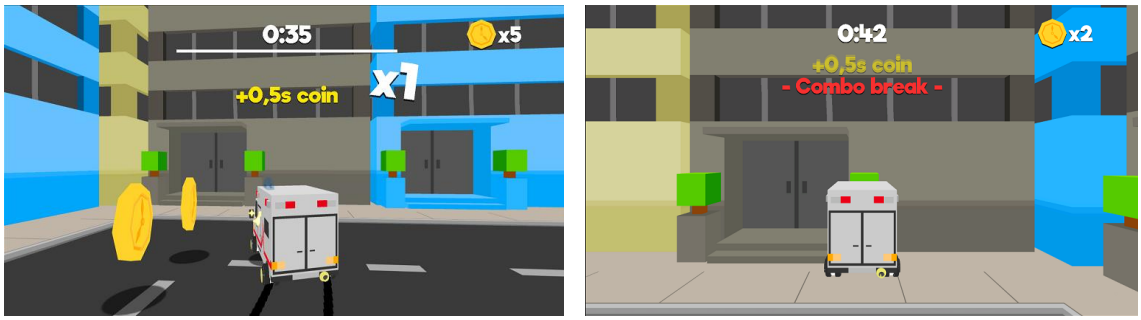

Figura 7.70. Sistema de combo en acció.

7.7 Pantalles del videojoc

7.7.1 Menús

Tots els menús del videojoc utilitzen la mateixa estructura: un conjunt de botons amb funcionalitats diferents i una classe que permet el desplaçament entre ells. Les funcionalitats dels botons poden ser molt variades. Per exemple en el menú principal (vegeu apartat 6.7.3) es troba el botó de *play*, el qual permet anar a la pantalla del selector de nivells; el botó d'opcions, el qual permet anat a la pantalla d'opcions; i el botó de sortir, el qual permet sortir del joc.

7.7.1.1 Base Botó – SelectableObject

Per aconseguir crear un sistema que permeti crear una funcionalitat diferent per a cada botó s'ha utilitzat herència. Es parteix d'una classe base anomenada *SelectableObject*, la qual té implementades totes les funcionalitats que comparteixen entre botons. Les funcionalitats que comparteixen són:

- **Seleccionar**: permet deixar marcat el botó com a pròxim a ser utilitzat. En el projecte es mostra que el botó està marcat utilitzant els símbols ">" i "<" al costat del text del botó (vegeu Figura 7.71).

```
public void Select(bool playSound = true)
{
    if (playSound) { //Si es vol que soni el botó
        AudioManager.PlayButtonSelect(); //Fer sonar el so
    }

    selected = true; //Guardar que està seleccionat

    textMesh.text = "> " + initialText + " <"; //Marcar
}
```

Figura 7.71. Funcionalitat base de seleccionar.

- **Utilitzar**: permet activar la funcionalitat del botó. Aquesta funcionalitat serà sobreescrita amb herència per crear la funcionalitat específica de cada botó.

```
public virtual void Use() { //Herència
    AudioManager.PlayButtonUse();  //Fer sonar el so d'utilitzar
}
```

Figura 7.72. Funcionalitat base d'utilitzar.

- **Deseleccionar:** permet desmarcar el botó com a pròxim a ser utilitzat.

```
public void Deselect()
{
    textMesh.text = initialText; //Treure > i <
    selected = false; //Deseleccionar
}
```

Figura 7.73. Funcionalitat pròpia de deseleccionar.

Figura 7.74. Mostra d'un botó deseleccionat i seleccionat.

7.7.1.2 Funcionalitats específiques

Les funcionalitats específiques creades pel projecte són:

- ***Event caller:*** crida una funció d'un *script*, la qual es pot escollir des de l'editor d'*Unity*.
- ***Exit Application:*** surt del joc.
- ***Exit pause menu:*** surt del menú de pausa, continuant la partida.
- ***Change scene:*** canvia l'escena[19].
- ***Next level:*** canvia d'escena al següent nivell. Els nivells són: nivell de tutorial 1, nivell de tutorial 2, ciutat generada mida petita, generada mida mitjana i generada mida llarga.
- ***Reload scene:*** recarrega l'escena, internament és el mateix que canviar d'escena a l'escena actual.
- ***Audio controller:*** canvia el volum dels efectes sonors del joc.
- ***Music controller:*** canvia el volum de la música del joc.
- ***Show FPS:*** canvia la configuració per mostrar o no mostrar els *FPS* del jugador.
- ***Show time in miliseconds:*** canvia la configuració per mostrar el temps restant de partida amb mil·lisegons o sense.

[19] Escena: contenidors dels objectes del joc. Per exemple, en el projecte existeix una escena de menú principal, els nivells de tutorial i el nivell de generació procedimental.

7.7.1.3 Utilització dels botons – PanelController

Per poder utilitzar els botons s'utilitza la classe *PanelController. Serveix* per gestionar el conjunt de botons segons l'input del jugador. El *PanelController* té una referència a tots els botons que ha de gestionar i pot tenir un extra, el qual s'executa quan el jugador utilitza l'acció de tirar enrere. Per exemple, si està en la pantalla de Selector de nivells i prement la tecla Esc, es torna al menú principal.

```
public class PanelController : MonoBehaviour
{
    public Optional<SelectableObject> onBack; //Acció per tirar enrere
    public SelectableObject[] selectableObjects; //Conjunt de botons
```

Figura 7.75. Variables del PanelController.

Per gestionar l'*input* s'utilitza el codi de la Figura 7.76.

```
public void Update()
{
    if (Control.DownPressed())
    {
        MoveForward(); //Seleccionar el següent botó
    }
    else if (Control.UpPressed())
    {
        MoveBackward(); //Seleccionar el botó previ
    }
    else if (Control.ActivateButtonPressed())
    {
        Use(); //Utilitzar el botó
    }
    else if (Control.BackPressed())
    {
        Back(); //Cridar la funcionalitat del botó back.
    }
}
```

Figura 7.76. Gestió de l'input del jugador del PanelController.

7.7.1.4 Resultats finals

A continuació es poden observar les pantalles finals amb el seu disseny inicial (vegeu Figures 7.77 a 7.83).

7.7.1.4.1 Pantalla de recomanació de comandament

Figura 7.77. Disseny i resultat final de la pantalla de recomanació de comandament.

7.7.1.4.2 Pantalla de títol

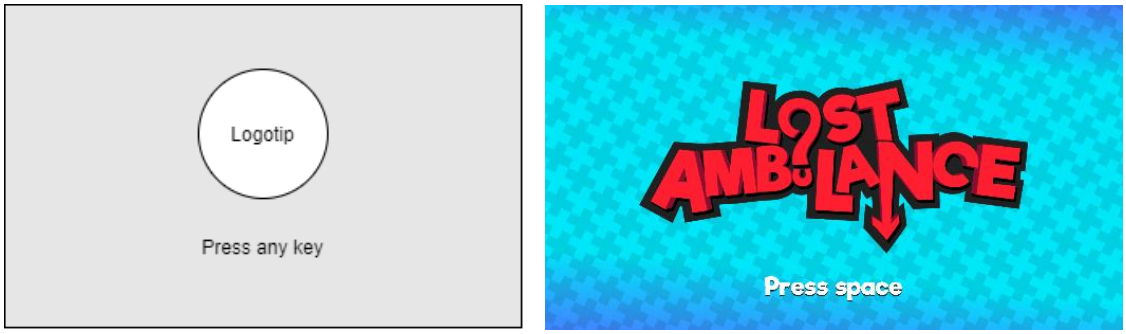

Figura 7.78. Disseny i resultat final de la pantalla de títol.

7.7.1.4.3 Menú principal

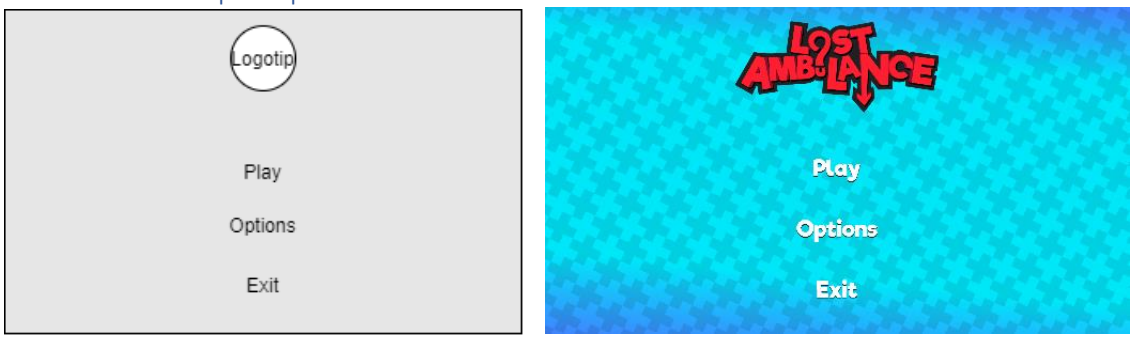

Figura 7.79. Disseny i resultat final del menú principal.

7.7.1.4.4 Selector de nivells

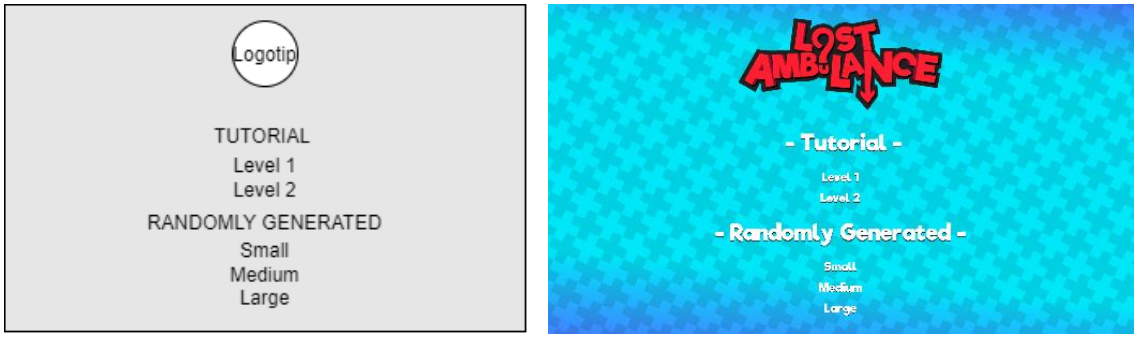

Figura 7.80. Disseny i resultat final del selector de nivells.

7.7.1.4.5 Menú d'opcions

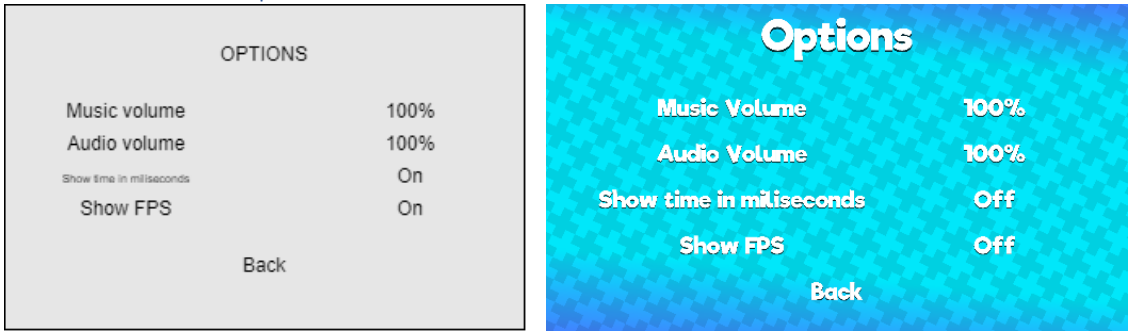

Figura 7.81. Disseny i resultat final del menú d'opcions.

