UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA GEODÉSICA CARTOGRÁFICA Y TOPOGRÁFICA

Diseño e implementación de un modelado interactivo 3D de la ETSIGCT mediante el motor gráfico UNITY.

ALUMNO
DANIEL RUANO FOLCH

TUTORES
JESÚS MANUEL PALOMAR VÁZQUEZ,
ANA BELÉN ANQUELA JULIÁN

E.T.S.I. GEODÉSICA, CARTOGRÁFICA Y TOPOGRÁFICA UNIVERSITAT POLITÈCNICA DE VALÈNCIA
MÁSTER UNIVERSITARIO EN INGENIERÍA GEOMÁTICA Y GEOINFORMACIÓN

SEPTIEMBRE, 2021

AGRADECIMIENTOS.

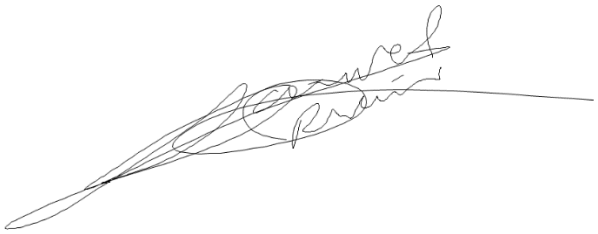
En primer lugar, quiero agradecerles a los profesores Jesús Manuel Palomar Vázquez y Ana Belén Anquela Julián todo el apoyo prestado y su dedicación durante la realización de este proyecto, así como a todos los profesores que han tenido la puerta de su despacho abierta a todas las dudas que he tenido en la realización de este Trabajo Fin de Grado. Me gustaría hacer especial mención al profesor José Carlos Martínez Llario que me ayudó mucho en la selección de la base de datos a utilizar en el proyecto.

En segundo lugar, me gustaría dar las gracias a todo el personal de administración y servicios de la ETSIGCT por la información y ayuda prestada. También quisiera agradecer a todo el personal docente e investigador que durante todos estos años me ha formado con las capacidades necesarias para hacer frente a este TFM.

En último lugar, quiero agradecer el apoyo prestado por mi familia y amigos. También quiero dar las gracias a todos mis compañeros que me han aportado ideas para mejorar mi proyecto.

COMPROMISO

"El presente documento ha sido realizado completamente por el firmante; no ha sido entregado como otro trabajo académico previo y todo el material tomado de otras fuentes ha sido convenientemente entrecomillado y citado su origen en el texto, así como referenciado en la bibliografía"



ÍNDICE DE FIGURAS

FIG. 1. PRIMERA PLANTA – DATOS DE PARTIDA 15
FIG. 2. PLANTA BAJA Y TERCERA PLANTA – DATOS DE PARTIDA 15
FIG. 3. CARTOGRAFÍA ACTUALIZADA Y GENERALIZADA DE LA PLANTA BAJA 16
FIG. 4. PLANTA BAJA GIS PROYECTO FINAL DE GRADO 16
FIG. 5. BORRADO DE CAPAS DE AUTOCAD 21
FIG. 6. POLÍGONOS DE LA PLANTA BAJA EN SKETCHUP 21
FIG. 7. ELEVANDO ESCALERAS DE LA PLANTA BAJA 22
FIG. 8. ELEVANDO MUROS DE LA PLANTA BAJA 22
FIG. 9. SUPERPOSICIÓN DEL MAPA ORIGINAL EN EL MODELADO 23
FIG. 10. MODELADO REFINADO DE LA PLANTA BAJA 23
FIG. 11. DETALLE DEL MODELADO REFINADO DE LA PLANTA BAJA 24
FIG. 12. CREANDO UN MATERIAL SIMPLE 25
FIG. 13. APLICANDO UN MATERIAL SIMPLE 25
FIG. 14. CREANDO UN MATERIAL TRANSPARENTE 26
FIG. 15. APLICANDO UN MATERIAL TRANSPARENTE 26
FIG. 16. CREANDO UN MATERIAL CON IMAGEN DE TEXTURA 27
FIG. 17. APLICANDO UN MATERIAL CON IMAGEN DE TEXTURA 27
FIG. 18. MODIFICANDO UN MATERIAL CON IMAGEN DE TEXTURA 28
FIG. 19. EJEMPLO DE MATERIAL CON IMAGEN DE TEXTURA QUE PERMITE TRANSPARENCIA 28
FIG. 20. MODIFICANDO EL MODELO PARA PODER TEXTURIZAR 28
FIG. 21. PROCESO DE TEXTURIZACIÓN DE IMÁGENES 29
FIG. 22. PLANTA BAJA TEXTURIZADA 29
FIG. 23. EJEMPLOS DE OBJETOS COMPLEJOS DE LA ESTRUCTURA DEL EDIFICIO 30
FIG. 24. EJEMPLOS DE MATERIALES CON IMÁGENES DE TEXTURA SIN REPETICIÓN DE LA IMAGEN 31
FIG. 25. DETALLES ESTRUCTURALES DE LAS HABITACIONES ACCESIBLES 32
FIG. 26. TECHO DE LA CUARTA PLANTA DEL EDIFICIO 32
FIG. 27. MODELADO FINAL DEL APARCAMIENTO 33
FIG. 28. MODELADO FINAL DE LA PLANTA BAJA 33
FIG. 29. MODELADO FINAL DE LA PRIMERA PLANTA 33
FIG. 30. MODELADO FINAL DE LA SEGUNDA PLANTA 34
FIG. 31. MODELADO FINAL DE LA TERCERA PLANTA 34
FIG. 32. MODELADO FINAL DE LA CUARTA PLANTA 34
FIG. 33. MODELADO FINAL DEL EDIFICIO SIN LA PARTE DEL EDIFICIO 7J 35
FIG. 34. MODELADO FINAL DEL EDIFICIO 35
FIG. 35. MODELADOS DE OBJETOS DEL EDIFICIO 37
FIG. 36. MODELADO DE ASCENSORES 38
FIG. 37. PÁGINA DE WAREHOUSE 3D 38
FIG. 38. MODELADOS OBTENIDOS DE WAREHOUSE 3D 38
FIG. 39. MODELOS MIXTOS – PARTE MODELADA Y PARTE OBTENIDA DE WAREHOUSE 3D 39
FIG. 40. PROCEDIMIENTO DE MARCADO DE OBJETOS 40
FIG. 41. RESULTADO DEL MARCADO DE OBJETOS DE ALGUNAS PLANTAS 40
FIG. 42. CREANDO EL PROYECTO EN UNITY 41

FIG. 43. SISTEMA DE CARPETAS EMPLEADO EN UNITY 41
FIG. 44. EXPORTANDO EL MODELO DESDE SKETCHUP 42
FIG. 45. MODELO EN BLENDER 42
FIG. 46. MODELO EN UNITY 43
FIG. 47. MODIFICACIÓN DE LOS MATERIALES 43
FIG. 48. MODELO EN UNITY TRAS MODIFICACIÓN DE LOS MATERIALES 43
FIG. 49. ESTRUCTURACIÓN DE LOS MODELADOS 44
FIG. 50. GENERACIÓN DE LOS MESH COLLIDERS 44
FIG. 51. GENERACIÓN DE LOS BOX COLLIDERS 45
FIG. 52. RESULTADO DE LA IMPORTACIÓN DEL MODELADO DE LAS PLANTAS 45
FIG. 53. EJEMPLO DE PREFAB 46
FIG. 54. POSICIONAMIENTO DE LOS PREFABS 46
FIG. 55. ESTRUCTURACIÓN DE LOS PREFABS 47
FIG. 56. STANDARD ASSETS 47
FIG. 57. FPSCONTROLLER 48
FIG. 58. VISTAS CON EL FPSCONTROLLER 48
FIG. 59. EXPORTANDO LA BASE DE DATOS 50
FIG. 60. IMPORTANDO CSV A LA NUEVA BASE DE DATOS 51
FIG. 61. ADECUANDO LAS TABLAS IMPORTADAS 51
FIG. 62. RESULTADO FINAL DE LA IMPORTACIÓN DEL MODELO DE DATOS 51
FIG. 63. CREANDO EL CANVAS UI 54
FIG. 64. POSICIONANDO LA STARTCAMERA 55
FIG. 65. AÑADIENDO EL PANELSTART 55
FIG. 66. MODIFICANDO EL TAMAÑO DE LOS PANELES 56
FIG. 67. AÑADIENDO EL LOGO DE LA ESCUELA A LA PANTALLA DE INICIO 57
FIG. 68. AÑADIENDO LOS BOTONES A LA PANTALLA DE INICIO 57
FIG. 69. RESULTADO FINAL DE LA PANTALLA DE INICIO 58
FIG. 70. INFORMACIÓN DEL PANEL GAME 59
FIG. 71. MIRA EMPLEADA EN EL PANEL GAME 60
FIG. 72. SPRITE DEL PLAYER 60
FIG. 73. CÁMARA DEL MINI MAPA 61
FIG. 74. AÑADIENDO EL RENDER TEXTURE AL UI 61
FIG. 75. AÑADIENDO LOS SPRITES DE MAPAS AL MINI MAPA 62
FIG. 76. AÑADIENDO LOS SPRITES DE MISIÓN AL MINI MAPA 62
FIG. 77. DISEÑO COMÚN DEL PANEL ROOM 63
FIG. 78. ESTRUCTURA DEL PANEL ROOM 64
FIG. 79. DISEÑO DE DESPACHOS DEL PANEL ROOM 64
FIG. 80. DISEÑO DE SERVICIOS DEL PANEL ROOM 65
FIG. 81. DISEÑO DE SEMINARIOS DEL PANEL ROOM 66
FIG. 82. DISEÑO DE AULAS DEL PANEL ROOM 66
FIG. 83. DISEÑO DE HORARIO DE AULAS DEL PANEL ROOM 67
FIG. 84. DISEÑO DEL PANEL MAP 68
FIG. 85. DISEÑO DEL PANEL QUEST 69
FIG. 86. AÑADIENDO UNA SCROLLBAR AL PANEL QUEST 70
FIG. 87. DISEÑO DEL PANEL MESSAGE QUEST 71
FIG. 88. DISEÑO DEL PANEL ELEVATOR 72
FIG. 89. AÑADIENDO EL SCRIPT PLAYERCONTROLLER 78

FIG. 90. AÑADIENDO EL SCRIPT STARTPANEL 81
FIG. 91. REFERENCIANDO LOS BOTONES AL SCRIPT STARTPANEL 82
FIG. 92. TIPOS DE PUERTAS EXISTENTES EN EL MODELADO 82
FIG. 93. PREFABS DE PUERTAS 94
FIG. 94. AÑADIENDO LOS SCRIPTS A LOS PREFABS DE PUERTAS 95
FIG. 95. MODIFICANDO LOS BOX COLLIDERS DE LAS PUERTAS 95
FIG. 96. ELEMENTOS QUE DISPONDRÁN DE SCRIPTS EN LA EXTENSIÓN ROOM 108
FIG. 97. AÑADIENDO SCRIPTS A LOS PANELES DE LOS PASILLOS 114
FIG. 98. AÑADIENDO ELEMENTOS A LOS CARTELES DE LAS PUERTAS 114
FIG. 99. ADECUANDO LOS TRIGGERS Y VARIABLES A LOS CARTELES DE LAS PUERTAS 115
FIG. 100. PREFAB DE LOS TRIGGERS DE LAS HABITACIONES. 115
FIG. 101. ASOCIANDO SCRIPTS AL PANELROOM 127
FIG. 102. ASOCIANDO EL MAPPLAYERCONTROLLER AL PLAYER. 131
FIG. 103. CONFIGURANDO LOS PREFABS DE LOS DIRECTORIOS 134
FIG. 104. EJE DE COORDENADAS DEL PLAYER EN EL ESPACIO 3D Y 2D. 138
FIG. 105. ELEMENTOS EXISTENTES EN EL TRIGGER DE LOS ASCENSORES 149
FIG. 106. AÑADIENDO SCRIPTS A LOS ELEMENTOS GAME DE LOS ASCENSORES 151
FIG. 107. CONFIGURANDO LAS CLASES UI DE LA EXTENSIÓN ELEVATOR 156
FIG. 108. MODELO 3D EMPLEADO DE ADOBE MIXAMO 163
FIG. 109. ANIMACIONES EMPLEADAS DE ADOBE MIXAMO 163
FIG. 110. CREANDO LOS MATERIALES DEL PERSONAJE 3D 164
FIG. 111. ADAPTANDO EL PERSONAJE 3D 164
FIG. 112. ADAPTANDO LAS ANIMACIONES 3D 165
FIG. 113. CONFIGURANDO EL MODELO EN LA ESCENA 165
FIG. 114. AÑADIENDO ANIMACIONES Y TRANSICIONES AL ANIMATOR CONTROLLER 166
FIG. 115. AÑADIENDO ANIMACIONES Y TRANSICIONES AL ANIMATOR CONTROLLER 166
FIG. 116. AÑADIENDO SCRIPT QUESTCOMPLETED AL PREFAB 169
FIG. 117. CREANDO PREFAB DE BOTÓN DE SELECCIÓN DE MISIÓN. 172
FIG. 118. BUILD SETTINGS Y DISEÑO DEL LOGO DE LA APLICACIÓN 177
FIG. 119. MODIFICANDO PARÁMETROS DE LA APLICACIÓN 177
FIG. 120. MODIFICANDO CALIDAD GRÁFICA DE LA APLICACIÓN 178
FIG. 121. APLICACIÓN CON ERROR EN LA BASE DE DATOS 178
FIG. 122. AÑADIENDO LA BASE DE DATOS A LA APLICACIÓN 178
FIG. 123. APLICACIÓN CON LA BASE DE DATOS INTEGRADA 179

ÍNDICE DE TABLAS

TABLA 1. MODELO DE DATOS DE PARTIDA 17
TABLA 2. DOMINIO SERVICIOS_TIPO 18
TABLA 3. DOMINIO DESPACHOS_TIPO 19
TABLA 4. DOMINIO AULAS_TIPO 19
TABLA 5. DOMINIO UNIDAD DOCENTE 19
TABLA 6. DOMINIO DEPARTAMENTO 19
TABLA 7. DOMINIO UNIDAD DOCENTE 20
TABLA 8. MODELO DE DATOS DE LA APLICACIÓN 49

ÍNDICE

INTRODUCCIÓN **13**
OBJETIVO **13**
DATOS DE PARTIDA **15**
METODOLOGÍA **21**
1. MODELACIÓN 3D DEL EDIFICIO 21
1.1. MODELACIÓN BÁSICA DEL EDIFICIO 21
1.2. REFINANDO EL MODELO BÁSICO 23
1.3. TEXTURIZANDO EL MODELO 24
1.4. FINALIZACIÓN DEL MODELO – OBJETOS MÁS COMPLEJOS 30
1.5. AÑADIENDO EL CONTEXTO AL MODELO 35
2. MODELACIÓN 3D DE OBJETOS DEL EDIFICIO 36
2.1. CREACIÓN 3D DE ELEMENTOS EXISTENTES 36
2.2. MARCADO DEL POSICIONAMIENTO DE LOS OBJETOS 39
3. DESARROLLO BÁSICO EN UNITY 41
3.1. CREACIÓN DEL PROYECTO EN UNITY 41
3.2. CARGA DEL MODELADO DE LAS PLANTAS DEL EDIFICIO 41
3.2.1. EXPORTACIÓN DEL MODELADO DE LA PLANTA A UNITY 42
3.2.2. MODIFICACIÓN DE LAS TEXTURAS 43
3.2.3. CORRECCIÓN DE LA ESTRUCTURA DEL MODELADO 44
3.2.4. GENERACIÓN DE COLLIDERS 44
3.2.5. ENSAMBLAJE DE LAS PLANTAS DEL EDIFICIO 45
3.3. CARGA DEL MODELADO DE LOS OBJETOS 45
3.3.1. CREACIÓN DE UN PREFAB 45
3.3.2. POSICIONAMIENTO DE LOS PREFABS 46
3.3.3. ESTRUCTURACIÓN DE LOS PREFABS 47
3.4. ADICCIÓN DE LOS ASSETS BÁSICOS DE UNITY 47
3.5. ADICCIÓN DE BASE DE DATOS 49
3.5.1. DEFINICIÓN DEL MODELO DE DATOS 49
3.5.2. EXPORTACIÓN DE LA BASE DE DATOS 50
3.5.3. ACCESO A LA BASE DE DATOS DESDE LA APLICACIÓN 52
3.6. CREACION DE PANTALLAS DE INTERFAZ UI 53
3.6.1. PANEL START 54
3.6.2. PANEL GAME 58