7.7.1.4.6 H.U.D

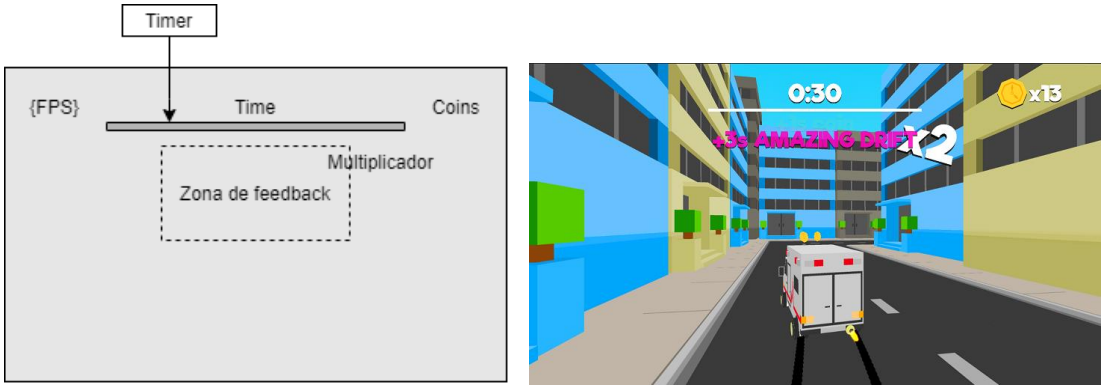

Figura 7.82. Disseny i resultat final del H.U.D.

7.7.1.4.7 Menú de pausa

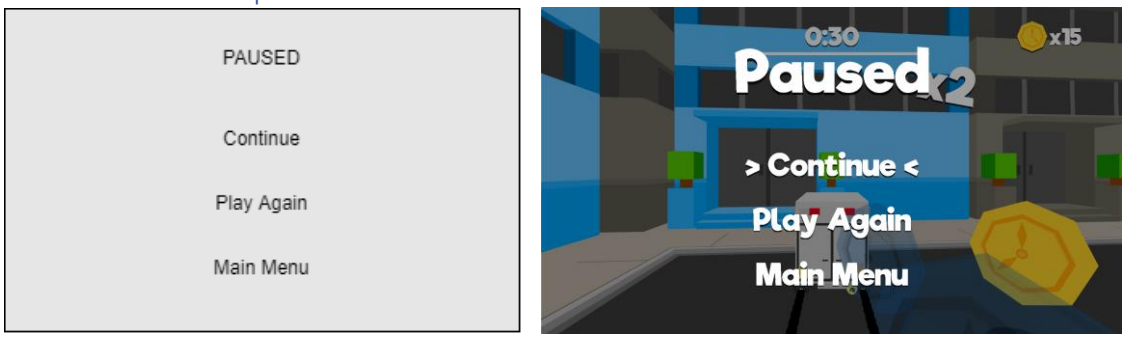

Figura 7.83. Disseny i resultat final del menú de pausa.

7.7.2 Pantalla de final de partida - *EndScreenPanel*

La pantalla de final de partida és l'únic menú del videojoc el qual necessita una classe especial per mostrar per pantalla la informació de la partida. La classe s'anomena *EndScreenPanel* i al finalitzar la partida, el *GameManager* li entrega el *ScoreSystem* (vegeu Apartat 7.5.2), del qual es llegeix tota la informació a mostrar per pantalla (vegeu Figura 7.77).

```
public void OpenScreen(ScoreSystem score)
{
    //Mostrar títol correcte
    victoryTitle.SetActive(score.victory);
    defeatSTitle.SetActive(!score.victory);

    //Escriure informació
    time.text = "Time: " + Utility.GetTimeInSeconds(score.time); //Temps
    coins.text = "Coins: " + score.coins; //Monedes
    basicDrifts.text = "Basic drifts: " + score.basicDrifts; //Derrapatges de nivell 1
    greatDrifts.text = "Great drifts: " + score.greatDrifts; //Derrapatges de nivell 2
    amazingDrifts.text = "Amazing drifts: " + score.amazingDrifts; //Derrapatges de nivell 3
    comboBreaks.text = "Combo breaks: " + score.comboBreaks; //Trencaments de combo

    //Escriure nota final
    char charResult = score.CalculateResult(); //Aconseguir la nota
    result.text = charResult.ToString(); //Escriure la nota
    result.color = colorResult[charResult]; //Pintar la nota

    //Escriure la nota en un objecte que s'utilitza per afegir una sombra a la nota
    result_shadow.text = charResult.ToString();

    //Activar
    gameObject.SetActive(true);
}
```

Figura 7.84. Funció OpenScreen del EndScreenPanel

A continuació, a la Figura 7.78, es pot observar la Pantalla de final de partida en acció:

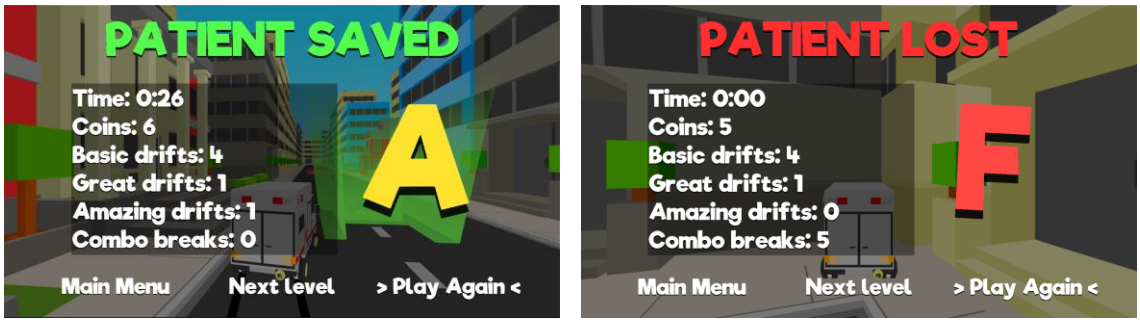

Figura 7.85. Pantalla de final de partida en el videojoc.

7.8 Monedes

Les monedes apareixen en grans quantitats en el videojoc i per tant s'ha d'aconseguir que es vegin professionals. Per aconseguir-ho s'utilitza una petita ombra afegida per relacionar la moneda amb el món, la qual és una imatge circular i no utilitza el sistema d'il·luminació d'*Unity*, a més d'una animació la qual les fa rotar i flotar. Vegeu Figura 7.86.

Figura 7.86. Model de la moneda amb textures i ombra.

Es podria caure en l'opció d'utilitzar l'*animator* d'*Unity* el qual permet crear animacions des de dintre el motor. Ara bé, en ser una animació tan simple, es pot generar per codi directament utilitzant la funció de sinus i així millorar l'eficiència, ja que l'*animator* consumiria massa recursos amb la quantitat de monedes que s'utilitzen en el joc.

Per utilitzar la funció de sinus per fer flotar la moneda, se li aplica a la posició vertical un desplaçament igual al resultat d'utilitzar el sinus en el temps del videojoc. Per crear l'animació de rotar modifiquem la rotació en l'eix Y per una quantitat determinada per una velocitat i per animar la mida de l'ombra (vegeu Figura 7.87).

```
float sin = Mathf.Sin(Time.time * frequency); //Resultat del sinus
Vector3 pos = new Vector3(0, sin * amplitude, 0); //Animació de flotar
rot.eulerAngles += new Vector3(0, rotationSpeed * Time.deltaTime, 0); //Animació de rotar

float scale = sin.Remap(-1, 1, minShadowScale, maxShadowScale);
Vector3 shadowScale = new Vector3(scale, scale, 1); //Animar ombra
```

Figura 7.87. Animacions moneda.

Tot i això, el codi d'una moneda és eficient, però el conjunt de totes les monedes no. Això és a causa que totes les monedes del joc han de fer els mateixos càlculs que donaran els mateixos resultats. Per tant per, millorar-ho, es mouen aquests càlculs a una classe superior anomenada *CoinAnimator.*

L'objectiu del *CoinAnimator* és extreure tots aquests càlculs i animar les monedes que apareixen per pantalla. Per aconseguir-ho s'utilitza un patró *Observer.* En aquest patró el *CoinAnimator* té una llista i les monedes quan s'instancien en el mapa s'hi afegeixen. Aleshores el CoinAnimator va cridant a totes les monedes actives per animar-les (vegeu Figura 7.88).

```
void Update()
{
    float sin = Mathf.Sin(Time.time * frequency); //Resultat del sinus
    Vector3 pos = new Vector3(0, sin * amplitude, 0); //Animació de flotar
    rot.eulerAngles += new Vector3(0, rotationSpeed * Time.deltaTime, 0); //Animació de rotar

    float scale = sin.Remap(-1, 1, minShadowScale, maxShadowScale);
    Vector3 shadowScale = new Vector3(scale, scale, 1); //Animar ombra


    foreach (int index in activeCoins) //Transmete informació a les monedes
    {
        coins[index].Animate(pos, rot, shadowScale);
    }
}
```

Figura 7.88. Animació de les monedes actives.

Per conèixer quines monedes estan actives en tot moment, s'utilitza una variació del patró *Observer*. Aquest cop el *CoinAnimator* se subscriu a una notificació de cada moneda que enviarà quan s'activi i es desactivi.

Quan aquesta s'activa es posa en un set d'enters amb l'índex de la moneda. Per contra, es treu del set quan aquesta es desactiva. S'ha utilitzat un *set* per aconseguir més eficiència a l'hora de treure i posar els índexs a la llista (vegeu Figura 7.89).

```
public int AddCoin(Coin c) //Subscriure una moneda al CoinAnimator
{
    coins.Add(c); //Afegir la moneda
    c.onEnabled += OnCoinEnabled; //Subscriure CoinAnimator a la moneda

    int index = coins.Count - 1; //Calcular índex de la mondea

    activeCoins.Add(index); //Col·locar l'index al set d'actius

    return index; //Retornar l'índex perquè la moneda pugui guardar la seva posició
}

1 referencia
public void OnCoinEnabled(int index, bool active)
{
    //Event quan una moneda s'activa o es desactiva
    if (active) //Si s'activa
    {
        activeCoins.Add(index); //Afegir l'índex al Set
    }
    else //Si no
    {
        if (activeCoins.Contains(index)) //Si existeix l'índex en el Set
        {
            activeCoins.Remove(index);  //Eliminar índex del Set
        }
    }
}
```

Figura 7.89. Sistema de subscripcions del CoinAnimator.

7.9 Sistema d'àudio – *AudioManager*

La implementació de l'àudio dels videojocs es fa normalment de dues maneres diferents: unida als objectes que utilitzen el so i independent dels objectes del joc. En la primera cada entitat té un sistema d'àudio propi el qual utilitza per tocar els sons que necessita. En la segona existeix un controlador d'àudio general i les entitats li demanen que toqui un so en concret.

En aquest projecte s'ha decidit utilitzar la segona, ja que és més simple i en ser un joc *casual*, no es necessita una gran quantitat de sons. Per implementar s'ha utilitzat el patró *Singleton*, ja que només es necessita un controlador d'àudio en el joc.

Aquest *AudioManager* és personalitzat a aquest projecte, és a dir, en un altre projecte no es podria utilitzar sense modificar. Tot i ser una mala pràctica, és el resultat de buscar que la classe sigui la més clara possible. Per exemple, ara mateix, si es vol fer sonar el so de quan el jugador agafa una moneda, es crida la funció *PlayPickUpCoin* i aquesta fa sonar el so. Això funciona així per evitar que les altres classes tinguin accés als sons de l'*AudioManager* o s'hagi d'utilitzar algun tipus d'id. D'aquesta manera assegurem, que encara que canvi l'*AudioManager*, quan es cridi la funció *PlayPickUpCoin* des d'una classe externa sempre funcionarà.

Tot i això, hi ha una capa de l'*AudioManager* que es pot reaprofitar en altres projectes. Es pot utilitzar tot el sistema base que fa sonar els sons

determinats, aleshores, canviant els sons i creant les noves funcions per fer-los sonar.

```
[Header("Audio clips")]
[SerializeField] private Optional<AudioClip> musicTrack; //Banda sonora
[SerializeField] private Optional<AudioClip> turboSound; //So d'accelerar
[SerializeField] private Optional<AudioClip> pickUpCoin; //So d'agafar una moneda
[SerializeField] private Optional<AudioClip> buttonSelect; //So de seleccionar un botó
[SerializeField] private Optional<AudioClip> buttonUse; //So d'utilitzar un botó
[SerializeField] private Optional<AudioClip> victorySound; //So de victòria
[SerializeField] private Optional<AudioClip> defeatSound; //So de derrota
[SerializeField] private Optional<AudioClip> collisionSound; //So de col·lisió amb una paret
```

Figura 7.90. Capa pròpia del videojoc.

Per fer sonar cada so s'utilitza una funció estàtica per cridar que soni, la qual si existeix una instància, el fa sonar (vegeu Figura 7.91).

```
public static void PlayTurboSound()
{
    Instance?.PlayTurboSoundInstance();
}
1 referencia
private void PlayTurboSoundInstance()
{
    if (!turboSound.Enabled) { return; }

    PlaySound(turboSound.Value);
}
```

Figura 7.91. Funció per fer sonar un so.

7.9.1 Sistema de *pooling*

En el projecte poden sonar una gran quantitat d'efectes de so a la vegada, per tant es necessiten tants *AudioSource*[20] com sons puguin sonar alhora. El problema és que no se sap quina és la quantitat màxima de sons simultanis. Per solucionar-ho, s'ha implementat un sistema de *pooling*, un sistema de *pooling* permet mantenir guardada un conjunt (*pool*) d'entitats les quals es van demanant a mesura que són necessàries. Si se'n demana alguna quan la *pool* està buida, se'n genera una de nova i es guarda a la *pool* (vegeu Figura 7.92).

```
protected AudioSource GetAudioSource() //Audio source pooling
{
    //Buscar si hi ha algun AudioSource disponible
    for (int i = 0; i < _audioSources.Count; ++i)
    {
        if (!_audioSources[i].isPlaying) //Si no està tocant està disponible
        {
            return _audioSources[i]; //Retorna el disponible
        }
    }

    //Si no n'hi ha cap disponible generar-ne un de nou
    AudioSource newSource = gameObject.AddComponent<AudioSource>();
    newSource.loop = false;
    newSource.playOnAwake = false;
    _audioSources.Add(newSource);
    return newSource; //Retornar l'audioSource
}
```

Figura 7.92. Sistema de pooling del videojoc.

[20] *AudioSource*: Classe d'*Unity* que permet fer tocar un so.

Cada vegada que s'hagi de tocar un so en el videojoc, es cridarà aquest sistema de *pooling*, se li ficarà la pista d'àudio correcta i es farà sonar (vegeu Figura 7.93).

```
protected AudioSource PlaySound(AudioClip sound, float pitchVariation = 0, float volumeVariation = 0)
{
    AudioSource audioSource = GetAudioSource(); //Sistema de pooling
    audioSource.clip = sound; //Triar pista d'àudio
    audioSource.pitch = 1 + Random.Range(-pitchVariation, pitchVariation); //Crear variació aleatòria de pitch
    audioSource.volume = 1 + volumeVariation; //Variació de volum
    audioSource.outputAudioMixerGroup = audioBus; //Marcar el so com efectes de so
    audioSource.Play(); //Tocar el so
    return audioSource;
}
```

Figura 7.93. Funció PlaySound de l'AudioManager

7.9.2 Cas especial: So del motor

Tots els sons del projecte funcionen de la mateixa manera excepte el so del motor. Aquest és un so continu, el qual varia entre dos nivells de volum, un baix per quan el jugador no accelera i un alt quan el jugador està accelerant.

Per aconseguir aquest efecte el so del motor té el seu propi *AudioSource* el qual està en bucle i a cada *frame* es fa una interpolació entre el volum, actual i el volum objectiu. Quan el jugador deixa d'accelerar el volum objectiu s'iguala al volum baix i quan el jugador comença a accelerar el volum objectiu s'iguala a l'alt (vegeu Figura 7.94).

```
private void Update()
{
    if (_motorSource.isPlaying) //Si està sonant
    {
        //Interpolar volum al desitjat
        _motorSource.volume = Mathf.Lerp(_motorSource.volume, wantedVolume, lerpValue * Time.deltaTime);
    }
}
public static void ActivateMotor() //Jugador accelera
{
    Instance?.ActivateMotorInstance(true);
}

1 referencia
public static void DeactivateMotor() //Jugador deixa d'accelerar
{
    Instance?.ActivateMotorInstance(false);
}

2 referencias
private void ActivateMotorInstance(bool active)
{
    if (!motorSound.Enabled) { return; } //Si no existeix el so del motor return

    if (!_motorSource.isPlaying) { _motorSource.Play(); } //Si el motor no està sonant, es fa sonar

    wantedVolume = active ? activatedVolume : baseVolume; //Escollir el volum desitjat
}
```

Figura 7.94. Codi per fer sonar el motor.

7.10 Marques de les rodes

El rastre del vehicle ha de ser permanent mentre duri la partida, ja que si aquest desapareix en algun moment, perdria tota la seva funcionalitat (vegeu Apartat 6.4.2). Per conseqüència, la solució ha de permetre un rastre infinit, no es poden utilitzar sistemes que requereixin més memòria a mesura que el rastre fos més gran, com un sistema de partícules o anar afegint polígons en el nivell.

La solució és utilitzar *RenderTextures*[21] i una càmera pròpia que renderitza un *trail*[22] a sobre la textura. Per tant, cal col·locar un pla amb la textura a sobre de les carreteres de la ciutat, a l'ambulància se li afegeix el *trail* i la càmera anirà pintant el *trail* a la textura per donar l'efecte que l'ambulància deixa les marques. A continuació s'explicarà cada component en detall.

7.10.1 *Trail – TrailPortalRenderer*

Per implementar el *trail* s'ha utilitzat el component d'*Unity* anomenat *TrailRenderer,* el qual s'ha modificat amb la classe *TrailPortalRenderer* per evitar problemes quan aquest travessa un portal.

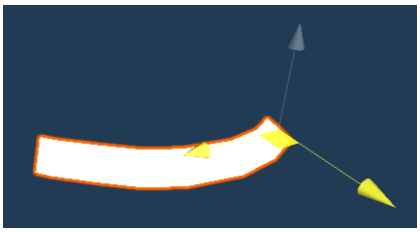

Figura 7.95. Efecte de trail del projecte.

El problema amb els portals apareix del funcionament del *TrailRenderer*, que utilitza la posició en el *frame* anterior per dibuixar l'efecte. Aleshores, en utilitzar el portal, considera que ha fet un desplaçament normal i fa crea un *trail* d'un portal a l'altre. Per evitar-ho es desactiva el component abans de ser transportat i, un cop acabat, esborra l'efecte actiu que pogués haver-hi a l'altra banda del portal i torna a activar el component (vegeu Figura 7.96).

---

[21] *RenderTextures*: Són textures a les quals se'ls hi pot posar el resultat del render d'una càmera.

[22] *Trail*: Rastre de polígons per donar un efecte com la cua d'una estrella fugaç.

```
private void OnPreTeleport()
{
    trail.enabled = false; //Desactivar el trail
}

1 referencia
private void OnTeleport(Vector3 lastPos, Vector3 newPos, Quaternion lastRot, Quaternion newRot, Portal from, Portal to)
{
    trail.Clear(); //Netejar el trail
    trail.enabled = true; //Activar el trail
}
```

Figura 7.96. Codi per solucionar problema amb els portals del TrailRenderer.

7.10.2 Càmera de marques – CameraTrailRenderer

La càmera de marques està configurada com es pot veure a la Figura 7.97.

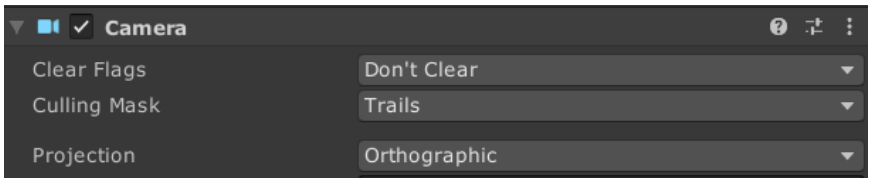

Figura 7.97. Configuració inicial de la càmera.

Clear Flags – Don't Clear: es configura la càmera perquè no esborri la textura cada vegada que renderitza, si no que vagi afegint a la textura.

Culling Mask – Tracks: es configura la càmera perquè només renderitzi els objectes de la *layer* [23] *Trails*.

Projection – Orthographic: es configura la càmera perquè tingui una projecció ortogràfica. Si la projecció fos perspectiva, els *trails* renderitzats serien deformats quan més apartats estiguessin del centre.

Per acabar de configurar la càmera i crear la *RenderTexture* s'utilitza la funció *Configurate* de la classe *CameraTrailRenderer* mostrada a les Figures 7.97 i 7.98.

```
public void Configurate()
{
    //Si la càmera està desactivada activar-la
    if (!screen.gameObject.activeSelf)
    {
        screen.gameObject.SetActive(true);
    }

    cam = GetComponent<Camera>(); //Aconseguir referència a la càmera
    //Aconseguir mides del pla on s'aplicarà el RenderTexture
    Vector2Int sizes = new Vector2Int((int)screen.transform.localScale.x,
                                      (int)screen.transform.localScale.z );

    //Crear la render texture
    rt = new RenderTexture(sizes.x * definition,
                           sizes.y * definition,
                           16,
                           RenderTextureFormat.ARGB32);
    rt.Create();

    cam.targetTexture = rt; //Assignar la textura a la càmera
```

Figura 7.98. Configurate del CameraTrailRenderer I.

[23] *Layer*: Permet agrupar entitats del joc en diferents grups.

```
    //Configurar valors de la càmera pel correcte funcionament
    cam.orthographic = true;
    cam.orthographicSize = screen.transform.localScale.z * 5; //Planes are 10 units base

    cam.rect = new Rect(-sizes.x / 2, -sizes.y / 2, sizes.x, sizes.y);

    //Aplicar la textura al pla
    screen.material.SetTexture("_MainTex", rt);

    cam.Render(); //Renderitzar la càmera per primer cop

    gameObject.SetActive(false); //Apagar la càmera

    GameManager.AddToDestroyEvent(ManualCleaning); //Netejar la render texture
}
```

Figura 7.99. Configurate del CameraTrailRenderer II.