3.6.3. PANEL ROOM ... 63
3.6.4. PANEL MAP ... 67
3.6.5. PANEL QUEST ... 69
3.6.6. PANEL MESSAGE QUEST ... 71
3.6.7. PANEL ELEVATOR ... 71
4. IMPLEMETACIÓN DE NUEVAS FUNCIONALIDADES EN UNITY ... 72
4.1. EXTENSIÓN CORE ... 73
4.1.1. GAME POSITION MANAGER ... 73
4.1.2. GAME DATABASE MANAGER ... 75
4.1.3. PLAYER CONTROLLER ... 77
4.1.4. START PANEL ... 79
4.2. EXTENSIÓN DOORS ... 82
4.2.1. ENUMERADOS DE LA EXTENSIÓN DOORS ... 83
4.2.2. PUERTAS DE BISAGRAS ... 83
4.2.3. PUERTAS CORREDERAS ... 88
4.2.4. PUERTAS GIRATORIAS ... 92
4.2.5. ADICCIÓN Y CONFIGURACIÓN DE LAS PUERTAS ... 94
4.3. EXTENSIÓN ROOM ... 95
4.3.1. ENUMERADOS DE LA EXTENSIÓN ROOM ... 95
4.3.2. CLASES DTO DE LA EXTENSIÓN ROOM ... 96
4.3.3. GAME ROOM MANAGER ... 104
4.3.4. CLASES GAME DE LA EXTENSIÓN ROOM ... 108
4.3.1. CLASES UI DE LA EXTENSIÓN ROOM ... 115
4.4. EXTENSIÓN MAP ... 128
4.4.1. GAME MAP MANAGER ... 128
4.4.2. MAP PLAYER CONTROLLER ... 129
4.4.3. CLASES GAME DE LA EXTENSIÓN MAP ... 132
4.4.1. CLASES UI DE LA EXTENSIÓN MAP ... 135
4.5. EXTENSIÓN ELEVATOR ... 142
4.5.1. ENUMERADOS DE LA EXTENSIÓN ELEVATOR ... 142
4.5.2. GAME ELEVATOR MANAGER ... 142
4.5.3. CLASES GAME DE LA EXTENSIÓN ELEVATOR ... 145
4.5.4. CLASES UI DE LA EXTENSIÓN ELEVATOR ... 151
4.6. EXTENSIÓN QUEST ... 157
4.6.1. CLASE DTO DE LA EXTENSIÓN QUEST ... 157
4.6.2. GAME QUEST MANAGER ... 158
4.6.3. QUEST PLAYER CONTROLLER ... 161
4.6.4. CLASES GAME DE LA EXTENSIÓN QUEST ... 162
4.6.5. CLASES UI DE LA EXTENSIÓN QUEST ... 170

5. EXPORTACIÓN DEL PROYECTO 177

CONCLUSIONES Y OPINIÓN PERSONAL. 181

1. CONCLUSIONES. 181
2. OPINIÓN PERSONAL. 181

BIBLIOGRAFÍA. 183

1. PUBLICACIONES Y CURSOS ONLINE. 183
2. AYUDA DE PROGRAMAS. 183
3. PÁGINAS WEB. 183

INTRODUCCIÓN

Este proyecto nace con la intención de mejorar y llevar un paso más allá mi anterior TFG, mediante los nuevos conocimientos adquiridos en el Máster.

RUANO FOLCH, Daniel.
Diseño de un sistema de información geográfica de la ETSIGCT, visualización 2D/3D y análisis de caminos óptimos.
Dirigido por Peregrina Eloina Coll Aliaga | Universitat Politècnica de València | 2015.
Trabajo fin de grado.

Partiendo de cartografía 2D de la ETSIGCT y mediciones relativas tomadas en campo, se ha desarrollado un modelo 3D mediante el programa SketchUp. El modelado resultante comprende desde elementos estructurales hasta objetos existentes en el edificio. Posteriormente se ha exportado al motor gráfico de UNITY, donde partiendo de la base de datos generada en el TFG, se han ido incorporado funcionalidades programadas en C#, obteniendo así un modelo interactivo de la ETSIGCT.

Gracias al modelado realizado en SketchUp y las funcionalidades desarrolladas en UNITY se ha obtenido una aplicación más realista, accesible y con más valor para el público final.

OBJETIVO

Con la realización de este proyecto, se pretende mejorar el proyecto generado en la titulación de grado, haciendo énfasis en los siguientes aspectos:

- Añadir mayor realismo al modelo 3D.
- Expandir las funcionalidades, creado un modelo interactivo.
- Mejorar la accesibilidad del proyecto, obteniendo la posibilidad de ejecutarlo sin programas de terceros.

Para ello, se va a modelar de forma más realística el modelo 3D en SketchUp, se va a desarrollar funcionalidades en un motor de videojuegos (UNITY) y se va a exportar como aplicación de Windows.

Este proyecto queda dividido en diversas etapas claramente diferenciadas:

1. **Modelación 3D del edificio:** En esta etapa se pretende modelar en 3D el edificio, siendo un modelado lo más fiel posible a la realidad, teniendo que ir a medir a campo ciertos elementos de interés.
2. **Modelación 3D de objetos del edificio:** En este paso se generará modelos 3D de objetos existentes en el edificio, así como el posicionamiento de los mismos.

3. **Carga y adecuación del modelo 3D en UNITY:** Con esta etapa conseguiremos cargar en UNITY el modelo 3D y todos los objetos creados dentro de este.

4. **Adición de assets básicos de UNITY:** En esta fase se introducirán los assets básicos de UNITY, permitiendo así una navegación por el edificio. También añadiremos la base de datos y crearemos los distintos elementos UI de la aplicación.

5. **Creación de nuevas funcionalidades:** Con este paso se crearán todas estas funcionalidades custom al proyecto. Divididas en las extensiones CORE, DOORS, ROOM, MAP, ELEVATOR y QUEST.

6. **Creación de la aplicación:** Se creará la aplicación de escritorio para Windows.

En conclusión, el objeto de este proyecto es expandir y mejorar el proyecto realizado en el Grado con las nuevas tecnologías y conocimientos adquiridos en el Máster, obteniendo así un producto más competente y atractivo.

DATOS DE PARTIDA

Para la elaboración del TFM partiremos de los mapas originales del edificio, estos mapas se encuentran en formato .DWG de AutoCAD. Disponemos del aparcamiento, la planta baja y las cuatro plantas del edificio. Veamos por ejemplo el archivo de la primera planta:

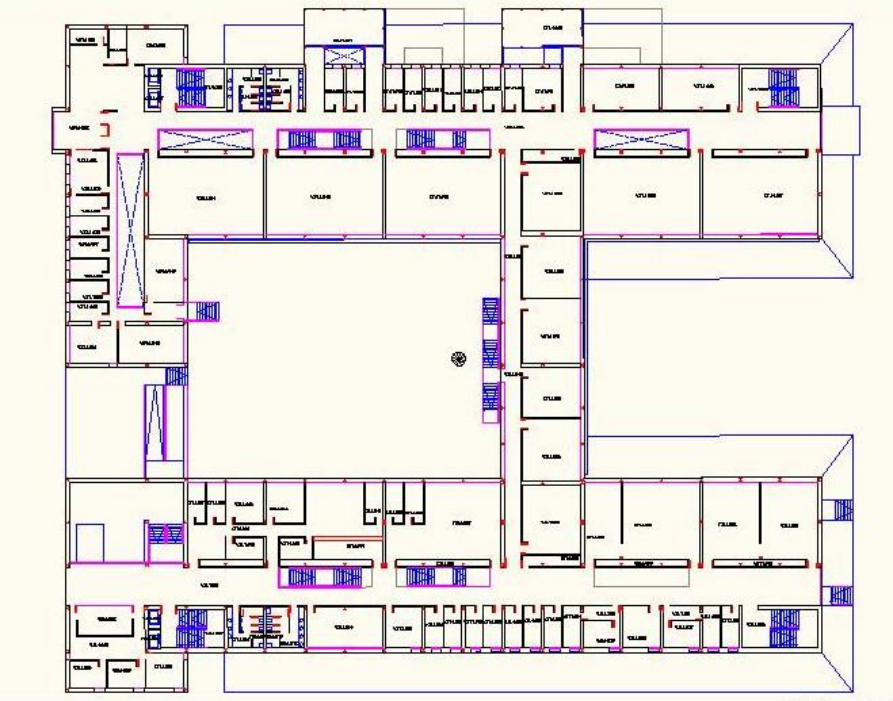


Fig. 1. Primera Planta – Datos de partida

Como se puede observar se trata de un mapa con muchos detalles, nos aparecen barandillas, escaleras, proyecciones de planta inferiores, puertas, elementos en los aseos, pilares, ascensores, ventanas etc. Además, nos aparece toda la geometría que pertenece a la escuela de ADE.

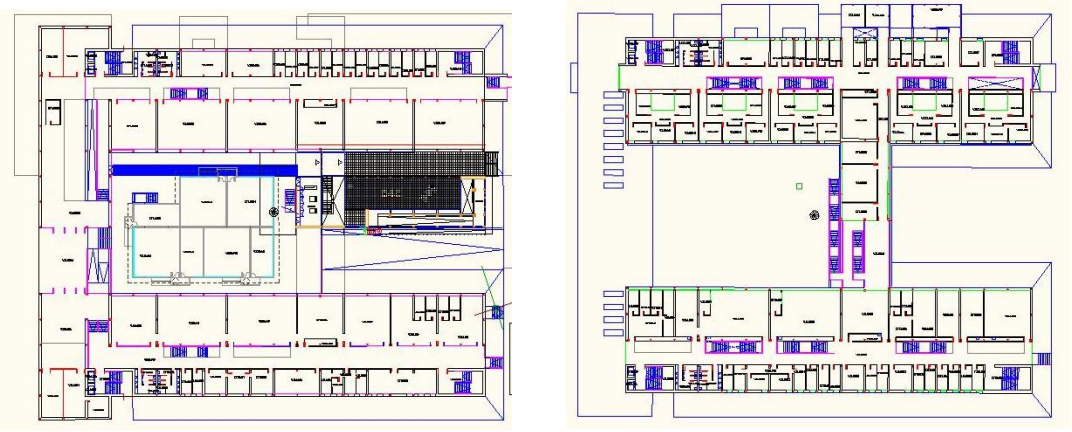


Fig. 2. Planta Baja y Tercera Planta – Datos de partida

Disponemos también de esta misma cartografía generalizada, en donde se han eliminado gran cantidad de detalles. A continuación, se enumeran los distintos procesos que se realizaron:

- Se eliminaron los detalles de los aseos, los pilares dentro de las estructuras, las puertas, la zona que pertenece a la escuela de ADE y las líneas que indican la dirección de subida de las escaleras.
- Se creó una única estructura continua por todo el recorrido, eliminando las diferenciaciones entre cristal, muro y barandilla.
- Se eliminaron todos aquellos objetos que no existan realmente en dicha planta, debido a que son proyecciones.
- Se creó una nueva capa para cerrar todos los espacios, con esto conseguiremos a posteriori tener diferenciado el pasillo de las aulas y así poder texturizar las zonas.
- Se eliminaron los ascensores, pero se mantuvo el espacio donde se encuentran. En las escaleras de los extremos sólo se dejaron los que subían a la siguiente planta, cambiando los números de las que bajaban.
- Se numeraron aquellos escalones que no estaban numerados y se movieron los códigos para que cayeran dentro del polígono al que pertenece.

Además, dicha cartografía se encuentra actualizada a la fecha del proyecto de Grado.

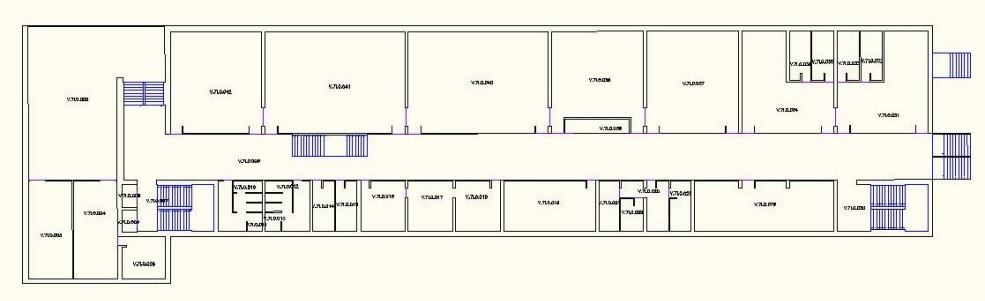


Fig. 3. Cartografía actualizada y generalizada de la Planta Baja

Para finalizar, disponemos del GIS desarrollado en el proyecto final de master, en él contamos con la base de datos, así como los mapas de las distintas plantas.

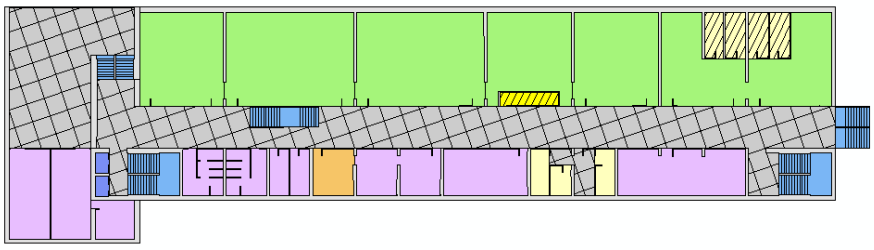
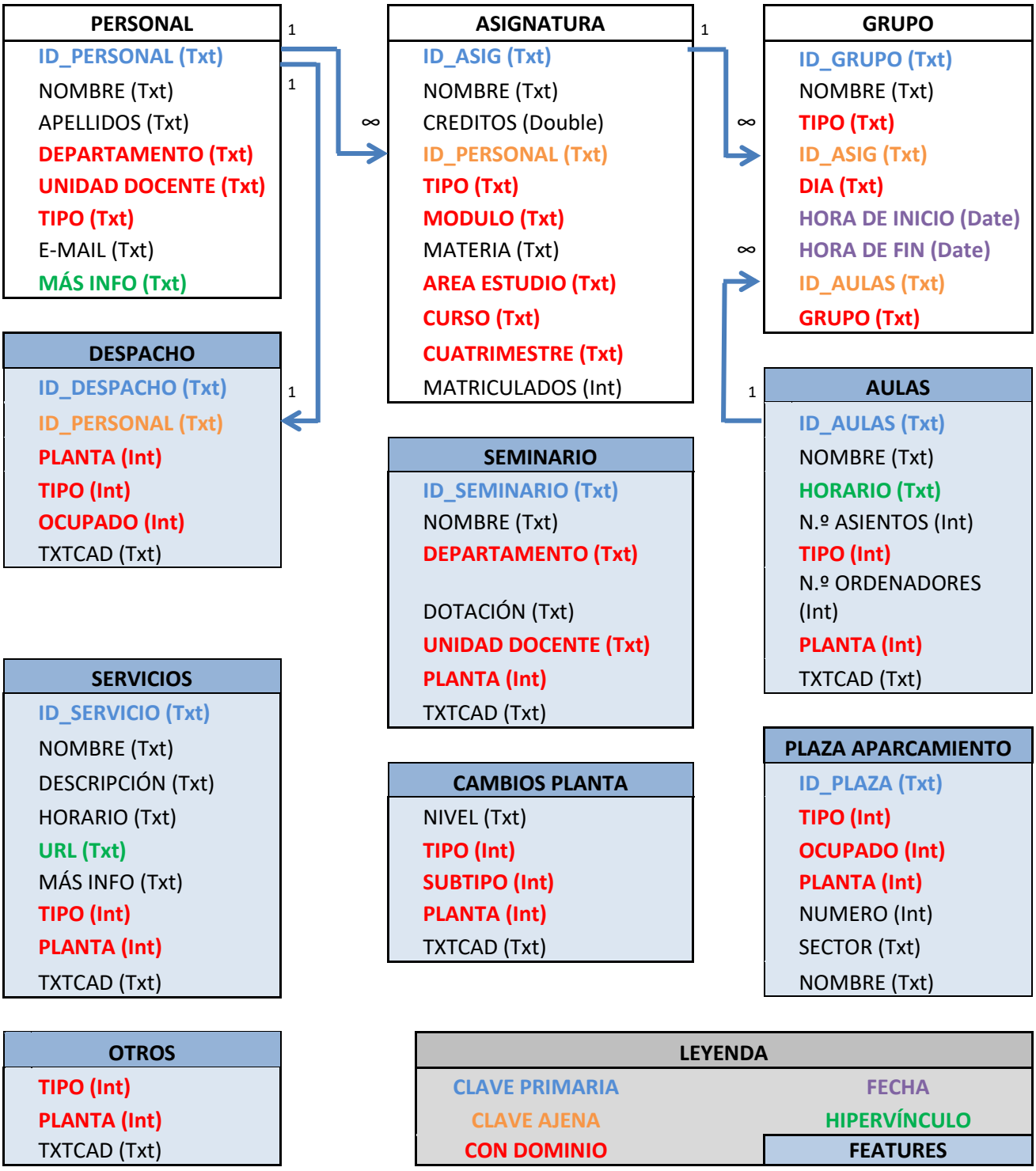


Fig. 4. Planta Baja GIS proyecto final de Grado

MODELO DE DATOS



NOTAS
· FEATURE FUNDACIÓN Y DICGF, IGUAL QUE SERVICIOS.
· TODAS LAS TABLAS TENDRAN LOS CAMPOS PROPIOS DE LA GEODATABASE.