A continuació es poden veure exemples del resultat del render de la càmera:

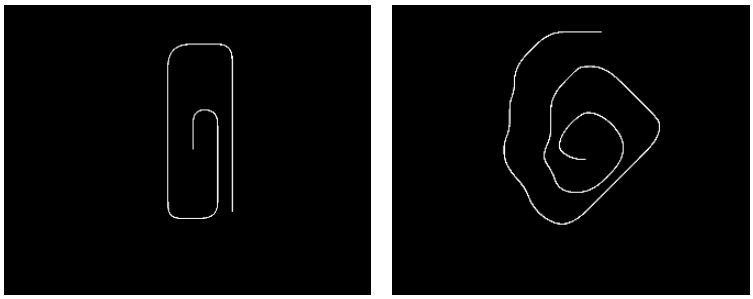

Figura 7.100. Exemples de render de la càmera.

7.10.3 Pla – TrailsShader

Observant la Figura 7.100 de l'apartat anterior es pot veure que no es pot utilitzar directament com a textura per la carretera, ja que la textura és negra i les línies blanques. Per poder-la utilitzar s'ha de transformar la textura en transparent i les línies blanques en color negra. Per fer-ho s'utilitza un *shader* anomenat *TrailsShader,* el qual es pot veure a la Figura 7.101.

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv); //Llegir textura

    col.a = col.r + col.g + col.b; //Transformar negre en transparent

    col *= _TrackColor; //Pintar les línies del color escollit

    return col;
}
```

Figura 7.101. FragmentShader del TrailShader.

7.10.4 Sistema de chunks – *TrailRenderChunk*

Inicialment, per aplicar aquest sistema al videojoc, es feia un pla de la mida de la ciutat del videojoc i s'hi aplicava una sola càmera. Malauradament fer una sola textura prou gran per enquibrir tota la ciutat significa generar una textura massa gran per a modificar a cada *frame,* i el *frame rate*[24] del videojoc baixava considerablement.

Per solucionar-ho s'utilitza un sistema de *chunks*, el qual divideix la textura gegant en un conjunt de textures, una al costat de l'altre. D'aquesta manera el videojoc només ha de treballar amb textures petites, de les quals pot activar i desactivar la càmera depenent si algun *trail* està dintre la zona que ocupa. Aconseguint que només renderitzin les càmeres necessàries.

Per controlar quan el *chunk* hauria d'estar activat o no, s'utilitza la classe *TrailRenderChunk,* la qual utilitza un comptador per saber quants *trails* hi ha dintre el *chunk*. Si aquest és 0 es desactiva la càmera, si n'hi ha més, s'activa.

```
private void OnTriggerEnter(Collider other) //Quan entra un trail
{
    count++; //Augmentar comptador
    cam.gameObject.SetActive(count != 0); //Activar la càmera si no n'hi ha 0
}

Mensaje de Unity | 0 referencias
private void OnTriggerExit(Collider other) //Quan surt un trail
{
    count--; //Decrementar el comptador
    cam.gameObject.SetActive(count != 0); //Desactivar la càmera si n'hi ha 0
}
```

Figura 7.102. Activació i desactivació de la càmera del TrailRenderChunk.

En el videojoc cada tile de carretera del projecte té un *TrailRenderChunk*, ja que no tenia sentit que per les tiles d'edifici, ja que el jugador no hi pot entrar. A la Figura 7.102 es pot veure el sistema en acció.

Figura 7.103. Sistema de marques de carretera en funcionament.

[24] *Frame rate:* Quantitat de *frames* per segon del videojoc, a major quantitat de *frames* més fluid serà el joc.

7.11 Optimitzacions – Sistema de chunks

El projecte consumeix una gran quantitat de recursos per culpa del nombre d'entitats que hi ha en el mapa: edificis amb models 3D i física, portals que obliguen a renderitzar l'escena més d'un cop, monedes amb model 3D i física, etc. A part, els recursos necessitats augmenten amb la mida del nivell, ja que hi haurà més entitats.

Per altra banda, el públic objectiu d'aquest projecte és ampli (vegeu Apartat 2.5). No tothom tindrà un ordinador prou potent per a executar aquest projecte.

Per tant, són necessàries optimitzacions en el videojoc, en primer lloc per augmentar el nombre de persones que el puguin jugar correctament i en segon lloc per independitzar el *frame rate* de la mida del mapa.

Per optimitzar el joc s'ha generat un sistema de *chunks*. Aquest sistema parteix la ciutat en trossos de mida d'una *tile*, anomenats *chunks*, els quals s'aniran desactivant i activant segons sigui necessari. Utilitzar aquest sistema produeix que el videojoc només hagi de processar els objectes dels *chunks* actius, aconseguint reduir el nombre total d'objectes a processar.

El sistema de *chunks* permet activar la part física i la part gràfica de l'objecte per separat. Per explicar la decisió s'utilitzarà el següent exemple. Imaginem que l'ambulància està al principi d'una recta llarga orientada cap al final. No té sentit que les físiques estiguin activades a tota la recta, ja que l'ambulància no està a prou distància per necessitar-les, però sí que es necessiten la part gràfica, perquè sí no hi fos, es veuria un buit a la ciutat. Vegeu Figura 7.104.

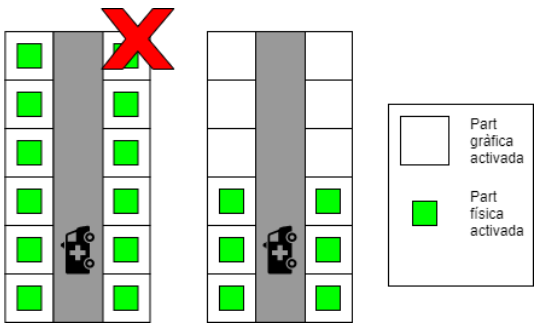

Figura 7.104. Partició entre els components físics i els components gràfics.

Per implementar el sistema de *chunks* s'ha estructurat la ciutat en un graf de *chunks* connectats entre si, i no en una matriu de *chunks*. Un graf de *chunks* permet tenir en compte fàcilment els portals. El graf de chunks es gestiona des de la classe *ChunkManager*.

7.11.1 Estructura del *chunk*

La classe *Chunk* del projecte actua com a node en el graf de *chunks* i està format per una llista de *ChunkObjects* i una llista de connexions.

7.11.1.1 ChunkObject

La classe *ChunkObject* defineix un objecte que pot ser desactivat o activat dintre un *Chunk.* Té implementada la lògica bàsica per desactivar o activar les físiques o els gràfics però sense estar lligats a cap entitat en concret (vegeu Figura 7.106).

```
public class ChunkObject : MonoBehaviour
{
    public bool physicsActive; //Físiques activades
    public bool graphicsActive; //Gràfics activats
```

Figura 7.105. Definició del ChunkObject.

```
    public virtual void SetPhysicsActive(bool active)
    {
        physicsActive = active; //Activar/Desactivar físiques
        SetObject(); //Activar/Desactivar l'objecte
    }

    8 referencias
    public virtual void SetGraphicsActive(bool active)
    {
        graphicsActive = active; //Activar/Desactivar gràfics
        SetObject(); //Activar/Desactivar l'objecte
    }

    3 referencias
    private void SetObject()
    {
        //Si l'objecte no està activat ni física ni graficament es desactiva
        gameObject.SetActive(physicsActive || graphicsActive);
    }
}
```

Figura 7.106. Lògica per activar o desactivar el ChunkObject.

Aleshores, utilitzant herència, apareixen classes capaces de gestionar qualsevol entitat del joc. Els components que hereten del *ChunkObject* creats per aquest projecte són:

- ***ChunkObject_CityTile:*** gestiona els objectes d'una *subtile* de la ciutat. Té una llista de tots els renders i tots els components físics i els desactiva o activa segons la necessitat.

- ***ChunkObject_Portal:*** gestiona els portals a dintre d'un *chunk*. Com que aquests estan entre *chunks*, s'utilitza un sistema de comptadors. Cada vegada que un *chunk* necessita el portal el comptador augmenta, i quan ja no el necessita, es decrementa. Si el comptador és 0, es desactiva el portal.

7.11.1.2 Connexions entre chunks - Connection

Per gestionar com els *chunks* es relacionen amb els veïns, s'utilitza una classe intermèdia anomenada *Connection*, la qual permet guardar tota la informació necessària per a la connexió. Cada *chunk* té un diccionari de connexions (vegeu Figura 7.107), les quals relacionen les direccions amb els *chunks* connectats en la direcció corresponent.

```
public Dictionary<Vector2Int, Connection> connected;
```

Figura 7.107. Diccionari de connexions del Chunk.

En el projecte existeixen dos tipus de connexions: normals i de portals. Les normals defineixen quan dos *chunks* estan al costat un de l'altre i no tenen cap portal entre ells. En canvi, les de portal simbolitzen quan dos *chunks* estan connectats per un portal. Per a poder gestionar les dues connexions, s'utilitza herència. La classe *Connection* (vegeu Figura 7.107) defineix l'estructura bàsica i les classes *NormalConnection* (vegeu Figura 7.108) i *PortalConnection* (vegeu Figura 7.109) perfilen la funcionalitat.

```
public class Connection
{
    protected Vector2Int dir; //Direcció de la connexió

    //Constructor
    2 referencias
    public Connection(Vector2Int dir) { this.dir = dir; }

    //Aconseguir posició de la següent posició
    4 referencias
    public virtual Vector2Int GetNextPosition() { return new Vector2Int(); }

    //Aconseguir chunk connectat
    4 referencias
    public virtual Chunk GetNextChunk() { return null; }

    //Aconseguir direcció següent (s'utilitza per solucionar un possible canvi de direcció al travessar un portal)
    3 referencias
    public virtual Vector2Int GetNextDir() { return new Vector2Int(); }
}
```

Figura 7.108. Codi de la classe Connection.

```
public class NormalConnection : Connection
{
    public Chunk connectedTo; //Chunk següent

    //Constructor
    1 referencia
    public NormalConnection(Chunk connectedTo, Vector2Int dir) : base(dir) { this.connectedTo = connectedTo; }

    //Aconseguir posició de la següent posició
    4 referencias
    public override Vector2Int GetNextPosition() { return connectedTo.position; }

    //Aconseguir chunk connectat
    4 referencias
    public override Chunk GetNextChunk(){ return connectedTo; }

    //Aconseguir direcció següent (s'utilitza per solucionar un possible canvi de direcció al travessar un portal)
    3 referencias
    public override Vector2Int GetNextDir(){ return dir; }
}
```

Figura 7.109. Codi de la classe NormalConnection.

```
public class PortalConnection : Connection
{
    public Portal fromPortal; //Portal

    8 referencias
    public Portal toPortal { get { return fromPortal.linkedPortal; } } //Altre costat del portal

    //Constructor
    1 referencia
    public PortalConnection(Portal from, Vector2Int dir) : base(dir) {
        fromPortal = from;
    }

    //Aconseguir posició de la següent posició
    4 referencias
    public override Vector2Int GetNextPosition()
    {
        return (toPortal.oneWay) ? toPortal.front : toPortal.back;
    }

    //Aconseguir chunk connectat
    4 referencias
    public override Chunk GetNextChunk()
    {
        return (toPortal.oneWay) ? toPortal.frontChunk : toPortal.backChunk;
    }

    //Aconseguir direcció següent (s'utilitza per solucionar un possible canvi de direcció al travessar un portal)
    3 referencias
    public override Vector2Int GetNextDir()
    {
        return (fromPortal.oneWay) ? -toPortal.direction : toPortal.direction;
    }
}
```

Figura 7.110. Codi de la classe PortalConnection

7.11.2 Activar i desactivar *chunks*

Un cop seleccionats quins *chunks* es necessiten activats i quins desactivats, s'han de posar en l'estat correcte. El *ChunkManager* s'encarrega, a cada frame, de buscar quins nodes s'ha d'activar el component físic i en quins el gràfic (vegeu Figura 7.111). A continuació s'explica en profunditat com es decideix per a cada un.