Tabla 1. Modelo de datos de partida

Cabe destacar cual fue el diseño de las claves primarias de las tablas, donde cada una dispone de una lógica diferente, solo mostraremos la información de las tablas que aprovecharemos en este proyecto:

- **Personal:** Empezará con el tipo de personal (PDI o PAS) y seguidamente aparecerá una numeración ascendente, primero el equipo directivo y luego por orden alfabético, según el apellido. Por ejemplo, PDI025 corresponde a Ricardo López Albiñana.

- **Despacho:** Aparecerá la letra D, seguida del código que aparece de cada despacho en la cartografía de los paneles existente en la ETSIGCT.

- **Seminario:** Empezará con las letras SM, seguidas del código que aparece de cada seminario en la cartografía de los paneles existente en la ETSIGCT.

- **Aulas:** Comenzará la letra A, seguida del código que aparece de cada aula en la cartografía de los paneles existente en la ETSIGCT.

- **Servicios:** Aparecerán las letras SV, seguidas del código que aparece de cada servicio en la cartografía de los paneles existente en la ETSIGCT. Debido a que el aparcamiento y la biblioteca no aparecen en los paneles, los servicios encontrados aquí empezarán con las letras SV, seguidas de la planta donde se encuentren (-2 o -1) y seguidas por las dos últimas cifras del código de la cartografía de AutoCAD.

- **Fundación:** Empezará con las letras FUN, seguidas del código que aparece de cada servicio en la cartografía de los paneles existente en la ETSIGCT.

- **DICGF:** Comenzará con las letras DICGF, seguidas del código que aparece de cada servicio en la cartografía de los paneles existente en la ETSIGCT.

La base de datos contiene una serie de campos codificados, aparecen en rojo en el modelo de datos. No todos los emplearemos y necesitaremos, aquí mostramos los campos que necesitaremos del modelo original de datos.

SERVICIOS_TIPO	
0	Otros
1	Aseos o Vestuarios
2	Salas de Reunión
3	Despachos
4	Almacenes
5	Aulas
6	Salas de Personal

Tabla 2. Dominio Servicios_Tipo

DESPACHOS_TIPO	
0	Sin clasificar
1	Despacho de personal docente
2	Despacho de personal técnico
3	Despacho de personal invitado
4	Despacho vacío

Tabla 3. Dominio Despachos_Tipo

AULAS_TIPO	
0	Sin Clasificar
1	Aula no informática
2	Aula Informática

Tabla 4. Dominio Aulas_Tipo

PERSONAL_TIPO	
NC	No Clasificado
PDI	Personal Docente e Investigador
PAS	Personal de Administración y Servicios

Tabla 5. Dominio Unidad docente

DEPARTAMENTO	
NC	No Clasificado
NA	No Aplicable
DEGA	Departamento de Expresión Gráfica Arquitectónica
DIG	Departamento de Ingeniería Gráfica
DFA	Departamento de Física Aplicada
DICGF	Departamento de Ingeniería Cartográfica, Geodesia y Fotogrametría
DIT	Departamento de Ingeniería del Terreno
DMAA	Departamento de Matemática Aplicada
DSIC	Departamento de Sistemas Informáticos y Computación
DU	Departamento de Urbanismo
DLA	Departamento de Lingüística Aplicada
DECS	Departamento de Economía y Ciencias Sociales
DIHMA	Departamento de Hidráulica y Medio Ambiente
DOE	Departamento de Organización de Empresas

Tabla 6. Dominio Departamento

UNIDAD DOCENTE	
NC	No Clasificado
NA	No Aplicable
UDCTG	Unidad docente de Cartografía, Teledetección y Geografía Física
UDF	Unidad docente de Fotogrametría
UDGTGNSS	Unidad docente de Geodesia y Tecnologías GNSS
UDGFA	Unidad docente de Geofísica y Astronomía
UDIT	Unidad docente de Instrumentos Topográficos
UDPCSIG	Unidad docente de Producción Cartográfica y SIG
UDTOP	Unidad docente de Topografía de Obras y Proyectos
UDTEC	Unidad docente de Topografía Escuela de Caminos
UDTA	Unidad docente de Topografía Agroforestal
UDTGEPSG	Unidad docente de Topografía y Geografía Escuela Politécnica Superior de Gandía

Tabla 7. Dominio Unidad docente

METODOLOGÍA

1. *MODELACIÓN 3D DEL EDIFICIO*

1.1. MODELACIÓN BÁSICA DEL EDIFICIO

En este paso se va a generar una modelación básica del edificio, trabajaremos planta a planta y el procedimiento será similar para todas ellas. Partiremos de la cartografía actualizada y generalizada y vamos a emplear el programa SketchUp para la modelación del edificio, empleamos este programa por su facilidad para la generación de modelos 3D orientados a arquitectura. A continuación, veremos el proceso seguido en la planta baja del edificio.

Abrimos SketchUp y creamos una nueva capa desde la herramienta LAYERS, en esta layer es donde trabajaremos con la planta actual. Seguidamente importaremos el .DWG de la planta baja, al importarlo vemos que nos crea varias capas que corresponden a las existentes en el archivo de AutoCAD, seleccionamos todas esas layers y las borramos con la opción de mantener las geometrías en la capa activa, la planta activa.

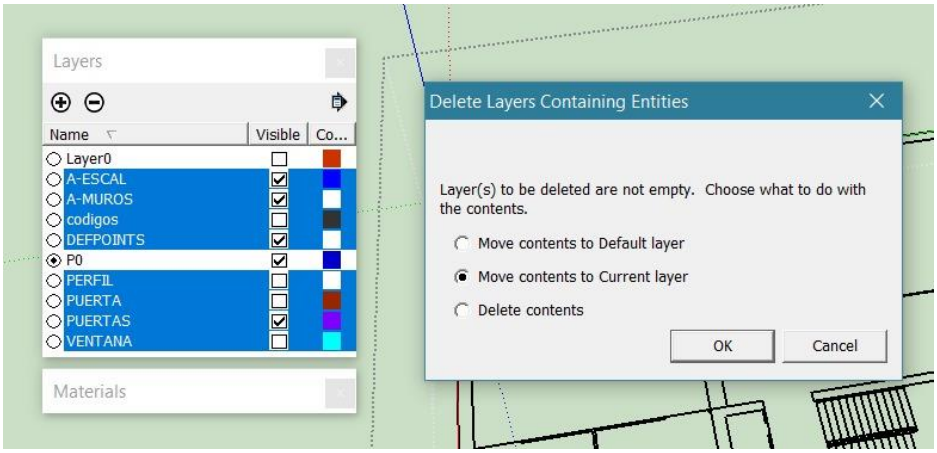


Fig. 5. Borrado de capas de AutoCAD

Por defecto nos lo importa como un componente, para trabajar más cómodamente lo extraemos mediante el EXPLODE y seguidamente cerramos todos los polígonos, asegurando que las caras queden por cara principal, cara blanca en el programa.

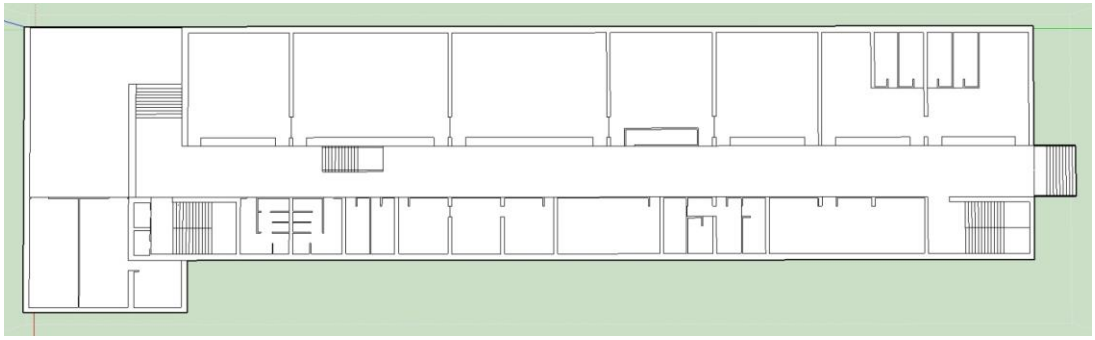


Fig. 6. Polígonos de la planta baja en SketchUp

Ahora vamos a empezar a levantar los muros, nuestro sistema local de cotas partirá de que las zonas que se encuentran al nivel terrestre tendrán la cota 0, además se considerará que el terreno es completamente plano. Para la obtención de las otras cotas emplearemos los escalones que nos venían numerados en el AutoCAD. Cada escalón mide 17 centímetros, este dato se ha obtenido de la medición en campo mediante una cinta métrica. Sabiendo esto, empezaremos a posicionar los distintos escalones a su altura correspondiente, para ello emplearemos la herramienta PUSH/PULL.

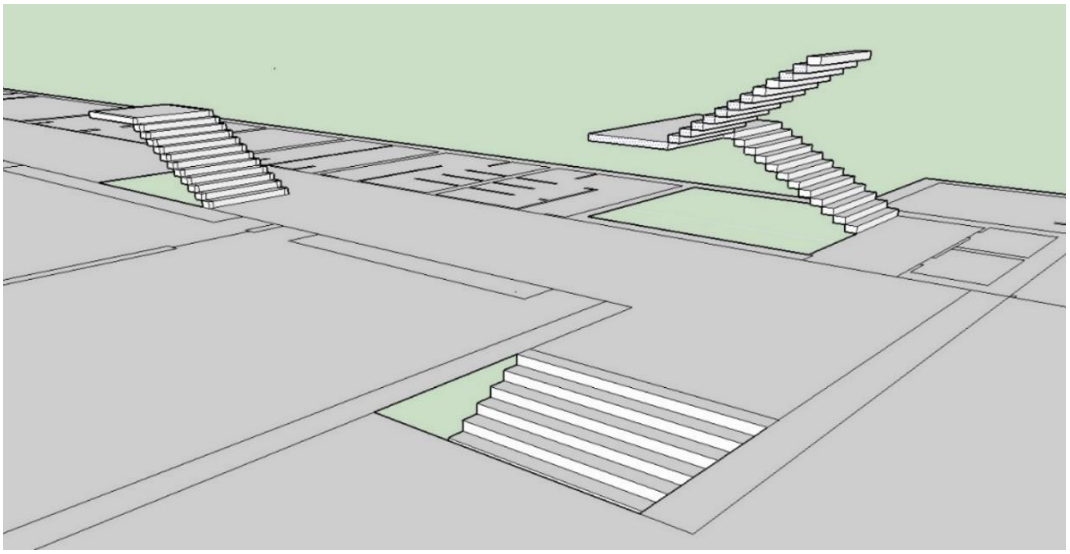


Fig. 7. Elevando escaleras de la planta baja

Una vez posicionado correctamente los escalones, pasaremos a arreglar los muros, llegando al techo con una distancia de 2.77m, este dato se ha obtenido de la medición en campo, mediante una cinta métrica.

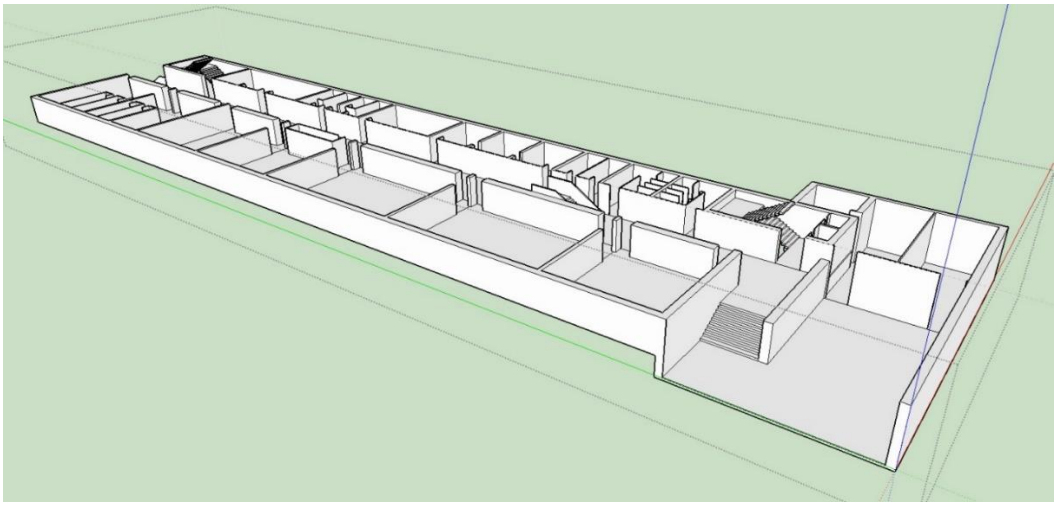


Fig. 8. Elevando muros de la planta baja

Realizaremos el mismo proceso para todas las plantas del edificio, así como para el aparcamiento del mismo. Con este procedimiento, obtendremos un primer modelo base, sobre el que iremos refinando hasta obtener el modelo final del edificio.

1.2. REFINANDO EL MODELO BÁSICO

Debido a que hemos partido de los mapas generalizados, el modelo no dispone de gran detalle y fidelidad, por ello vamos a superponer el mapa no generalizado, consiguiendo así detectar las zonas donde hay vidrios, muros que son cristaleras etc.

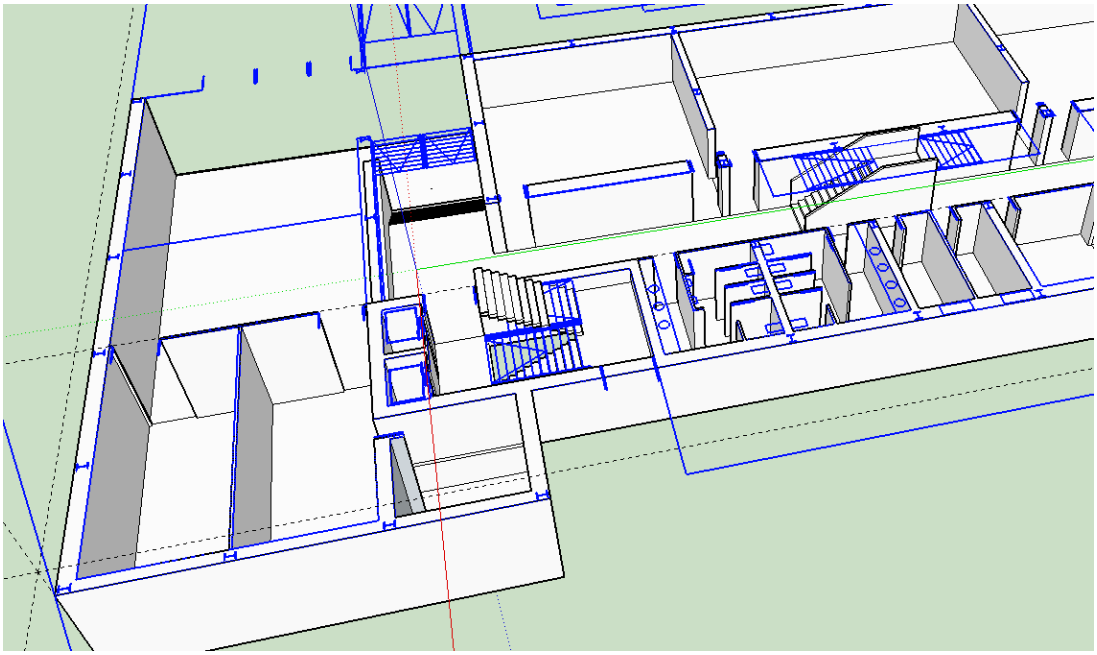


Fig. 9. Superposición del mapa original en el modelado

Empleando los datos de los mapas originales y datos de apoyo tomados en campo, se perfila el modelo, dejando los elementos complejos para realizarlos a posteriori. Para realizarlo, se emplean las herramientas de dibujo existentes, así como herramientas de apoyo como la creación de guías y medición sobre el modelo.

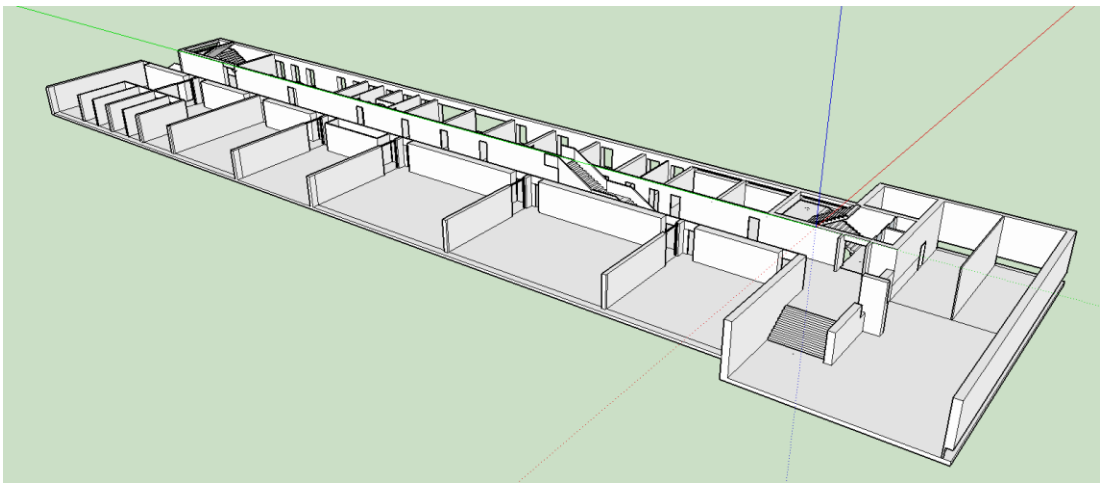


Fig. 10. Modelado refinado de la planta baja

Se puede observar cómo se han arreglado puertas, ventanas, muros y escaleras.

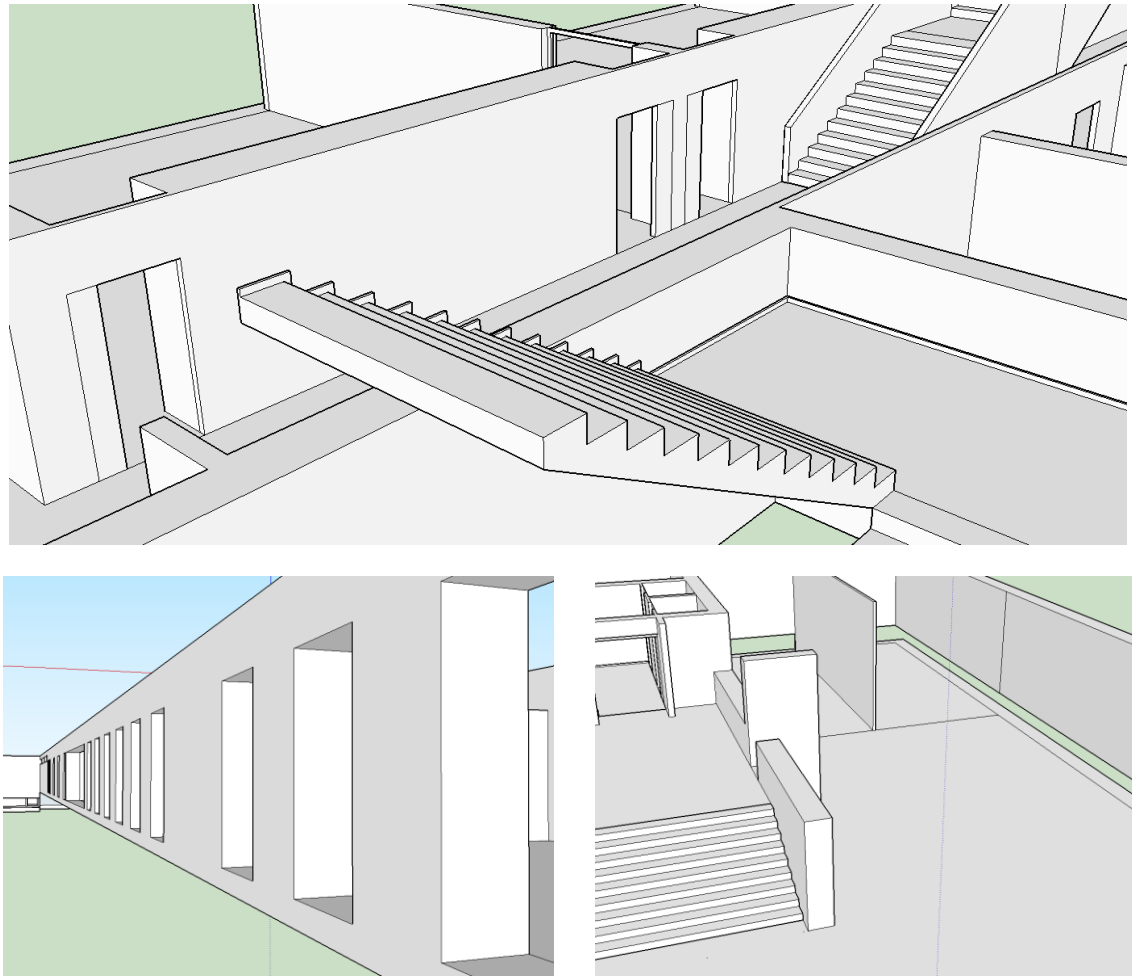


Fig. 11. Detalle del modelado refinado de la planta baja

Se debe destacar, que todos los datos de apoyo tomados en campo, han sido mediciones relativas a puntos existentes en el modelo, realizadas mediante cinta métrica, la cual nos indica que no tendremos una precisión excesiva en el modelo, no obstante, debido que el objeto del proyecto no necesita una gran precisión, es asumible el uso de esta metodología.

Realizaremos el mismo procedimiento en las distintas plantas y el aparcamiento. Tras este paso, ya disponemos de un modelo algo más fiel a la realidad. Sin embargo, aún debemos texturizarlo y añadir los objetos complejos al mismo, obteniendo así el modelo final del edificio.

1.3. TEXTURIZANDO EL MODELO

Antes de realizar los objetos más complejos (ventanas, barandillas, pilares, cristaleras, puertas…), se va a pasar a la texturización del modelo. Para ello, empleamos las herramientas que nos proporciona SketchUp.

La herramienta permite emplear un color único, jugar con la transparecia y emplear imágenes. En nuestro modelo dispondremos tanto de texturas de un único color, color con trasparecía y texturas con imágenes.

Veamos la creación de un material simple, accedemos a la sección de WINDOW / MATERIALS, allí hacemos clic en CREATE MATERIAL. Desde esta ventana creamos el material simple, no se le realiza mucha configuración, simplemente se le indica un nombre y se configura el color.

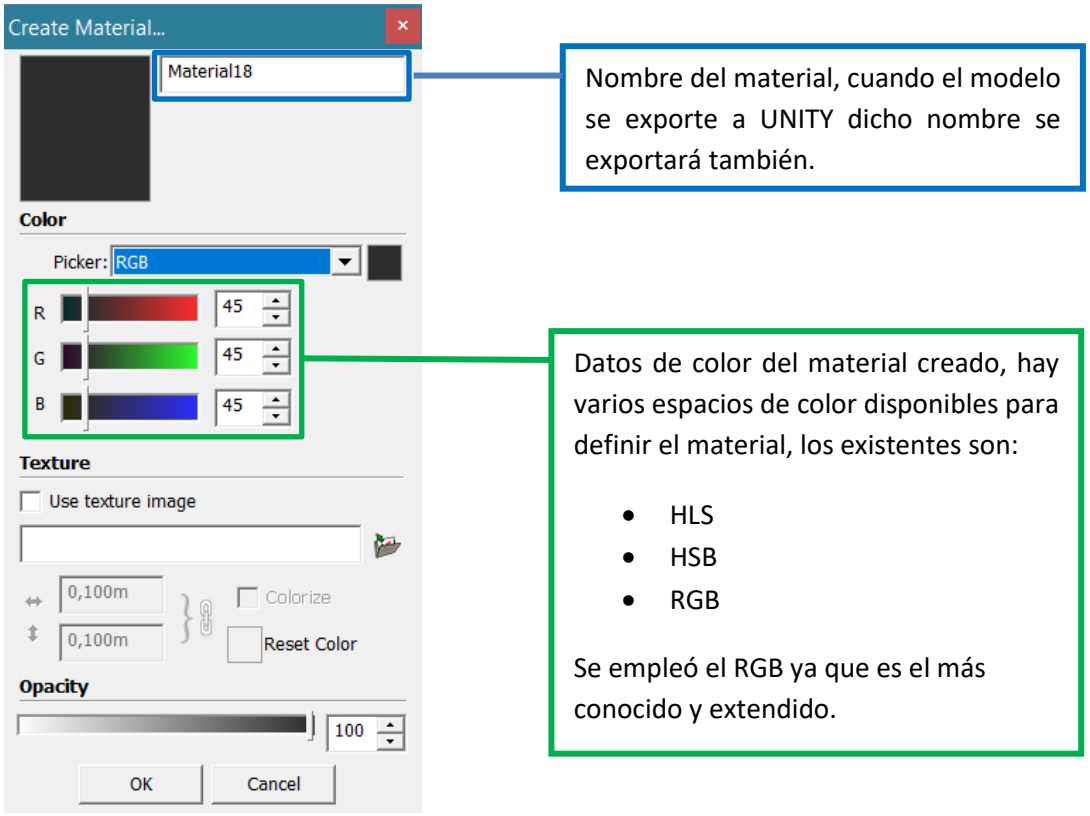


Fig. 12. Creando un material simple

Una vez terminado el material, pasamos aplicarlo a una cara del modelo mediante la herramienta de PAINT BUCKET.

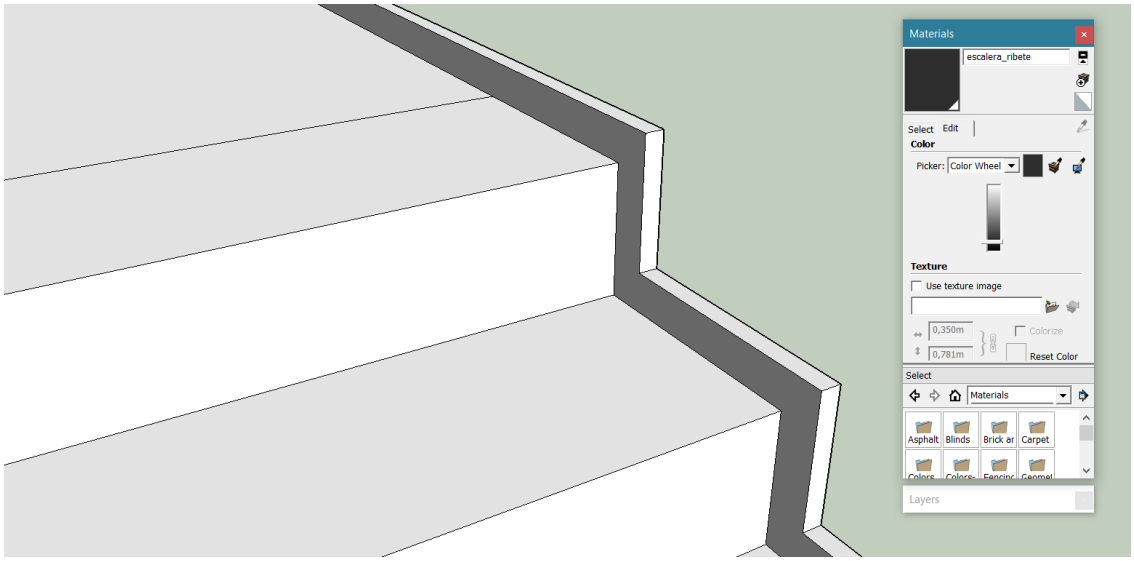


Fig. 13. Aplicando un material simple

Veamos ahora la creación de un material transparente, es muy similar al simple, pero en este caso definimos la opacidad del material, veámoslo en un ejemplo.

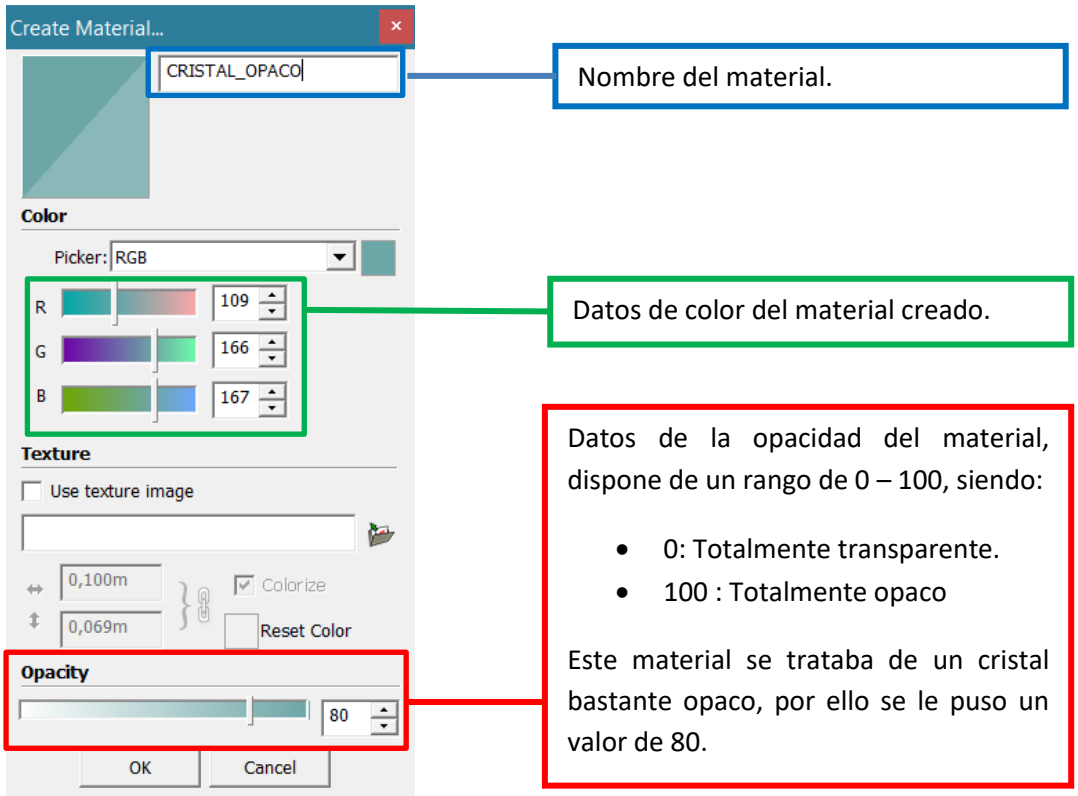


Fig. 14. Creando un material transparente

De la misma forma que se emplea un material simple podemos aplicar un material transparente. Se observa que al añadirlo permite visualizar elementos posteriores del plano texturizado, en mayor o menor medida según la opacidad del material.

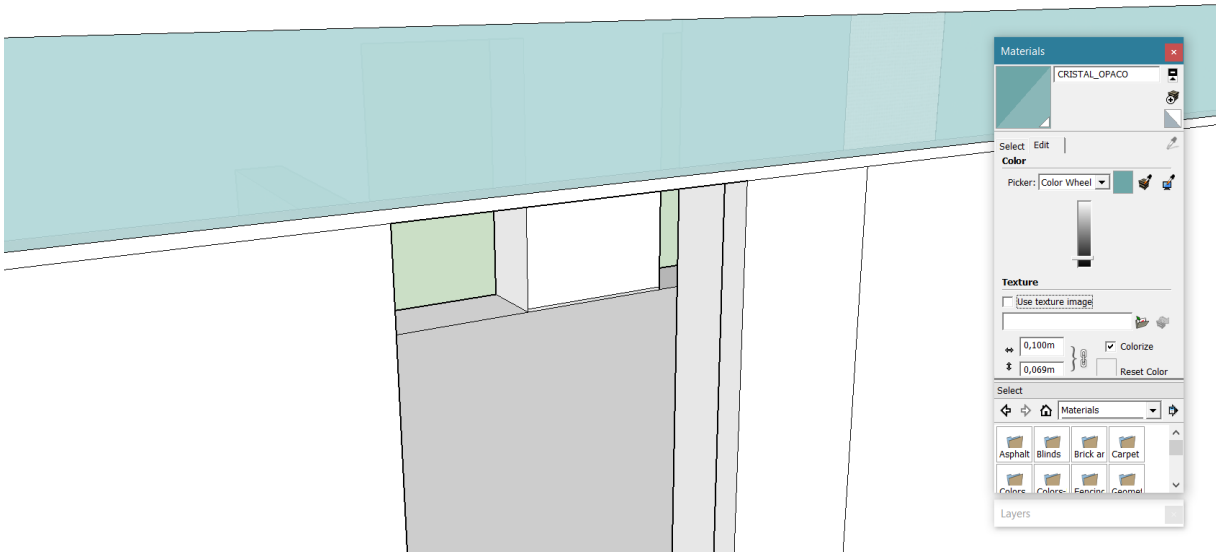


Fig. 15. Aplicando un material transparente

Veamos ahora la creación de un material con una imagen de textura, es similar a la creación de los anteriores materiales, sin embargo, aquí definimos una imagen y especificamos las dimensiones de la textura, veámoslo en un ejemplo.