```
private void UpdateChunks()
{
    //Aconseguir id del chunk on es troba el jugador
    Vector2Int playerPos = WorldToMatrix(transform.InverseTransformPoint(player.position));

    if (completeList.ContainsKey(playerPos)) //Si el chunk existeix
    {
        Chunk actualChunk = completeList[playerPos]; //Aconseguir chunk actual
        UpdateByProximity(playerPos, actualChunk); //Actualitzar part física
        UpdateByVisibility(playerPos, actualChunk); //Actualitzar part gràfica
    }
}
```

Figura 7.111. Actualitzar els components físics i gràfics dels chunks.

7.11.2.1 Part física – Algorisme de proximitat.

La part física d'un *chunk* només serà necessària activar-la si l'ambulància està a prop, ja que no té sentit activar la part física sabent que l'objecte mai col·lidirà. A continuació es pot veure l'algorisme utilitzat per activar la part física, anomenat algorisme de proximitat en el projecte.

1. Aconseguir *chunk* actual (C) del jugador.
2. Activar físiques de C.
3. Aconseguir llista de *chunks* (CL) connectats a C.
4. Per a cada *chunk* (C') a CL
   4.1. Activar físiques de C'.

L'algorisme agafa els *chunks* connectats al *chunk* on es troba el jugador i els hi activa la part física, tenint en compte els portals. A la figura 7.111 es pot veure el resultat d'aplicar l'algorisme.

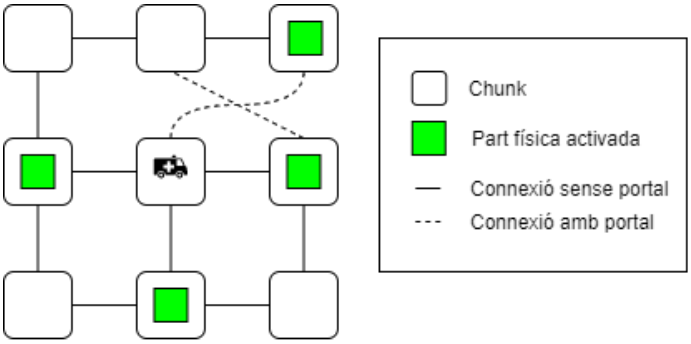

Figura 7.112. Exemple de funcionament de l'algorisme de proximitat.

Internament s'aprofita el sistema de graf per codificar l'algorisme de proximitat, com es veu a les Figures 7.113 i 7.114.

```
private void UpdateByProximity(Vector2Int playerPos, Chunk c)
{
    //Si el jugador no ha canviat de chunk no fa falta tornar a calcular.
    if (playerPos == lastChunkLoaded) { return; }
    lastChunkLoaded = playerPos; //Guardar l'últim carregat

    //Set amb les posicions dels chunks que han d'estar activats
    SortedSet<Vector2Int> newSet = new SortedSet<Vector2Int>(comparer);

    //Omplir el Set
    c.GetProximitySet(ref newSet);

    //Activar els chunks del set per proximitat
    SetActiveChunks(ref newSet, ref activatedByProximity, ActivatedBy.Proximity);
}
```

Figura 7.113. Funció UpdateByProximity del ChunkManager.

```
public void GetProximitySet(ref SortedSet<Vector2Int> newSet, int range = 1)
{
    newSet.Add(position); //Afegir el chunk actual

    if (range <= 0) { return; } //Si el rang d'exploracio es 0 return

    foreach (Connection c in connected.Values) //Per a cada chunk connectat
    {
        if (!newSet.Contains(c.GetNextPosition()))  //Si no existeix en el newSet
        {
            //Actualitzar el set amb el chunk connectat
            c.GetNextChunk().GetProximitySet(ref newSet, range - 1);
        }
    }
}
```

Figura 7.114. Funció GetProximitySet del ChunkManager.

7.11.2.2 Part gràfica – Algorisme de visió

La part gràfica d'un *chunk* serà necessària si apareix en el renderitzat de la càmera. L'algorisme que calcula si es necessita activar la part física del *chunk* s'anomena algorisme de visió i aprofita el graf de *chunks* per funcionar amb recursivitat. A continuació es pot veure l'algorisme.

Esquema algorisme de visió:

1. Crear Set de *chunks* (SC).
2. Afegir a SC el *chunk* actual (C) i els veïns.
3. Per a cada direcció possible (D):
   3.1. Si el jugador està mirant cap a D:
      3.1.1. ExplorarChunk(C, SC, D).
4. Per a cada *chunk* (C') a SC
   4.1. Activar gràfics de C'.

Esquema ExplorarChunk(*chunk* C, set SC, direcció D):

1. Afegir a SC el *chunk* C i els seus veïns.
2. Si existeix un *chunk* connectat (CC) en la direcció D:
   2.1. ExplorarChunk(CC, SC, D)

L'algorisme de visió a partir de la direcció del jugador determina quines direccions s'han "d'explorar" i decideix quins han d'estar activats a partir d'aquesta exploració (vegeu Figura 7.115). Explorar una direcció significa afegir tots els *chunks* connectats entre si, tenint en compte els portals, en aquella direcció (*chunks* rosats a la Figura 7.115), i tots els *chunks* veïns dels *chunks* explorats (chunks blaus a la Figura 7.115). Afegir els *chunks* veïns evita que apareguin buits en la ciutat, ja que, depenent de

l'orientació de l'ambulància, aquesta podria veure el *chunk* en diagonal, el qual no estaria activat si no s'hagués afegit en el pas dels chunks veïns.

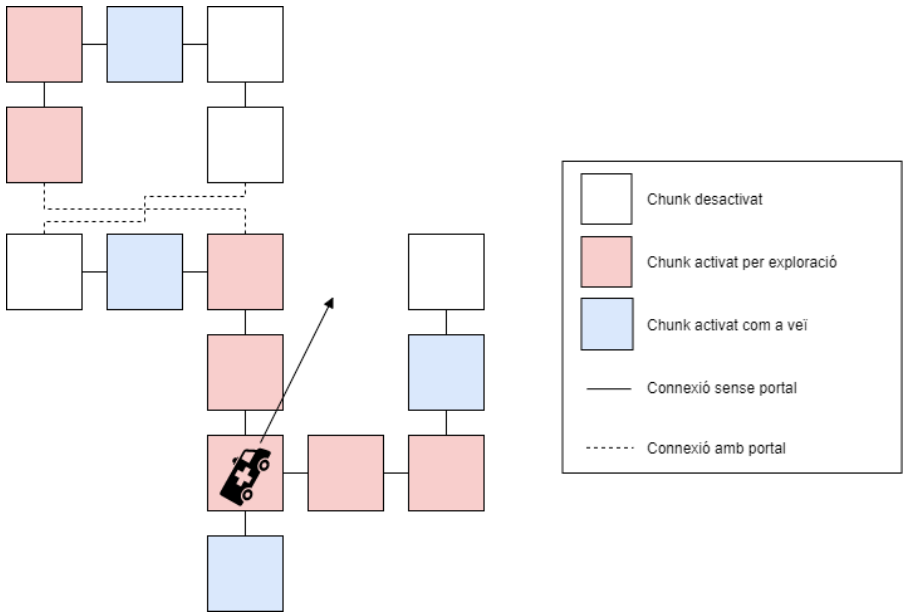

Figura 7.115. Exemple de funcionament de l'algorisme de visió.

Internament s'aprofita el sistema de graf per codificar l'algorisme de visió, com es veu a les Figures 7.115 i 7.116.

```
private void UpdateByVisibility(Vector2Int playerPos, Chunk c)
{
    //Calcular direcció jugador
    Vector2 playerDir = new Vector2(player.forward.x, player.forward.z);

    //Optimització - el jugador ha d'haver fet un canvi mínim per tornar a calcular
    float magnitude = (playerDir - lastPlayerDir).sqrMagnitude; //Calcular quantitat de rotació
    summedRotation += Mathf.Abs(magnitude); //Guardar quantitat
    lastPlayerDir = playerDir; //Guardar última direcció
    if (summedRotation < minRotation){ return; } //Si no ha girat prou return
    summedRotation = 0;  //Reiniciar la suma

    //Set amb les posicions dels chunks que han de ser activats
    SortedSet<Vector2Int> newSet = new SortedSet<Vector2Int>(comparer);

    //Afegir veïns - Una direcció (0,0) equival a cap direcció, per tant s'afegira a ell mateix i continuarà
    c.GetVisibilitySet(ref newSet, new Vector2Int(0, 0));

    //Per a cada direcció
    for (int i = 0; i < directions.Length; i++)
    {
        if (c.IsTransitable(directions[i])) //Si hi ha connexió en la direcció
        {
            //Utilitzar el dot product per saber la direcció que està mirant el jugador
            if (Vector2.Dot(playerDir.normalized, directions[i]) > 1f - opening)
            {
                //Omplir set
                c.GetVisibilitySet(ref newSet, directions[i]);
            }
        }
    }

    //Activar els chunks del set per proximitat
    SetActiveChunks(ref newSet, ref activatedByVisibility, ActivatedBy.Visibility);
}
```

Figura 7.116. Funció UpdateByVisibility del ChunkManager.

```
public void GetVisibilitySet(ref SortedSet<Vector2Int> newSet, Vector2Int dir)
{
    newSet.Add(position); //Afegir posició actual

    foreach (Connection item in connected.Values) //Afegir veïns
    {
        newSet.Add(item.GetNextPosition());
    }