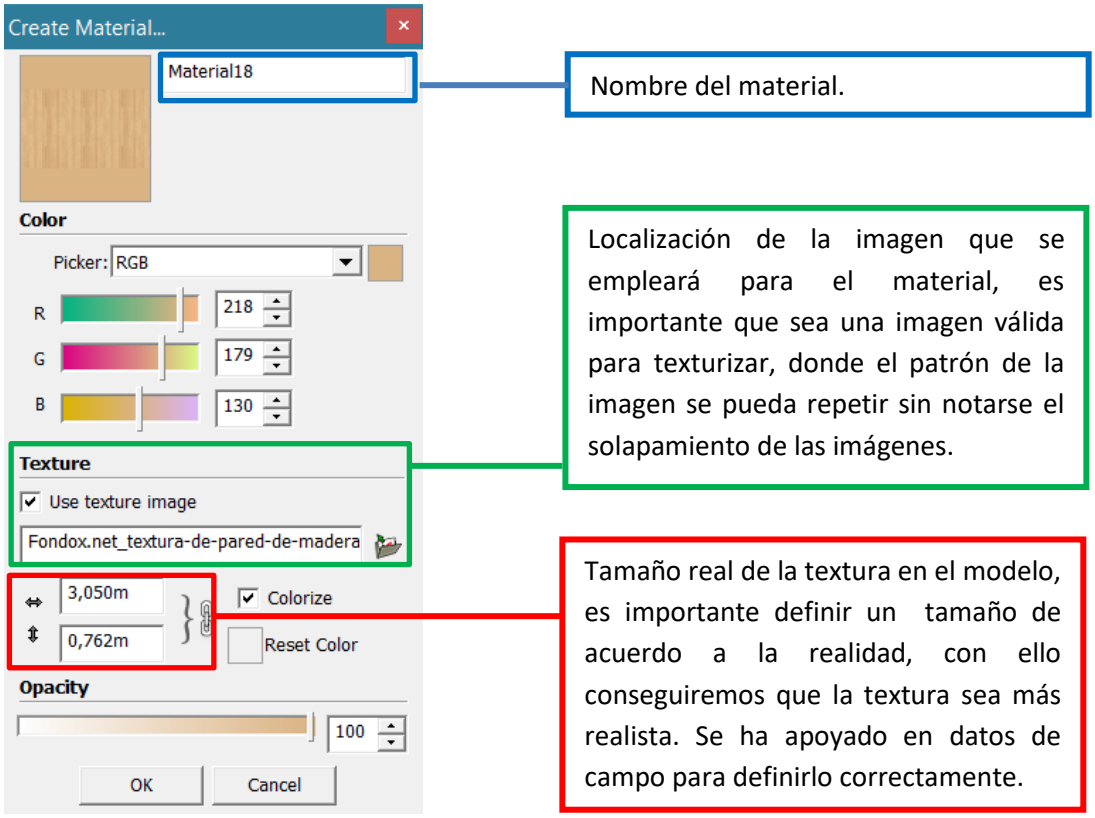


Fig. 16. Creando un material con imagen de textura

Mediante el PAINT BUCKET se puede añadir el material, tal y como se realiza con los simples y con los transparentes.

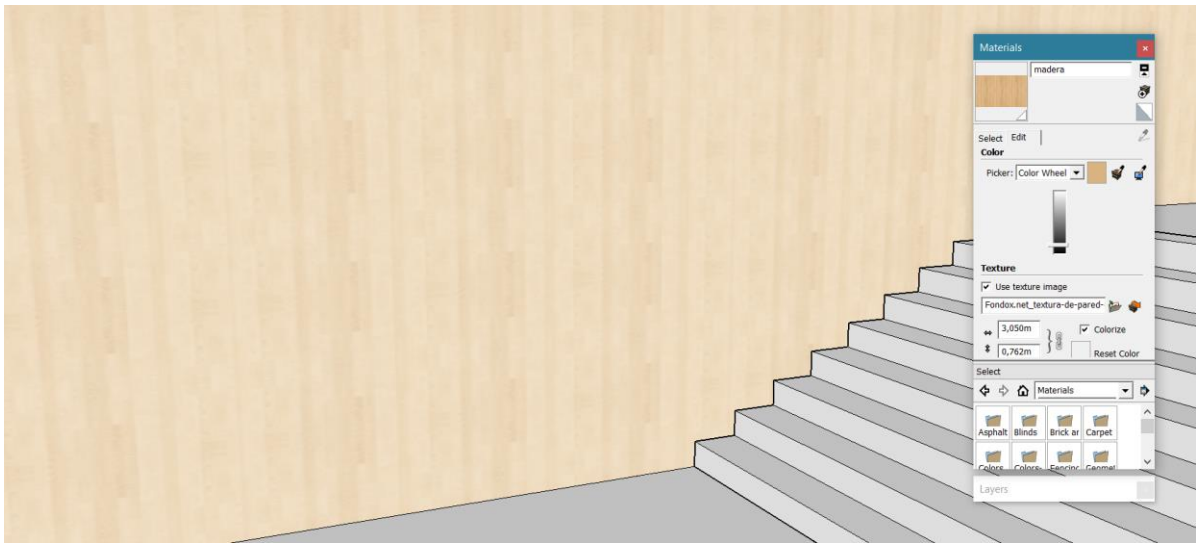


Fig. 17. Aplicando un material con imagen de textura

Cabe destacar que los materiales con imagen de textura se pueden modificar en posición, rotación, escala y proyección, pudiendo así posicionar la textura correctamente en cada plano del modelo. Para ello hacemos clic derecho sobre el plano y accedemos a TEXTURE / POSITION, desde ahí podemos mover, rotar, escalar y deformar la imagen de la textura a nuestro gusto.

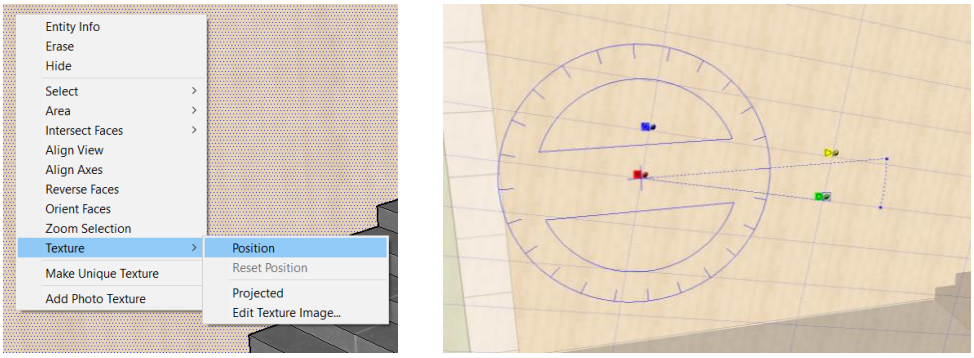


Fig. 18. Modificando un material con imagen de textura

Una particularidad que permiten los materiales con imágenes de texturas, es que guardan el espacio RGBA de la imagen .PNG, permitiendo así que la imagen tenga zonas transparentes y zonas opacas, con ello podemos conseguir mediante texturación ahorrarnos geometrías.

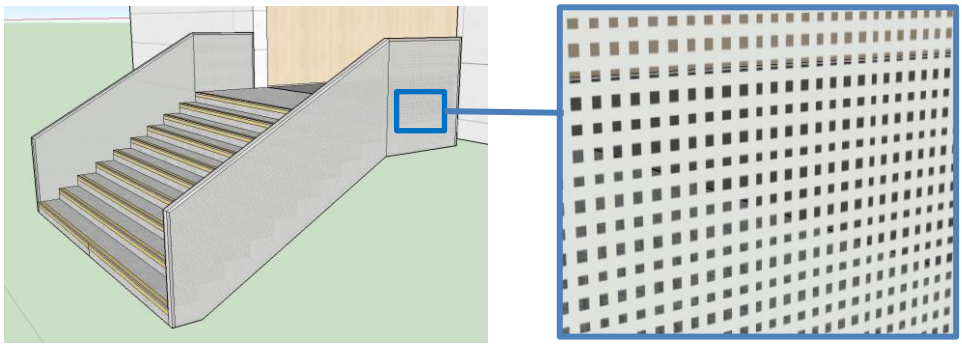


Fig. 19. Ejemplo de material con imagen de textura que permite transparencia

Siguiendo estos pasos, se van realizando los distintos materiales y texturizando el modelo. Para la aplicación correcta de los distintos materiales, se ha tenido que realizar nuevas geometrías para poder aplicar los materiales en las distintas zonas. Para poder disponer de la información de dimensiones y posiciones de las texturas, se ha tenido que realizar mediciones en campo, mediante cinta métrica.

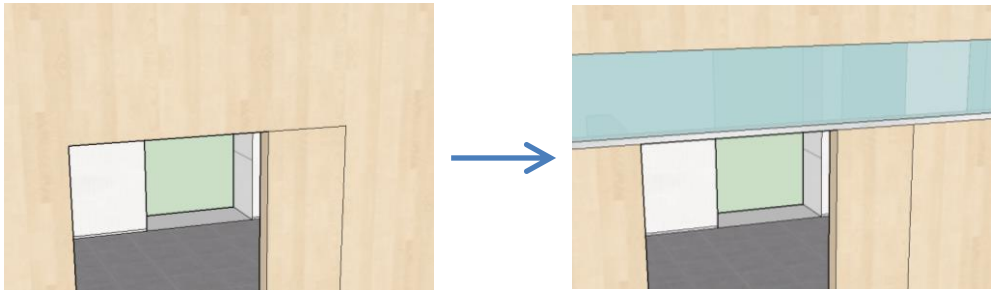


Fig. 20. Modificando el modelo para poder texturizar

Las imágenes de textura, se han obtenido tanto de webs gratuitas de texturas, creadas desde 0 y desde fotografías tomadas en campo. Los dos últimos tipos terminadas en programas de edición de imágenes para que puedan ser válidas para texturizar. Donde se ha aplicado la técnica básica de texturización de imágenes, este proceso consiste en partiendo de una imagen, se crean imágenes iguales, pero con orientaciones espejo entre ellas, de tal forma que el inicio y el final de la misma sea igual, tanto en horizontal como en vertical, con ello conseguimos que la textura se note homogénea en toda la superficie del modelo.

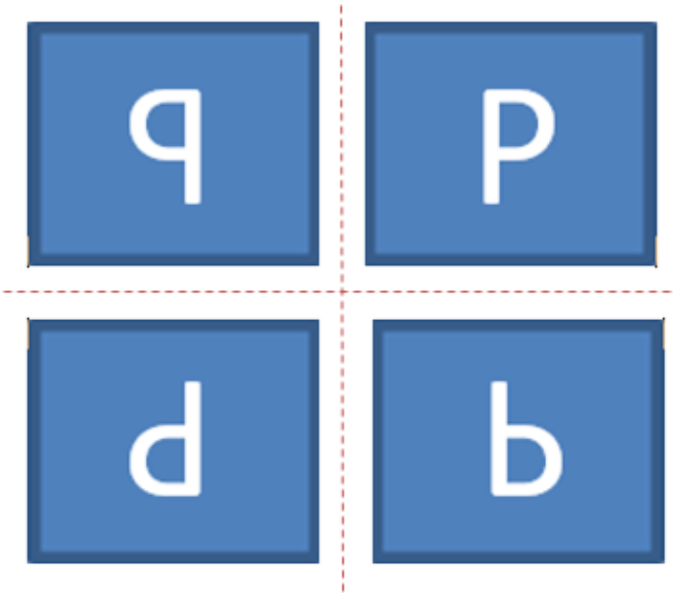


Fig. 21. Proceso de texturización de imágenes

Siguiendo todos estos procedimientos, conseguimos finalmente texturizar el modelo, obteniendo un modelo más similar a la realidad. Realizamos este mismo procedimiento para las distintas plantas del edificio y el aparcamiento.

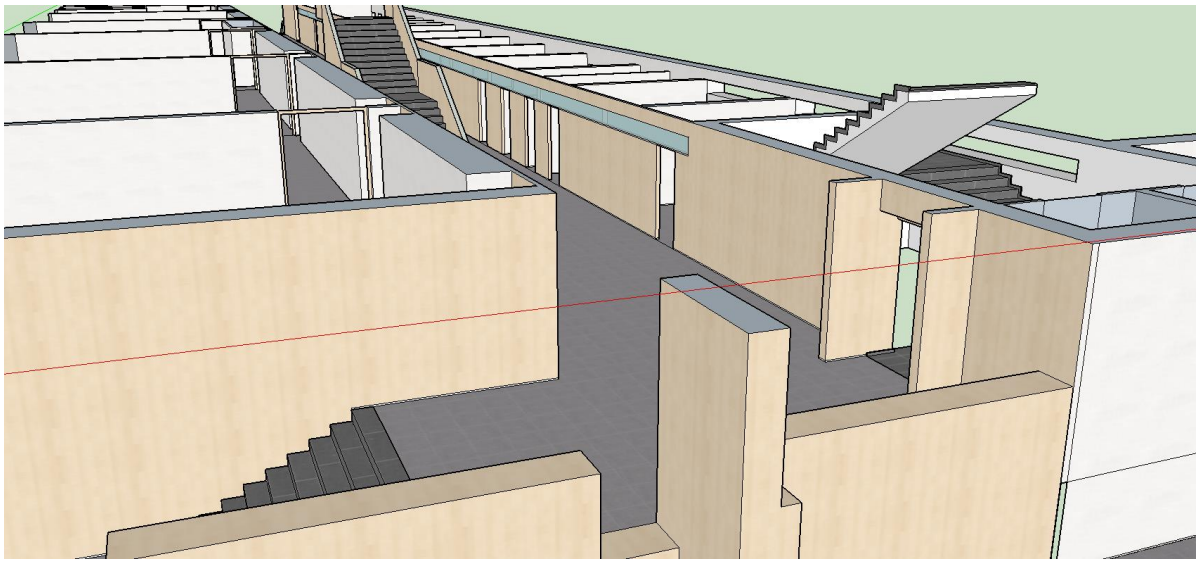


Fig. 22. Planta baja texturizada

1.4. FINALIZACIÓN DEL MODELO – OBJETOS MÁS COMPLEJOS

Para finalizar el modelo del edificio, pasaremos a realizar los objetos complejos de la estructura del edificio, siendo estas ventanas, puertas, cristaleras, barandillas... Para ello, se procede a ir uno a uno, creando en cada uno de ellos un grupo separado, con ello conseguiremos que al importarlo en UNITY se mantengan en subgrupos. Para la realización de estos objetos, se emplean datos obtenidos en campo mediante una cinta métrica. A continuación, se observan varios ejemplos de elementos complejos creados:

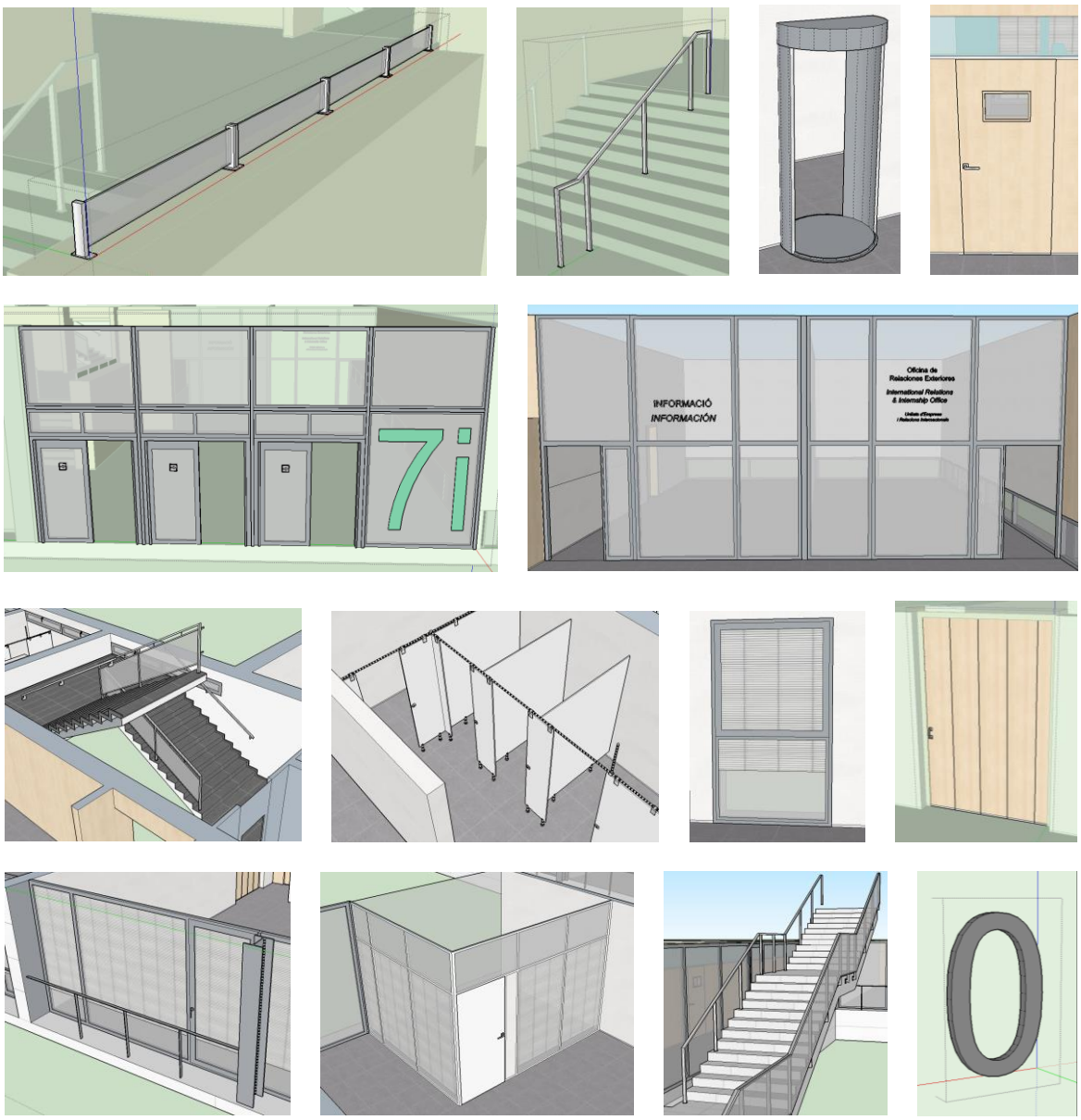


Fig. 23. Ejemplos de objetos complejos de la estructura del edificio

Se debe comentar una peculiaridad de algunos materiales generados, estos se tratan de materiales con imagen de textura, pero con la particularidad de que el propósito es definir toda la zona a texturizar, no se requiere que se repitan en la zona, veamos unos ejemplos:

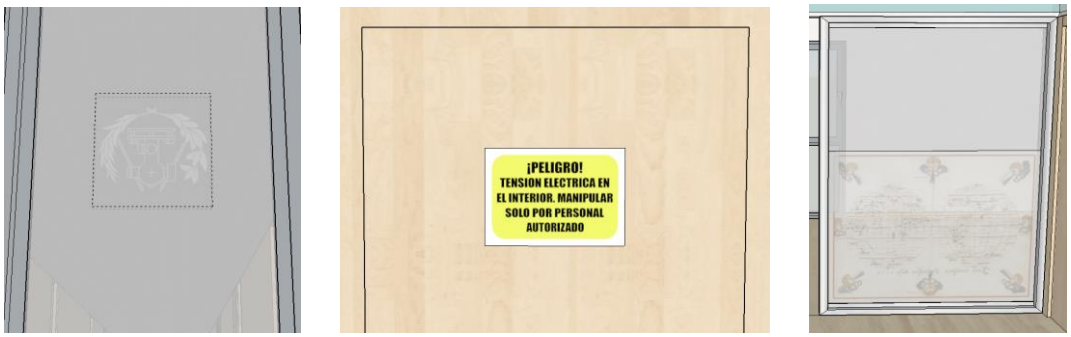


Fig. 24. Ejemplos de materiales con imágenes de textura sin repetición de la imagen

Para que el modelado no sea muy pesado y ahorrar así tiempo y recursos, se ha definido sólo algunas habitaciones donde se podrá acceder, con esto limitamos tiempos de modelado de interiores de algunas salas y menos objetos en el modelo final, mejorando así el futuro rendimiento de la aplicación generada en Unity, para que haya diversidad, se han seleccionado distintas habitaciones, repartidas en las distintas plantas. Veamos a continuación que zonas se han elegido.

- Todas las zonas de paso a otras habitaciones del edificio, ya que son necesarias para poder acceder a futura información de las salas.
- Todos los aseos de las distintas plantas del edificio.
- **PLANTA BAJA**
  - A042 – 0.1 Gauss.
  - A040 – 0.3 Marqués de la ensenada.
  - A034 – LAB L0.1
  - A031 – LAB L0.2
  - SV004 – Información.
  - SV003 - Unidad de prácticas en empresa y relaciones internacionales.
  - SV025 – Profesor Francisco Córdoba Parra.
  - SV017 – Delegación de alumnos.
- **PRIMERA PLANTA**
  - A137 – L1.1 Laboratorio de Física Aplicada.
  - SV105 – Secretaría dirección.
  - SV133 – 1.4 Pirineos.
  - SV151 – 1.1 Proyectos.
  - SV140 – Secretaría.
- **SEGUNDA PLANTA**
  - A232 – Laboratorio de Fotogrametría Aplicada.
  - SV201 – Salón de actos.
  - SV234 – Sala de revelado.
- **TERCERA PLANTA**
  - A340 – CGF-1.
  - A338 – CGF-2.
  - D325 – Despacho de Jesús Manuel Palomar Vázquez.
- **CUARTA PLANTA**
  - A430 – 4.1 Justo Nieto.
  - D410 – Despacho de Santiago Guillem Picó.
  - FUN443 – Dirección – Reprografía.

En estas habitaciones se ha añadido más detalles estructurales, como tarimas y ventanas. Además, se ha realizado la texturación de los techos de estas habitaciones.

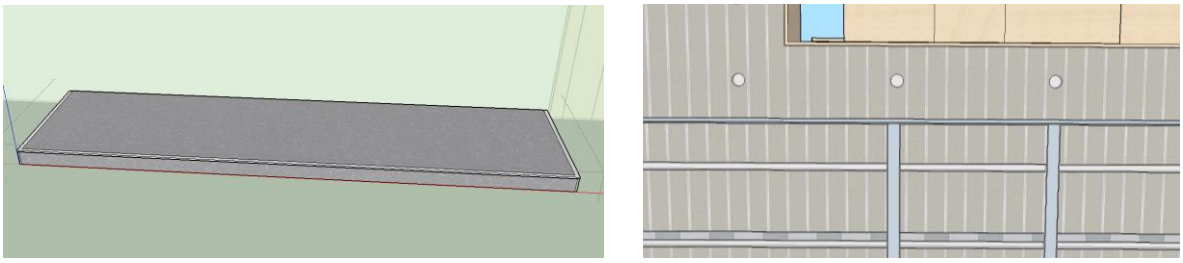


Fig. 25. Detalles estructurales de las habitaciones accesibles

Cabe mencionar que el techo de la cuarta planta es bastante diferente, este se ha tratado como una planta extra, a nivel de layers, modelándose y texturizándose gracias a datos de campo y mediciones tomadas desde ortofotos, ya que existían puntos donde no se podía acceder para medir con la cinta métrica, debido a la naturaleza del proyecto, el cual no está destinado a tener una precisión muy refinada, es asumible emplear estas mediciones.

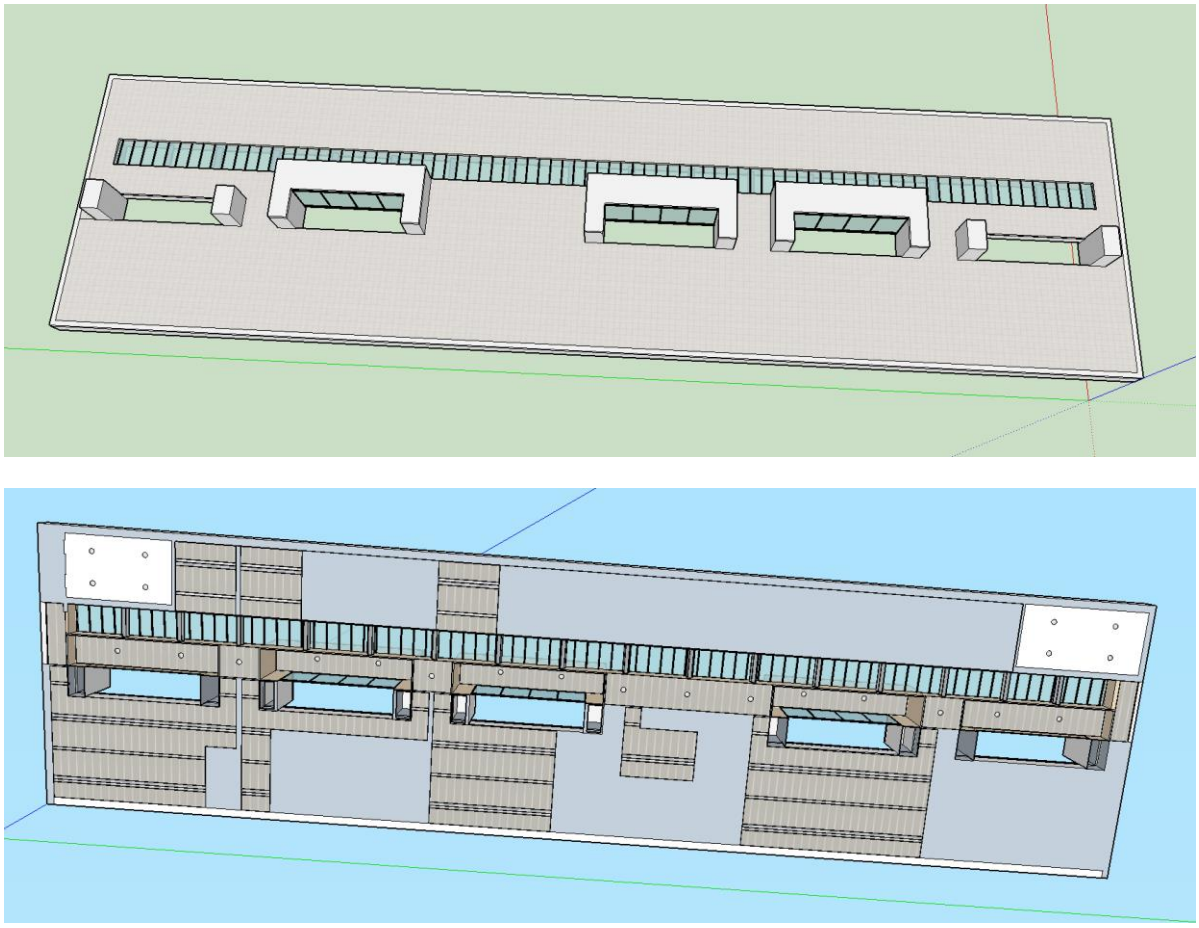


Fig. 26. Techo de la cuarta planta del edificio

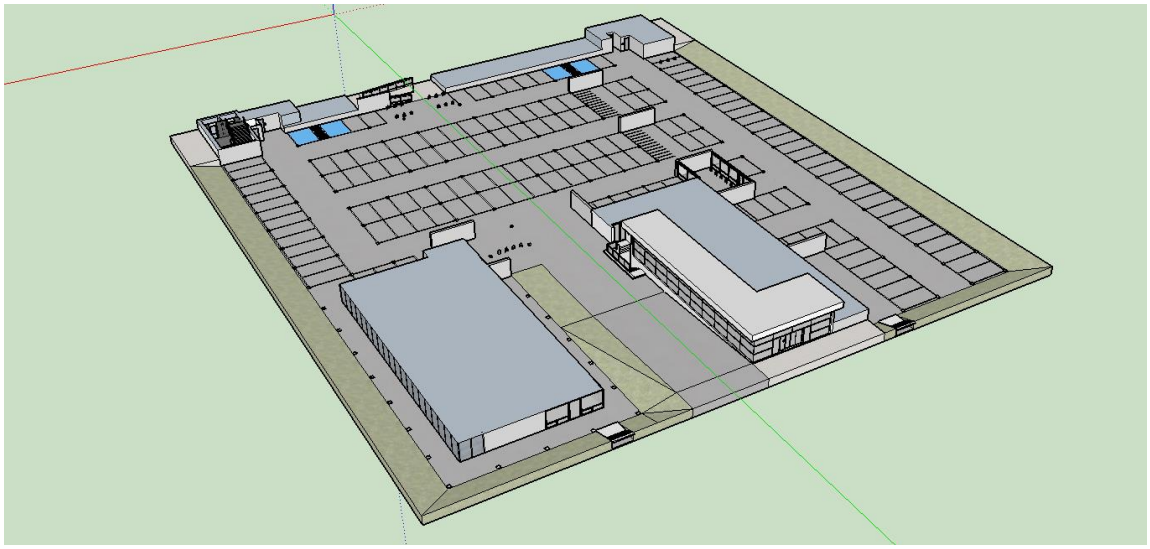


Fig. 27. Modelado final del aparcamiento

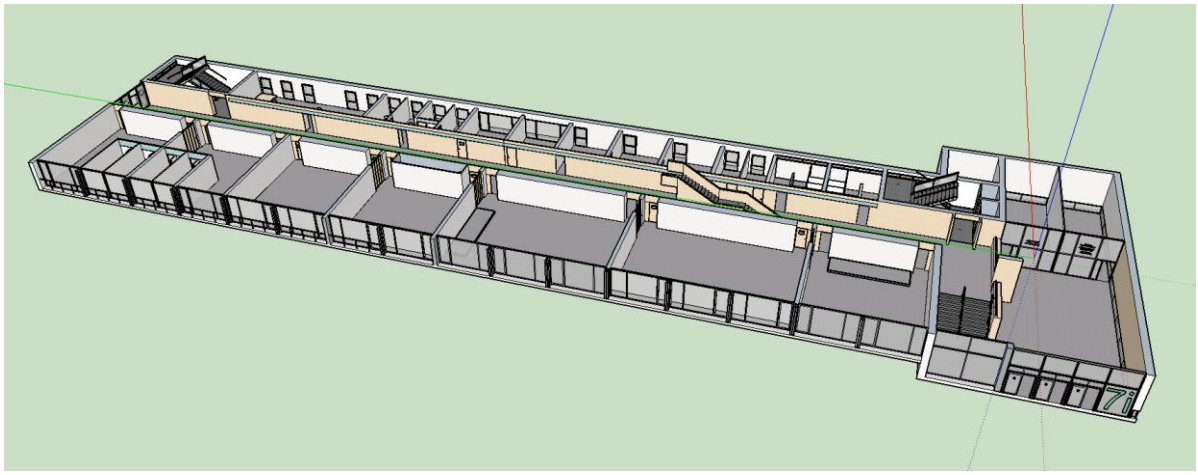


Fig. 28. Modelado final de la planta baja

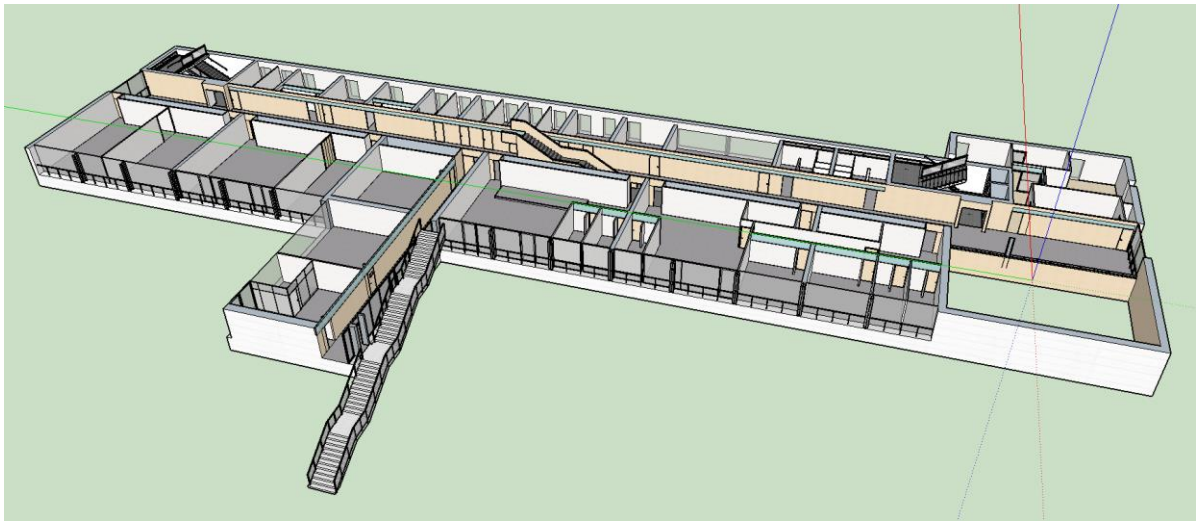


Fig. 29. Modelado final de la primera planta

Fig. 30. Modelado final de la segunda planta

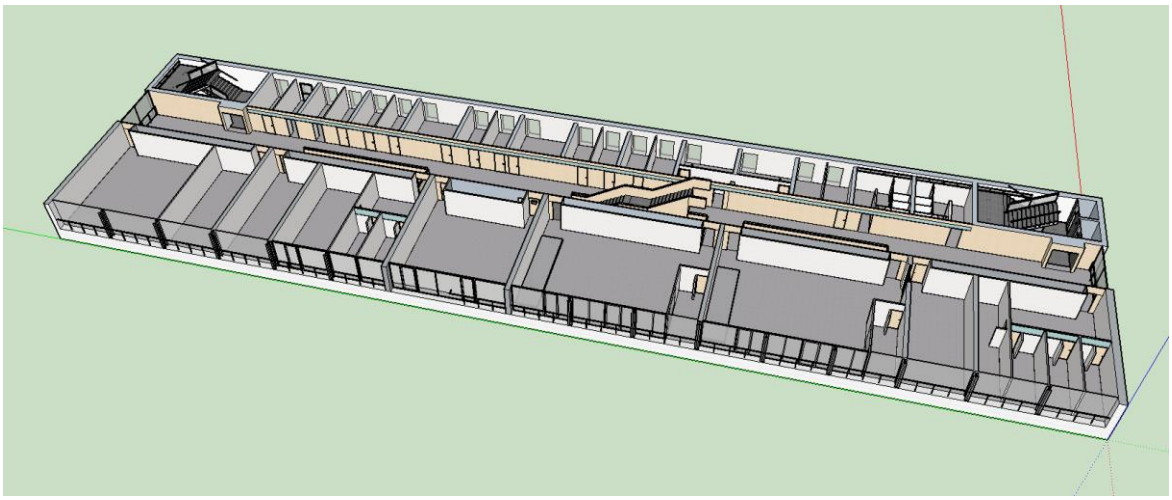


Fig. 31. Modelado final de la tercera planta

Fig. 32. Modelado final de la cuarta planta

Como podemos ver en las imágenes resultantes del procedimiento, se han eliminado todas aquellas puertas pertenecientes a habitaciones que van a ser accesibles, esto es debido a que posteriormente se añadirán como objetos independientes en UNITY, para así poderle dar funcionalidad a las mismas. Se puede observar que el modelado del aparcamiento, no aparecen los pilares, esto es debido a que se añadirán a posteriori en UNITY, siendo este un objeto que se duplica, esto ahorra tiempo de computación en la carga superficies en UNITY.