    if (connected.ContainsKey(dir)) //Si existeix chunk en la dirreció
    {
        Connection c = connected[dir]; //Explorar ¨chunk
        c.GetNextChunk().GetVisibilitySet(ref newSet, c.GetNextDir());
    }
}
```

Figura 7.117. Funció GetVisibilitySet del Chunk.

8 Proves

En el projecte s'ha seguit la metodologia Kanban (vegeu Apartat 3.2), aleshores no s'ha donat per finalitzada cap tasca fins que aquesta no estava cent per cent testejada. Per tant al finalitzar cada component del projecte s'ha dedicat un exhaustiu temps a solucionar qualsevol problema que hi pogués haver.

Per trobar i solucionar problemes s'han utilitzat les eines de *debug* del *Visual Studio*, que permeten executar el codi instrucció per instrucció i veure els valors de totes les variables. Això permet saber exactament què està passant en el codi i accelera la velocitat a la qual se solucionen els possibles problemes.

Tot i això, i com marca la planificació (vegeu Apartat 3.3), es va deixar un temps per fer testeig, no només per l'autor del projecte, sinó també pel *feedback* d'altres persones.

El *feedback* d'altres persones alienes és molt important en els videojocs, ja que permet tenir l'opinió de persones sobre el videojoc en sí i no sobre la seva creació. Es necessita invertir una gran quantitat de temps i esforç per deixar un bon resultat. Això produeix que la pròpia opinió del creador es vegi esbiaixada.

Tot i això, no tot el *feedback* s'ha de tenir en compte. El *feedback* és una gran eina per acabar de polir el joc, però res més. Si es decideix modificar el videojoc sense pensar bé respecte a tot el *feedback* que es rep, el joc deixa de ser una idea clara i intenta ser un conjunt d'idees per fer feliç al màxim de públic possible, però normalment acaba sent un conjunt avorrit.

Aleshores, tot *feedback* rebut s'ha d'avaluar respecte al tipus de persona del que prové. Posem el següent exemple: acabem de fer les proves del videojoc i tenim dos testers: un tester, el qual només juga a jocs de trets, i un altre tester, que és campió en jocs casual. En aquest cas el *feedback* rebut pel campió ajudarà molt més.

El testatge d'aquest videojoc es va produir amb amics a través d'un servei de telecomunicació en línia. Això és a causa que en el temps d'escriure aquest document existeix la pandèmia mundial de la Covid-19. Gràcies al testatge es van canviar les següents coses en el projecte:

8.1.1 Canvi d'esquema de controls

Quan el projecte es va començar a desenvolupar es van deixar l'esquema de controls d'*Unity* per defecte i, tot i ser incòmodes, no es van canviar per preferir dedicar temps a altres parts del projecte que es consideraven més importants.

En anar provant el projecte, l'autor es va acostumar als controls fins que els va interioritzar i l'hi semblaven naturals. En fer el testatge tots els testers es van queixar de l'esquema de controls, perquè era incòmode. Per tant es va canviar per un sistema de controls més entenedor i simple.

8.1.2 *Bugs*

Gràcies als testers es van trobar una gran quantitat de *bugs*, els quals no s'haurien trobat mai si no fos per ells. Entre els més importants hi ha:

- **No limitar el *FoV* de la càmera:** En el videojoc, quan l'ambulància acaba de derrapar, se li dóna un increment momentani de la velocitat. Per fer que el jugador notés que s'augmentava la velocitat també s'augmentava el *Field of View (FoV)* de la càmera. Com més gran és el *FoV* més s'allarga la imatge de la pantalla. El *bug* apareixia en encadenar derrapatges, el *FoV* anava augmentant sense tornar a l'inicial. Creant imatges com les que es poden veure a la Figura 8.1.

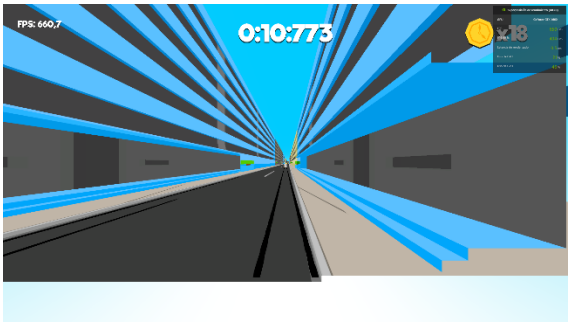

Figura 8.1. Bug - No limitar el FoV de la càmera.

- **Aixafament de l'ambulància**: en el videojoc l'ambulància fa un salt abans de derrapar. El salt té una animació d'estirar i aixafa per donar més *feedback*. El problema sorgia quan es premia el botó de saltar sense parar, que l'ambulància no tenia temps de recuperar-se i s'anava aixafant, donant resultats com els que es poden veure a la Figura 8.2.

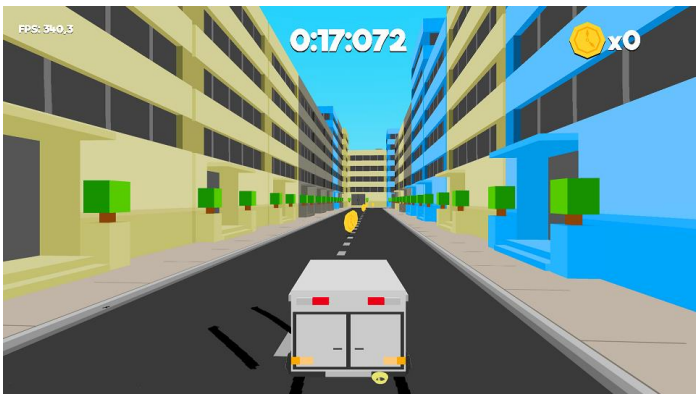

Figura 8.2. Bug - Aixafament de l'ambulància.

9 Resultats

9.1 Assoliment dels objectius

A continuació s'avaluarà el grau d'assoliment de cada objectiu del projecte (vegeu Apartat 1.2).

9.1.1 Estudiar i entendre el funcionament de l'algorisme *WFC*

En l'apartat 5.1 es pot observar l'explicació del *WFC* de manera clara i entenedora, per tant es considera l'assoliment aconseguit al cent per cent.

9.1.2 Crear un generador de ciutats

En l'apartat 7.2 es pot observar el resultat del generador de ciutats creat. Aquest generador compleix l'objectiu explicat a l'Apartat 1.2.2, ja que és capaç de crear una quantitat de variacions gairebé infinita i segueix les normes lògiques d'una ciutat.

9.1.3 Interconnectar els carrers de la ciutat generada

En l'apartat 7.3 es pot observar el funcionament de la interconnexió de carrers. Aquesta interconnexió és invisible i per tant es compleix amb l'objectiu.

9.1.4 Afegir mecàniques de videojoc a la ciutat

En l'apartat 6 es pot observar el disseny del videojoc i en el 7 la implementació d'aquest. El videojoc resultant aprofita bé la ciutat generada i la seva particularitat, per tant, es compleix amb l'objectiu.

9.2 Legislació i normativa vigent

En el projecte mai guarda informació de caràcter personal del jugador, per tant no hi ha manera d'incomplir la LOPD (Llei Orgànica de Protecció de Dades). A part, no forma part de cap activitat econòmica, per tant tampoc s'incompleix la LSSICE (Llei de Serveis de la Societat de la Informació i Comerç Electrònic).

Respecte possibles problemes de *Copyright* tots els models 3D (vegeu Apartat 7.1) i el paquet de *Tessera* (vegeu Apartat 5.2) han estat adquirits de manera legal, la tipografia que utilitza el projecte és cent per cent