1.5. AÑADIENDO EL CONTEXTO AL MODELO

Tras estos pasos, ya disponemos de un modelo del edificio bastante fiel a la realidad. Sin embargo, vemos que realmente el modelo no se ve como un conjunto completo, debido a que sólo está representado la parte del edificio 7I.

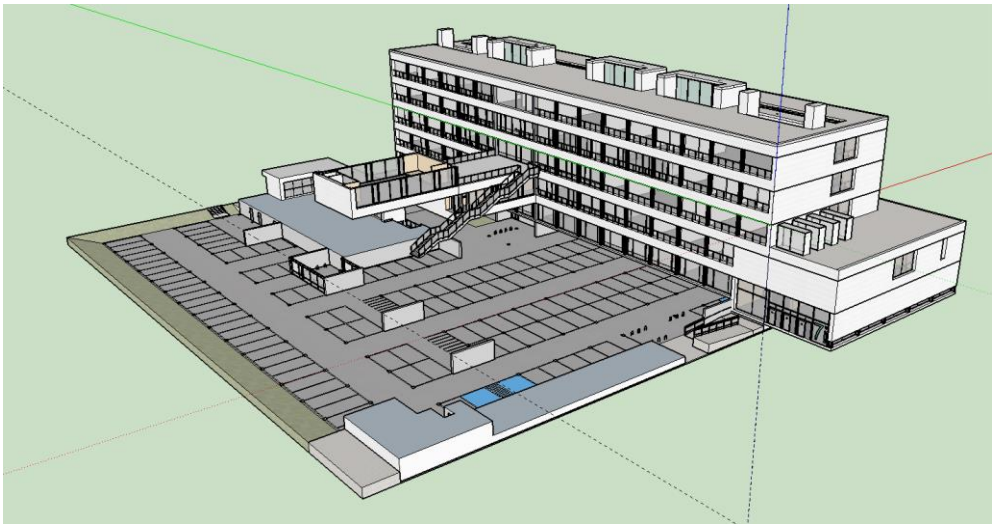


Fig. 33. Modelado final del edificio sin la parte del edificio 7J

En este apartado vamos a terminar el modelo añadiendo la parte correspondiente al edificio 7J. Para realizar el modelado y texturización se han empleado tanto los mapas detallados, mediciones en campo y mediciones desde ortofotografías. Se ha de recalcar que se ha realizado un modelado bastante simple, donde realmente se ha modelado sólo la fachada del edificio, intentando ser lo más fiel posible, pero sin invertir mucho tiempo y desarrollo en el mismo, ya que este no es el objeto del proyecto.

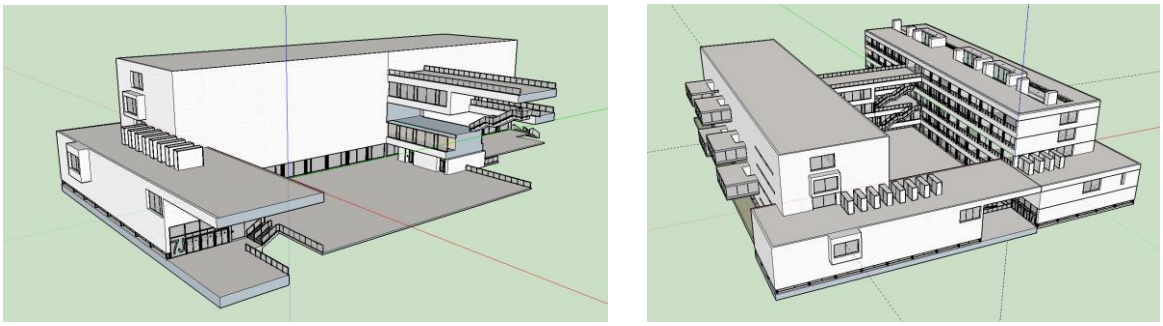


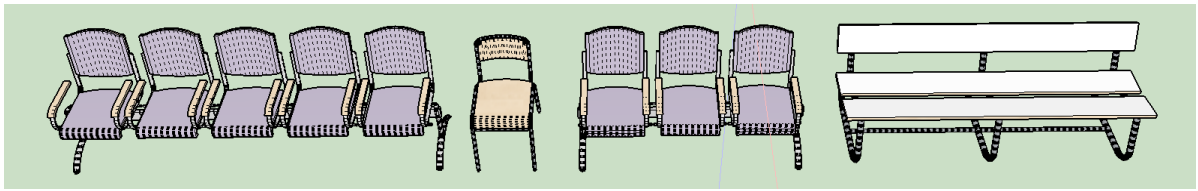
Fig. 34. Modelado final del edificio

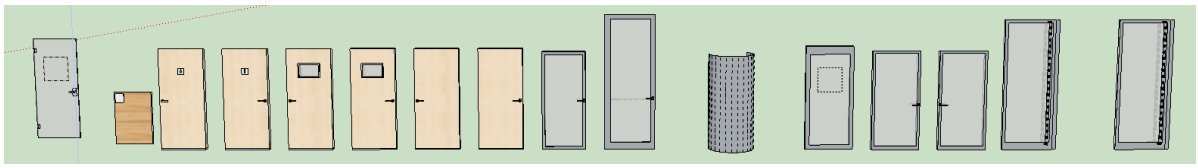
2. *MODELACIÓN 3D DE OBJETOS DEL EDIFICIO*

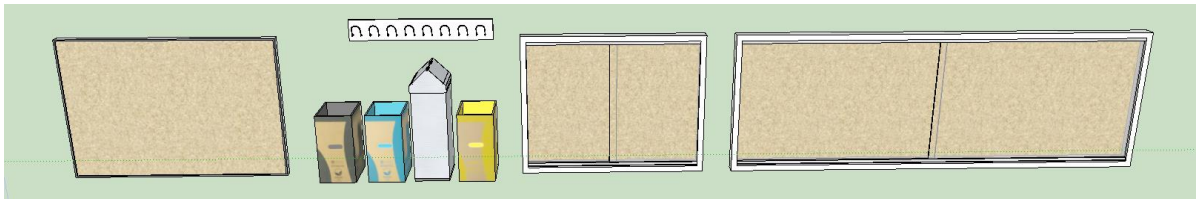
2.1. CREACIÓN 3D DE ELEMENTOS EXISTENTES

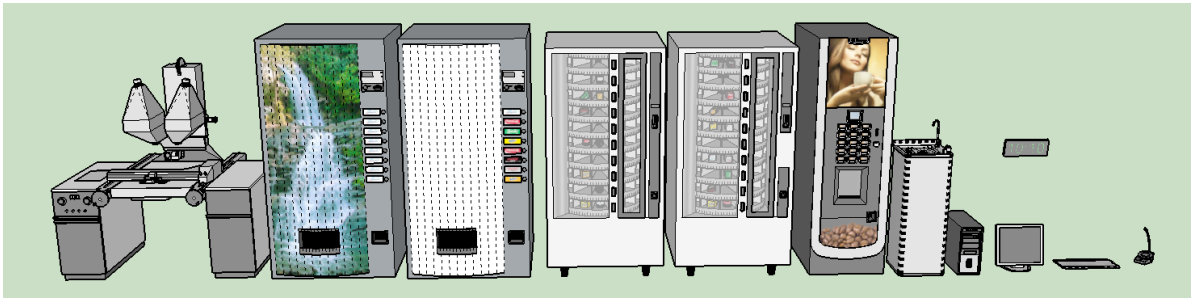
Una vez ya tenemos el modelado del edificio, el siguiente paso es la creación de los distintos objetos existentes dentro del edificio, como pueden ser sillas, armarios, mesas, estanterías, bancos, pizarras, elementos electrónicos etc... Para la realización de estos, se han empleado todas las herramientas de modelado y texturización que nos brinda SketchUp, las cuales ya han sido empleadas para la creación del modelo del edificio.

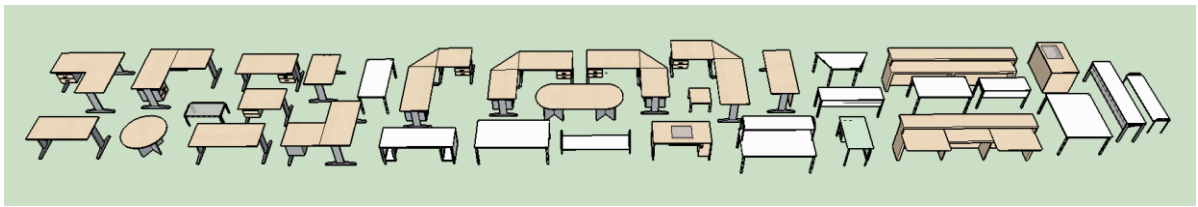
Para la realización de estos modelados, nos apoyamos de datos tomados en campo: medidas tomadas con cinta métrica y fotografías de los objetos. Se pretende que no sean objetos muy complejos, es por ello que realizaremos el modelado manualmente, ya que si empleamos procesos de fotogrametría obtendríamos modelos más realistas, pero con mayor número de aristas, siendo estos modelos más complejos de gestionar por el motor de videojuegos.











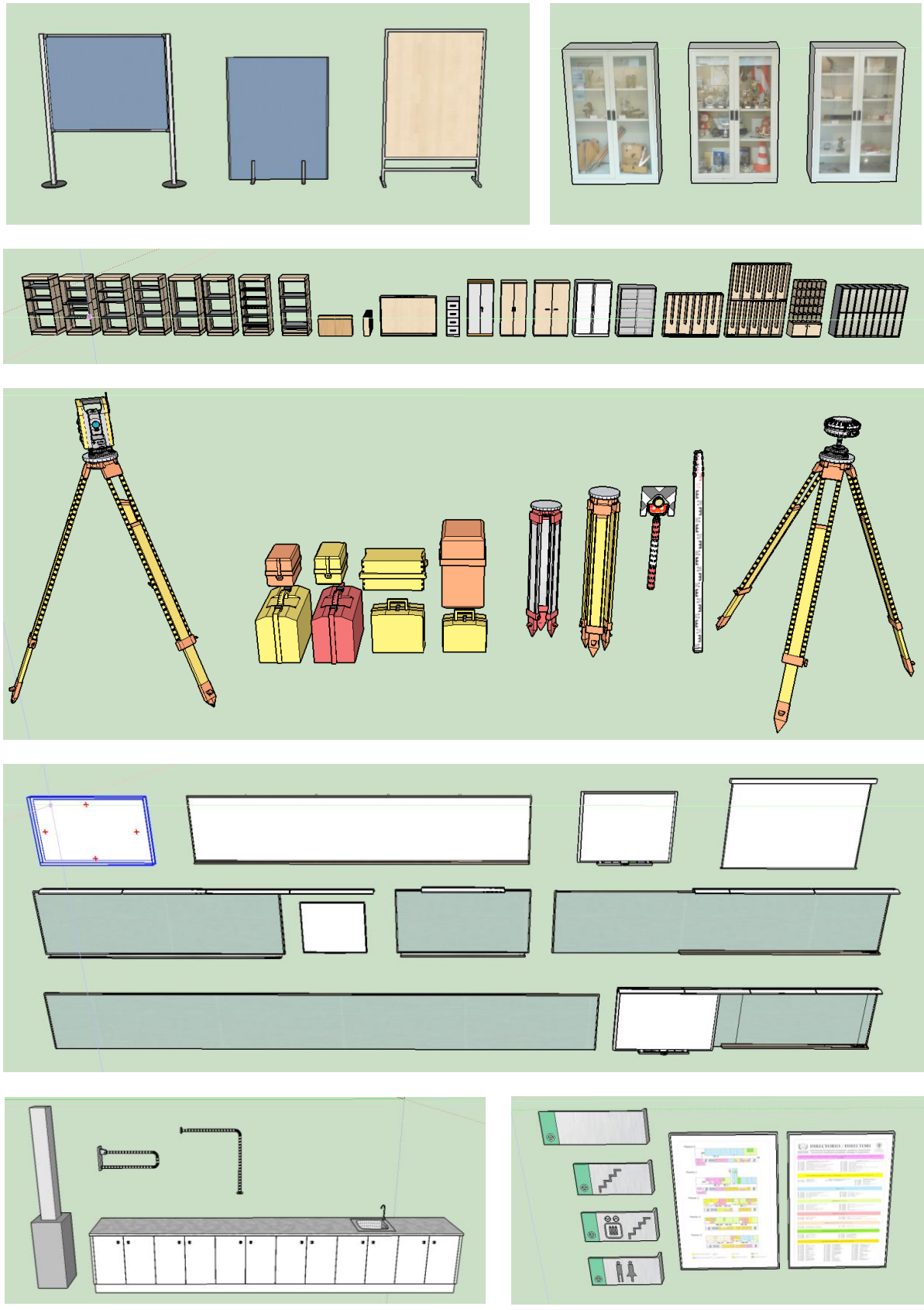


Fig. 35. Modelados de objetos del edificio

Cabe destacar el modelado de los ascensores, ya que se han realizado por fuera, por dentro y las puertas por separado. Esto es debido a que en UNITY se pretende que sean accesibles y tengan lógica de movimiento.

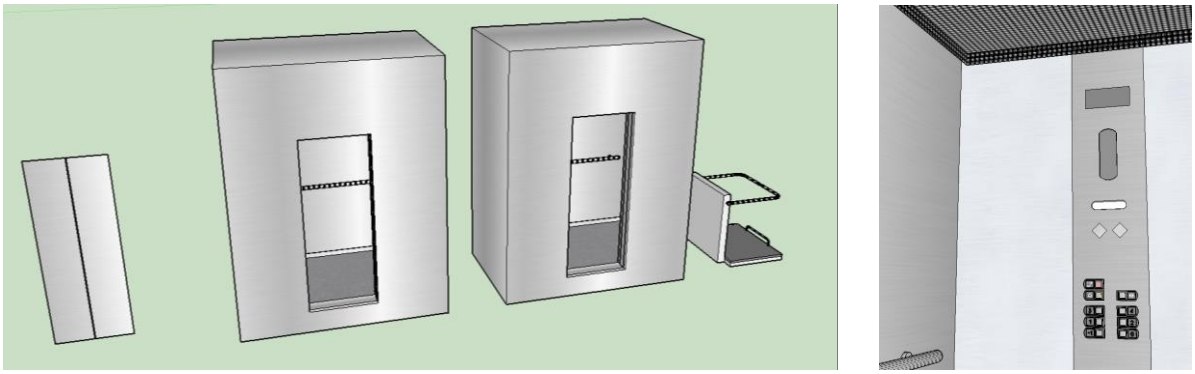


Fig. 36. Modelado de Ascensores

Algunos elementos más complejos y menos característicos de la escuela se obtuvieron de la 3D WAREHOUSE de SketchUp. 3D WAREHOUSE es una herramienta que permite a la comunidad compartir elementos 3D y utilizarlos en tu proyecto.