gratuïta, i tots els sons del projecte són d'ús lliure o propis per tant no generarien cap problema en el futur.

L'única cosa que caldria vigilar és el motor d'*Unity*. Com s'ha explicat a l'apartat 2.1.1, *Unity* gratuït sempre que la persona jurídica (empresa o persona normal) generi més de 100.000 $ a l'any utilitzant *Unity* s'hauria de pagar per una llicència professional.

9.3 *PEGI*

El *Pan European Game Information* (*PEGI*) és un sistema europeu de classificació sobre el contingut dels videojocs i software d'entreteniment. Aquest sistema va aparèixer per donar una manera als pares per saber quin tipus de videojocs els hi compren en els seus fills. Aquest sistema permet identificar el contingut dels videojocs (vegeu fila inferior de la Figura 9.1) i el mínim d'edat recomanat (vegeu fila superior de la Figura 9.1).

El *PEGI* no és cap llei, és una recomanació, per tant encara que tingui de qualificació 18 d'anys, podria ser comprat per un menor.

Figura 9.1. Símbols del PEGI.

Respecte al projecte no apareix cap contingut que pugui augmentar l'edat de cap manera, al final el jugador condueix per una ciutat amb una ambulància. No hi ha:

- Llenguatge obsés: no hi ha diàleg i no hi ha cap mala paraula a cap menú.
- Discriminació: no hi ha personatges en el videojoc.
- Drogues: no apareixen drogues en el videojoc.
- Por: el joc té un aire humorístic i divertit.
- Sexe: no apareix sexe en el videojoc.
- Violència: en els nivells només hi ha el jugador, per tant no pot produir accidents o per l'estil.

- Apostes: no apareixen apostes en el videojoc.
- En línia: el videojoc no és en línia.

Per tant la qualificació que hauria de tenir *Lost Ambulance* és *PEGI* 3, ara bé, s'utilitza el condicional perquè el joc no s'ha enviat a PEGI per avaluar.

9.4 Resultat final

A continuació es poden veure captures del videojoc per observar el resultat del projecte des de la Figura 9.2 a la 9.11. A part, a l'enllaç https://youtu.be/4afgLwCXwrg es pot veure el projecte en acció.

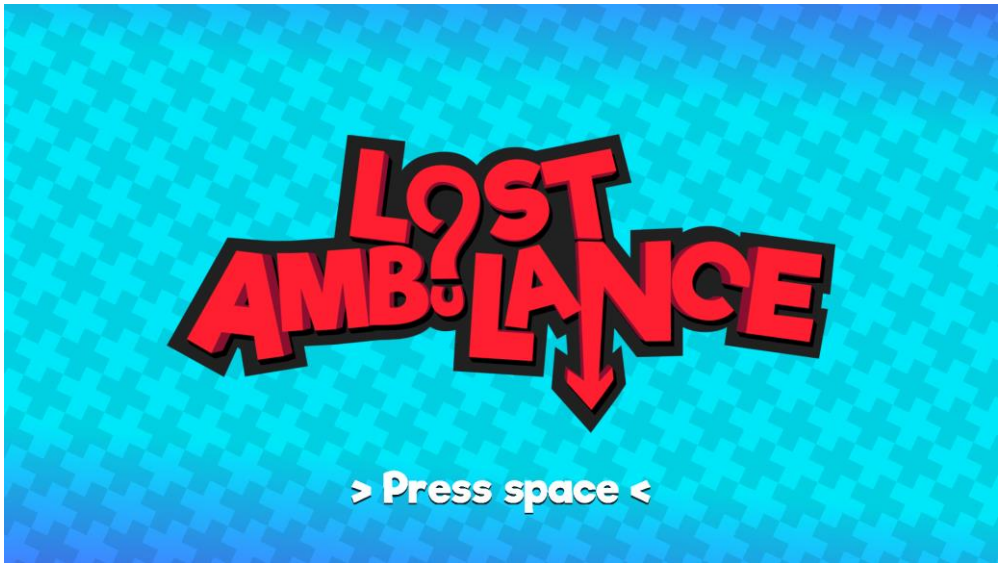

Figura 9.2. Pantalla de títol.

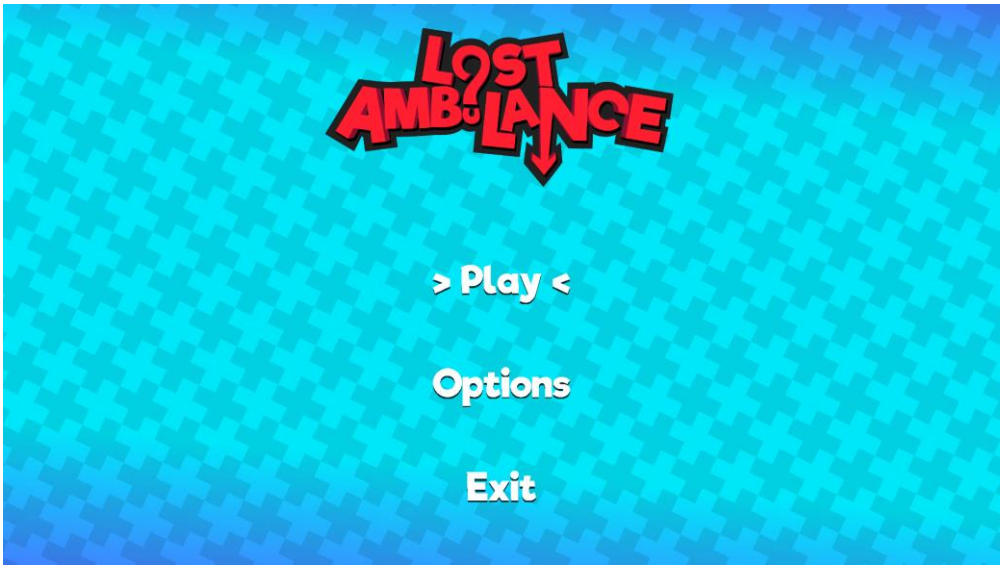

Figura 9.3. Menú principal.

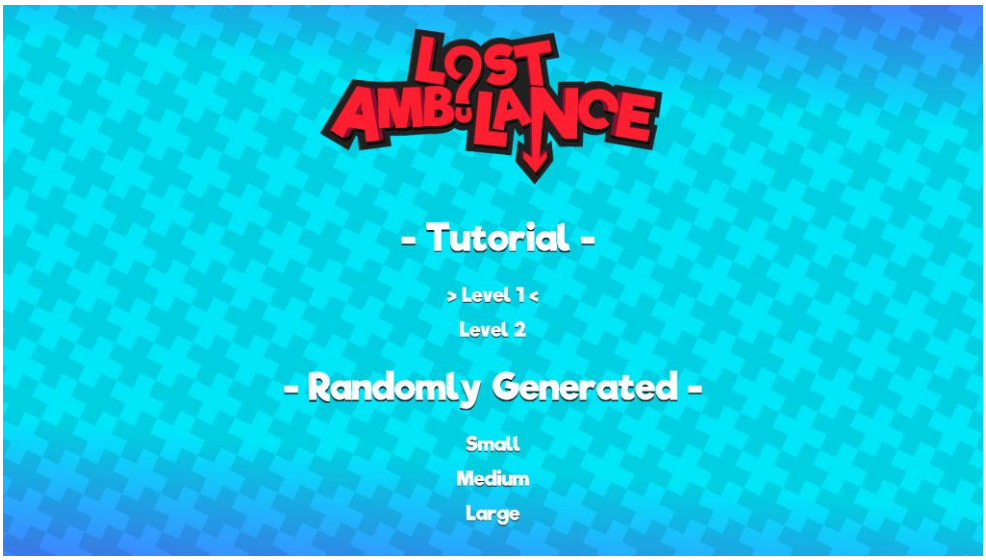

Figura 9.4. Selector de nivells.

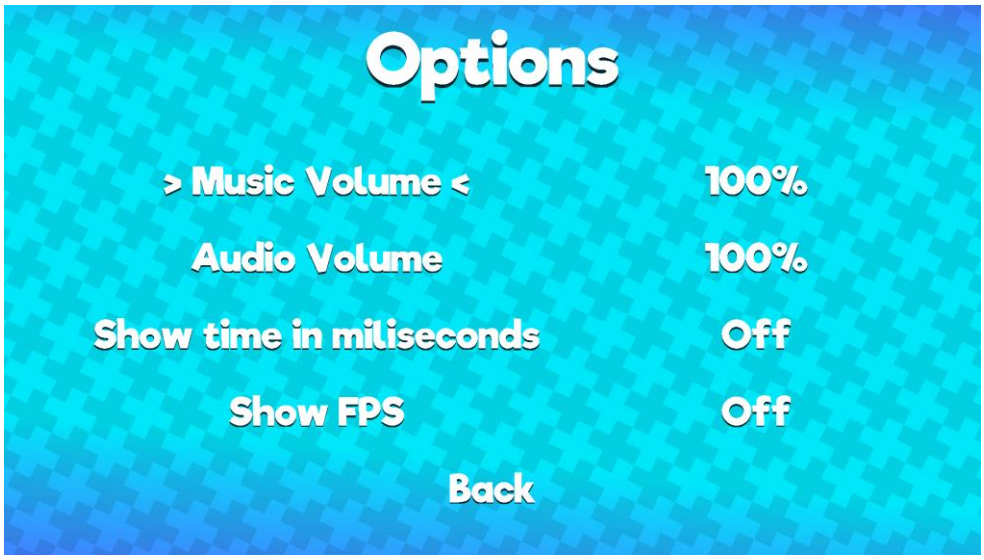

Figura 9.5. Menú d'opcions.

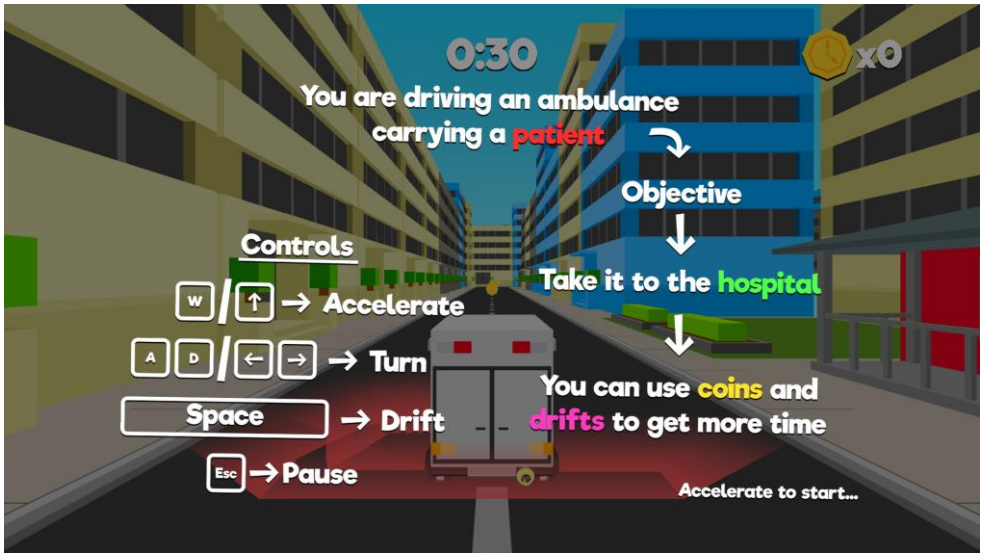

Figura 9.6. Pantalla de controls.

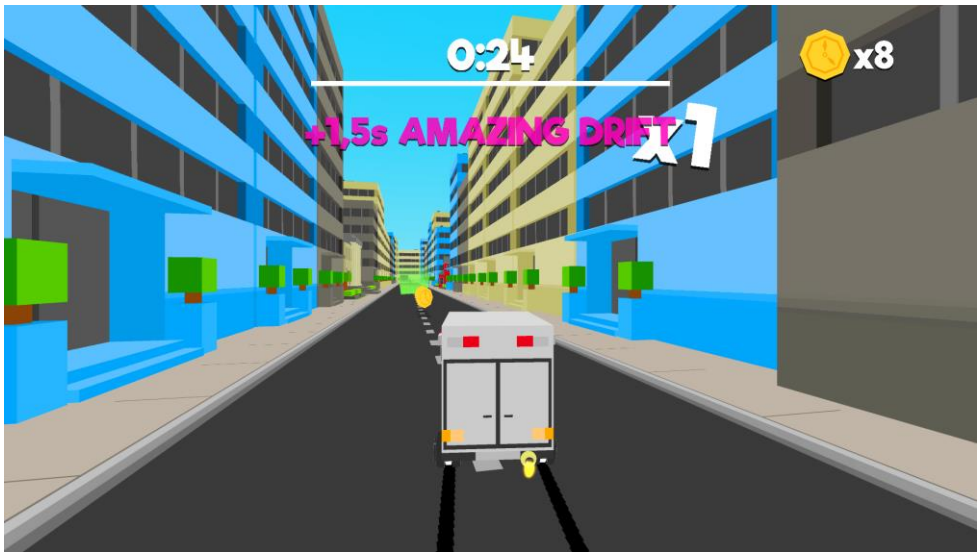

Figura 9.7. Nivell de tutorial 1.

Figura 9.8. Menú de pausa.

Figura 9.9. Exemple de nivell procedimental.

Figura 9.10. Pantalla de final de partida: derrota.

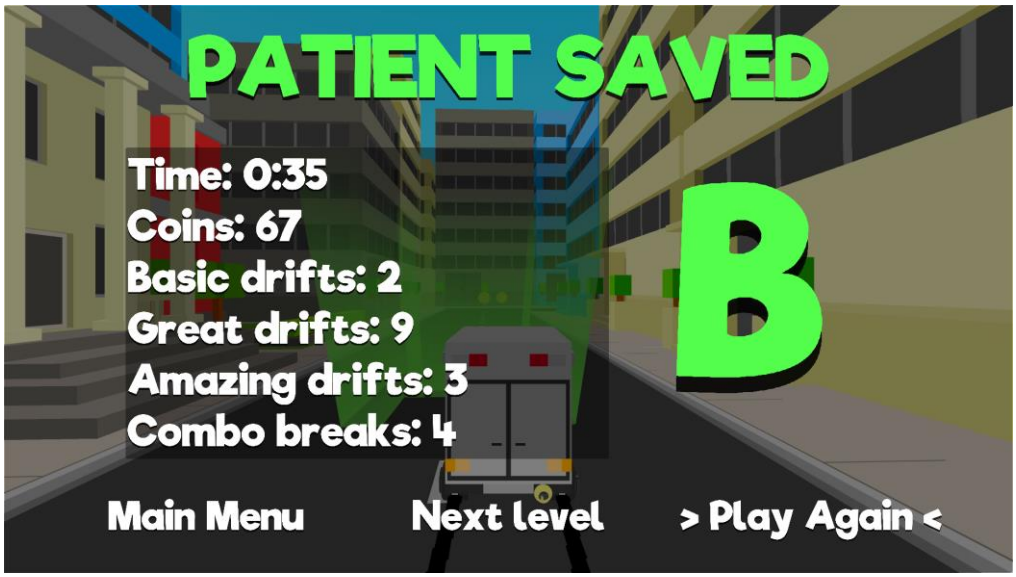

Figura 9.11. Pantalla de final de partida: victòria.

10 Conclusions

10.1 Valoració personal

Nota: Aquest apartat serà escrit en primera persona per reflectir millor les opinions personals.

Estic molt content amb el resultat final. Crec que el resultat és divertit i sobretot es veu professional. Es veritat que l'art professional ajuda, però els controls funcionen bé, les marques de la carretera són un efecte molt interessant, quan s'agafen les monedes apareixen partícules, el sistema de combo crida l'atenció, etc.

Que es vegi professional és el que més m'il·lusiona del projecte. Això és a causa que, fins ara, tots els videojocs que s'havien fet a la carrera eren prototips per la falta de temps a fer un videojoc complet i, per tant es veuen poc acabats. En canvi pel projecte es va poder dedicar un període de temps a polir el videojoc, i crec que es nota.

Per altra banda, ara mateix l'objectiu principal del jugador és completar tots els nivells, però s'aconsegueix relativament de pressa. Estaria bé introduir en el videojoc un objectiu a llarga durada per aconseguir atrapar més els jugadors.

Tot i això, per mi la bellesa principal del projecte no és el que es veu, sinó el contrari, com està estructurada la generació de la ciutat, com funciona la interconnexió i com el sistema de *chunks* ajuda al fet que pugui ser jugat a la majoria d'ordinadors. Crec que aquest projecte és un clar exemple que un videojoc és més del que veu el jugador.

10.2 Desviació de la planificació

La temporització final de cada tasca del projecte ha estat semblant a la que s'havia dissenyat en la planificació inicial. Tot i això, la poca experiència a l'hora de fer videojocs complets ha generat que algunes tasques fossin més ràpides i altres més lentes. A continuació s'explica amb detall cada tasca.

10.2.1 Generador ciutat

Gràcies a la utilització del paquet de *Tessera* (vegeu Apartat 5.2), s'ha aconseguit retallar el temps d'implementació de la ciutat, ja que no s'ha hagut d'implementar l'algorisme *WFC* de 0, la qual cosa hauria fins i tot ocupat més temps de l'esperat.

10.2.2 Interconnexió de carrers

En planificar el projecte s'esperava que la interconnexió de portals fos més ràpida de la resultant. Això és a causa que els portals són un tema conegut i hi ha una gran quantitat de tutorials en línia per aprendre. El que no es va tenir en compte va ser que els portals en aquest videojoc havien de ser en tercera persona, en comptes de la primera que és la més utilitzada.

A part, el sistema no es va poder donar per finalitzat fins que no funcionava amb l'ambulància del jugador. Per això està en paral·lel amb la tasca de Moviment del jugador.

10.2.3 Moviment del jugador

El moviment del jugador ha requerit més temps de l'esperat a causa de la implementació de tots els sistemes perquè funcionessin els portals (vegeu Apartats 7.4.2 i 7.4.3).

10.2.4 Lògica del videojoc

Durant la planificació no es va tenir en compte que la lògica del videojoc estava relacionada amb les tasques de pantalles i el sistema d'àudio. Per tant, el desenvolupament de la lògica del videojoc s'ha allargat per incloure la finalització d'aquestes.

10.2.5 Sistema d'àudio

La tasca ha utilitzat el temps esperat, però l'inici d'aquesta ha estat abans de l'esperat gràcies a la finalització de les altres tasques.

10.2.6 Pantalles

Gràcies a la utilització d'herència i poder compartir components entre pantalles (vegeu Apartat 7.7) el desenvolupament ha estat més ràpid de l'esperat.

10.2.7 Poliment

La tasca ha seguit perfectament la planificació.

10.2.8 Creació d'art faltant

Al ser una tasca secundària, la duració no era rígida i per tant no ha ocupat més temps de l'esperat.

10.2.9 Creació d'efectes de so.

De la mateixa manera que la tasca anterior, en ser una tasca secundària la planificació ha estat més laxa i no ha ocupat més temps de l'esperat.

10.2.10 Narrativa

La tasca ha seguit perfectament la planificació.

10.2.11 Documentació

La tasca ha seguit perfectament la planificació.

10.2.12 Testeig

La tasca ha seguit perfectament la planificació.

10.2.13 Sistema de *chunks*

En el projecte no s'esperava la necessitat d'implementar un sistema per optimitzar el joc, per tant, en la planificació inicial no existeix aquesta tasca. Tot i això, el sistema va ocupar una gran quantitat de temps, i es podria afegir a la tasca de Generador de ciutats o la interconnexió de carrers, però el sistema és tan important que s'ha decidit afegir-lo com una tasca nova en el diagrama de Gantt (vegeu Figura 10.1).

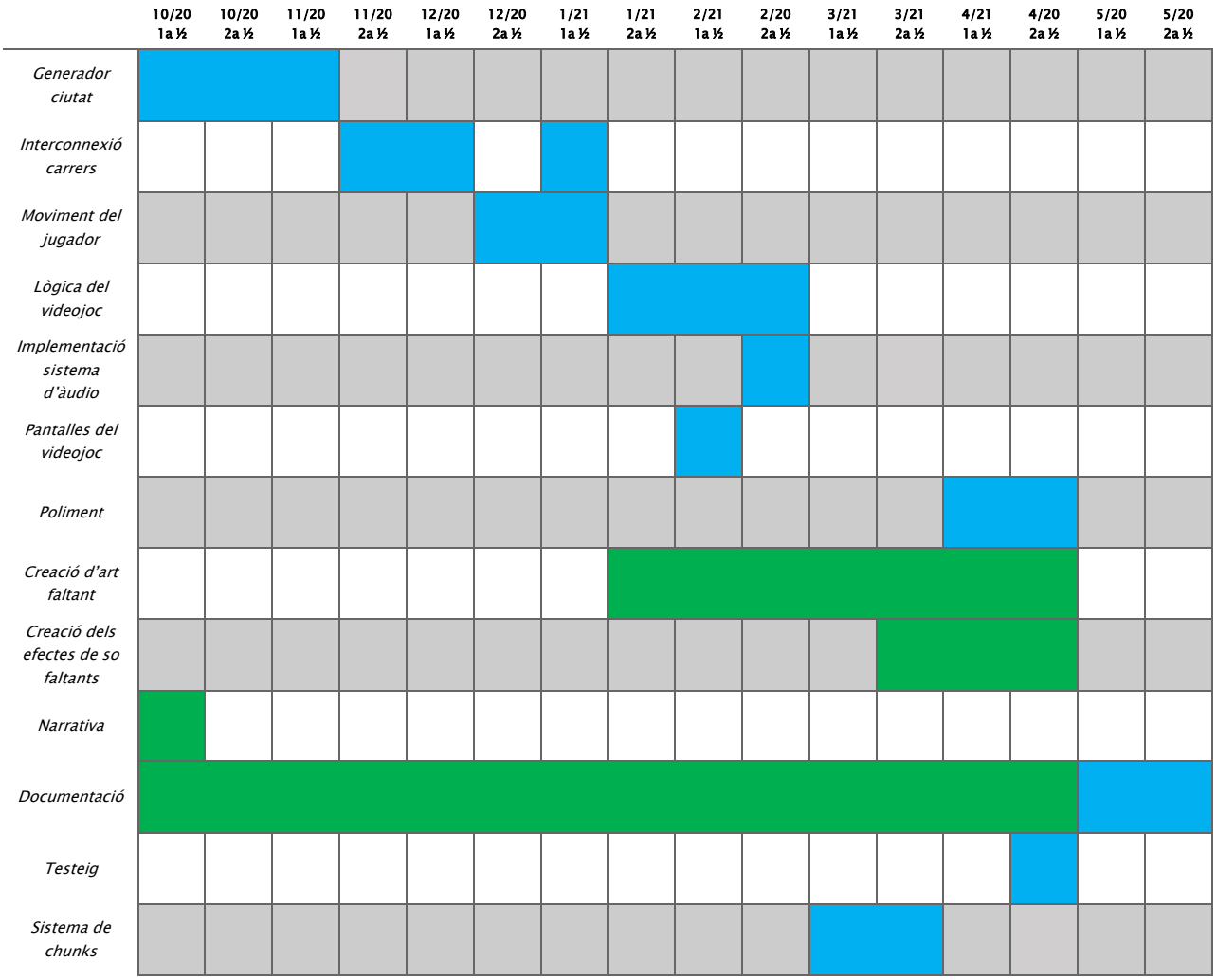

Figura 10.1. Diagrama de Gantt del projecte.

11 Treball futur

El resultat d'aquest projecte és un videojoc complet, i com s'ha explicat amb anterioritat (vegeu Apartat 4), l'objectiu d'aquest videojoc mai ha estat la comercialització. Tot i això, el projecte ha quedat prou professional i amb la capacitat per posar-lo a la venda si se li afegissin més funcionalitats. Per tant, en un futur es podria treballar a afegir les següents per millorar la comercialització del videojoc.

1. **Personalització de l'ambulància**: A través de la creació d'un nou menú, el jugador podria millorar aspectes de l'ambulància pagant amb monedes del joc. Aquestes millores podrien anar des de millorar la velocitat, fins a permetre augmentar el nivell màxim del sistema de combo.
2. **Canviar el vehicle**: A través d'un nou menú, el jugador podria canviar el vehicle per un altre pagant monedes del joc. Aquest seria un canvi estètic. Els diferents vehicles podrien ser cotxes, camions, cotxes de policia, etc.
3. **Multijugador**: aprofitant l'objectiu del jugador d'anar d'un punt A a un punt B es pot crear un videojoc de carreres entre jugadors, per generar competitivitat.
4. **Omplir la ciutat**: ara mateix la ciutat tot i tenir monedes està bastant buida. Seria interessant afegir cotxes o persones per la ciutat per afegir més vida a la ciutat. En el cas de permetre la interacció de l'ambulància amb els vianants (atropellaments), s'hauria de revisar el PEGI del videojoc, augmentant-lo, ja que el videojoc tindria violència.

12 Bibliografia

1. DigixArt. *Road 96.* https://www.road96.com

2. Electronic Arts. *Spore.* https://www.ea.com/es-es/games/spore/spore

3. Wikimedia Foundation. *Crazy Taxi.* https://es.wikipedia.org/wiki/Crazy_Taxi

4. Happy Volcano. *You Suck At Parking.* https://www.yousuckatparking.cc

5. Funselektor. *Absolute Drift*. http://absolutedrift.com

6. Marian. *Infinite procedurally generated city with the Wave Function Collapse algorithm.* https://marian42.de/article/wfc/

7. Plausible Concept AB. *Bad North*. https://www.badnorth.com

8. Synty Store. *Simple Town*. https://syntystore.com/products/simple-town-cartoon-assets

9. Sebastian Lague. *Vídeo "Coding adventure: portals".* https://www.youtube.com/watch?v=cWpFZbjtSQg

10. Mike Bostock. *Visualizing algorithms.* https://bost.ocks.org/mike/algorithms/

11. Boris The Brave. *Tessera Procedural Tile Based Generator.* https://assetstore.unity.com/packages/tools/level-design/tessera-procedural-tile-based-generator-155425

12. Mix and Jam. *Vídeo "Mario Kart's Drifting | Mix and Jam"*. https://www.youtube.com/watch?v=Ki-tWT50cEQ

13. MTG (Universitat Pompeu Fabra). *Freesound. https://freesound.org*

14. Unity technologies. *Unity Manual*. https://docs.unity3d.com/

15. Unity technologies. *Unity Blog.* https://blogs.unity3d.com/

13 Annexos

Com annex del projecte s'ha adjuntat el projecte complet d'*Unity* i el videojoc final.

14 Manual d'usuari i d'instal·lació

14.1 Manual de l'usuari

14.1.1 Controls per teclat

En menús:

- Fletxa dreta o amunt: canviar al botó anterior.
- Fletxa esquerra o avall: canviar al següent botó.
- Enter o espai: confirmar opció.
- Esc: tornar enrere.

En partida:

- Fletxa dreta o tecla a: girar a l'esquerra.
- Fletxa esquerra o tecla d: girar a la dreta.
- Fletxa amunt: accelerar.
- Espai: derrapar.
- Esc: obrir menú de pausa.

14.1.2 Controls per comandament

En menús:

- *Joystick* cap a la dreta o amunt: canviar al botó anterior.
- *Joystick* cap a l'esquerra o avall: canviar al següent botó.
- Botó A: confirmar opció.
- Botó B: tornar enrere.

En partida:

- *Joystick* cap a l'esquerra: girar a l'esquerra.
- *Joystick* cap a la dreta: girar a la dreta.
- Botó *Right Shoulder*: accelerar.
- Botó A: derrapar.
- Botó Start: obrir menú de pausa.

14.2 Manual d'instal·lació

El videojoc funciona en Windows i Mac.

Passos per executar a Windows:

- Descomprimir l'arxiu ZIP amb una eina de compressió de fitxers.
- Executar el fitxer "LostAmbulance.exe".
- Pot aparèixer un missatge respecte no es coneix el desenvolupador del software per protegir l'ordinador, per solucionar-ho, prémer **més informació** i **executar**.

Passos per executar a Mac:

- Descomprimir l'arxiu ZIP amb una eina de compressió de fitxers.
- Executar el fitxer "LostAmbulance".
- Pot aparèixer un missatge respecte no es coneix el desenvolupador del software per protegir l'ordinador, per solucionar-ho, prémer **obrir.**