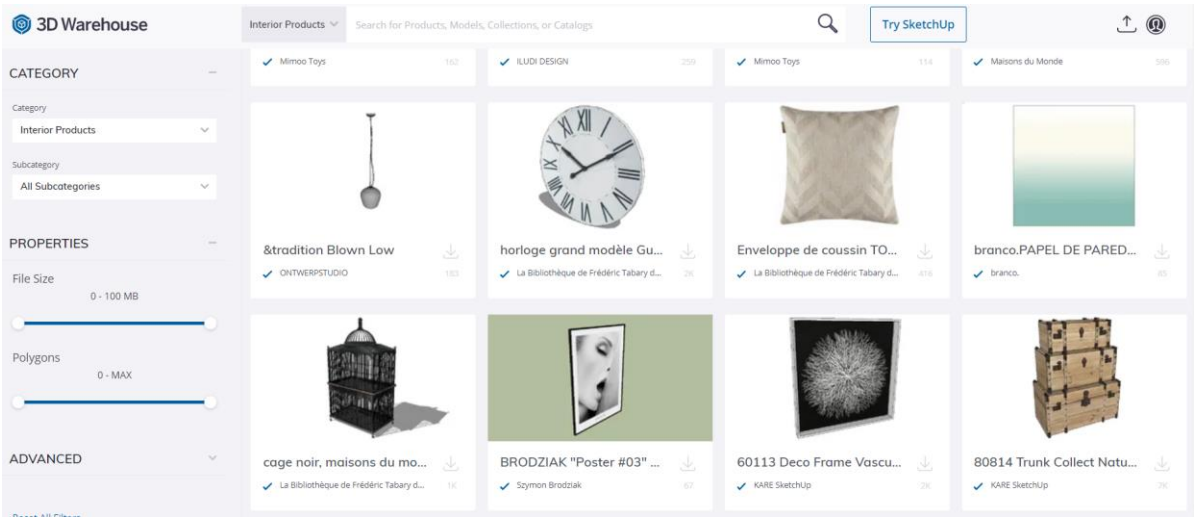


Fig. 37. Página de WAREHOUSE 3D

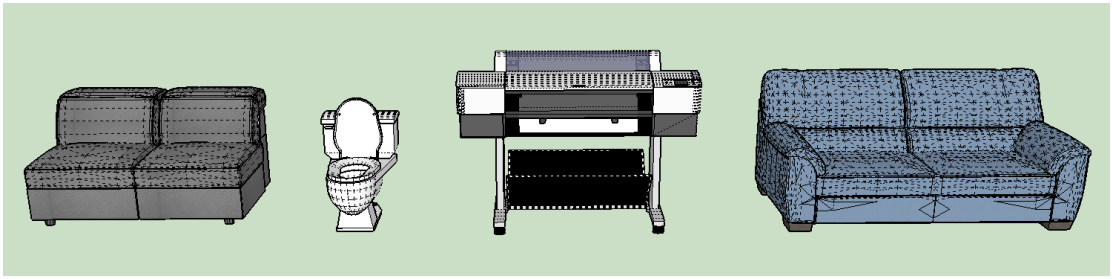


Fig. 38. Modelados obtenidos de WAREHOUSE 3D

Algunos elementos obtenidos del 3D WAREHOUSE han servido para terminar modelados propios, este es el caso de las plantas de las macetas y la grifería de los aseos. Por otro lado, en otros elementos se ha partido del modelo del 3D WAREHOUSE y se ha ido modelando desde ahí para que se asemeje al existente en el edificio, se realizó este procedimiento con las impresoras.

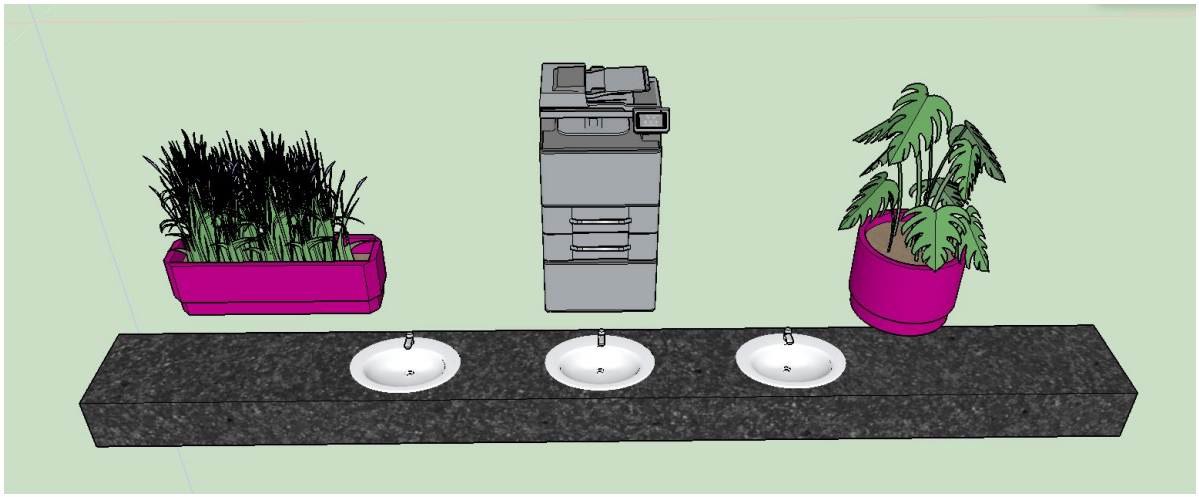


Fig. 39. Modelos mixtos – Parte modelada y parte obtenida de WAREHOUSE 3D

Se debe destacar que, para la creación de todos los objetos del edificio, se ha tenido en cuenta elementos que existen sólo en las zonas que hemos delimitado como accesibles, esto hace que muchos objetos existentes en la escuela no se hayan modelado. También se ha de mencionar que elementos muy complejos, como restituidores clásicos, han sido sustituidos por otro modelo más sencillo de modelar.

2.2. MARCADO DEL POSICIONAMIENTO DE LOS OBJETOS

El posicionamiento de los objetos en el edificio se realizará en UNITY, ya que, si los importamos todos al motor de videojuegos, este interpreta todas las geometrías como diferentes, ralentizando así la carga de las geometrías. Se pretende que un objeto que aparece más de una vez en el edificio sea procesado como un único objeto, es por ello que debemos dejar marcado en el modelado del edificio marcas que nos indiquen donde van posicionados los objetos. Se realizará este proceso en SketchUp, ya que brinda mejores opciones y herramientas de medición que UNITY.

En este procedimiento se han empleado mediciones relativas mediante cinta métrica, debido a que el objeto del producto final no pretende tener una precisión milimétrica, es completamente válido el empleo de este instrumento.

Para la realización de este proceso, dentro de cada una de las capas de las distintas plantas del edificio, se crea un grupo nuevo y mediante la herramienta de TAPE MEASURE TOOL, se va creando guías de las dimensiones que ocupa el elemento en la superficie, posteriormente se crea un polígono delimitando la zona del objeto.

Fig. 40. Procedimiento de marcado de objetos

Una vez finalizado este proceso en todas las plantas del edificio, ya se tiene todo lo necesario para empezar a trabajar con UNITY, dando vida y funcionalidad al modelado.

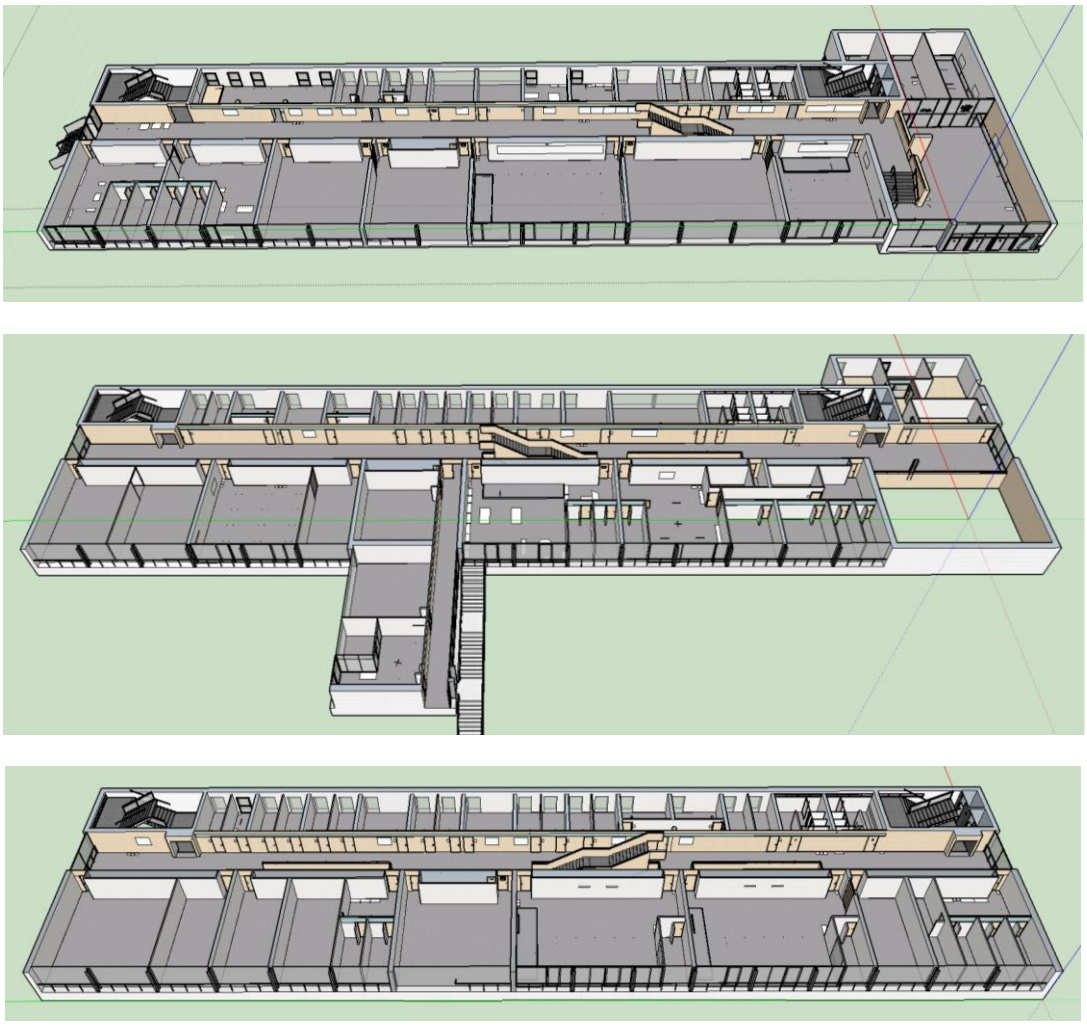


Fig. 41. Resultado del marcado de objetos de algunas plantas

3. *DESARROLLO BÁSICO EN UNITY*

3.1. CREACIÓN DEL PROYECTO EN UNITY

El primer paso es crear el proyecto en UNITY, para ello seleccionaremos entre los TEMPLATES la opción de 3D, esto nos creará un nuevo proyecto con dos GAME OBJECTS básicos. El primero es la cámara y el segundo la luz global de la escena.

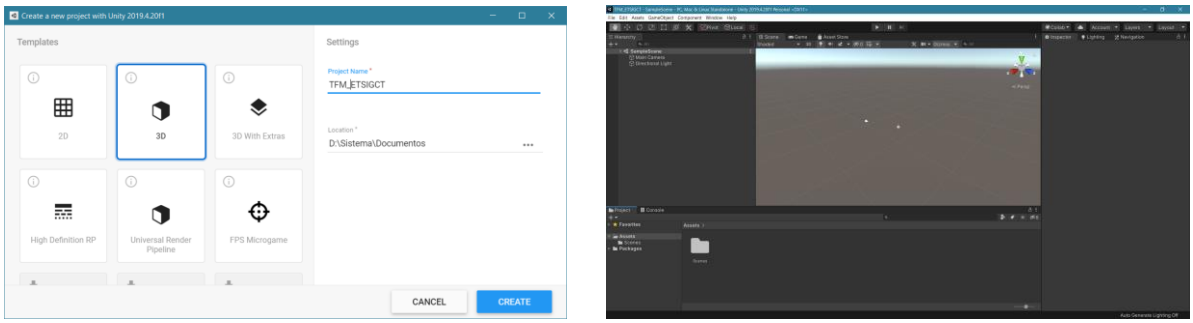


Fig. 42. Creando el proyecto en UNITY

Creamos el sistema de carpetas que vamos a utilizar:

- **FONTS**: Fuentes que se utilizarán en el proyecto.
- **MATERIALS**: Materiales e imágenes existentes en modelo.
- **MODELS**: Modelos desarrollados en SketchUp.
- **PLUGINS**: Carpeta donde se pondrán todos los archivos de PLUGINS de terceros o de UNITY.
- **PERFABS**: PREFABS creados en UNITY.
- **RESOURCES**: Elementos que se obtendrán durante la ejecución de la aplicación.
- **SCENES**: Escenas existentes en la aplicación.
- **SOUNDS**: Sonidos empleados en el proyecto.
- **SPRITES**: Imágenes empleadas como SPRITES.

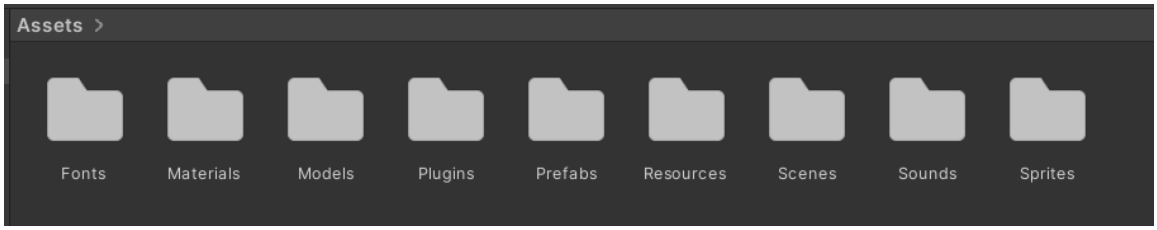


Fig. 43. Sistema de carpetas empleado en UNITY

3.2. CARGA DEL MODELADO DE LAS PLANTAS DEL EDIFICIO

En este apartado vamos a ver como a partir del modelado de la planta en SketchUp podemos importarlo a UNITY, visualizando correctamente los materiales, estructurando bien los modelados y generando bien las físicas de las estructuras y elementos del modelo. Este procedimiento se aplicó para todas las plantas modeladas.

3.2.1. EXPORTACIÓN DEL MODELADO DE LA PLANTA A UNITY

Una vez creado el proyecto en UNITY, pasaremos a cargar los modelos creados en SketchUp, para ello accedemos al programa y dejaremos activa sólo una de las LAYERS, seguidamente accederemos a FILE / EXPORT / 3D MODEL. Una vez allí, exportaremos el modelo en formato COLLADA. Realizaremos este procedimiento con todas las LAYERS creadas del edificio.

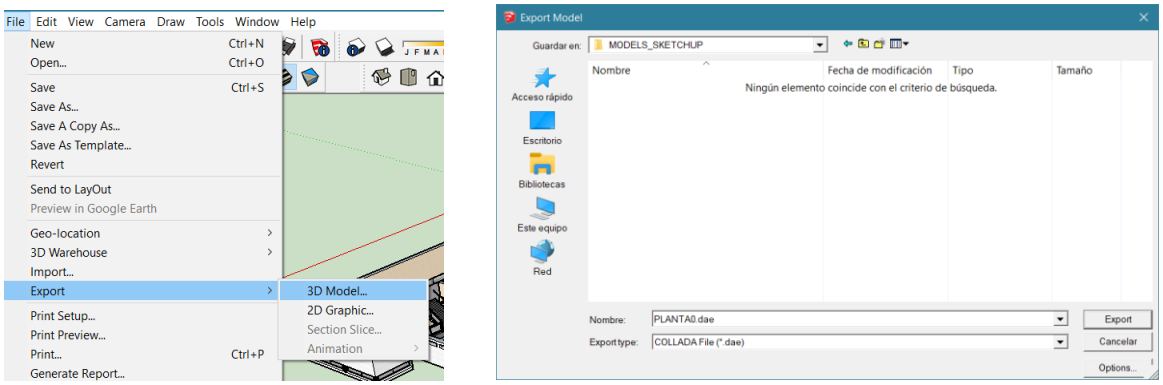


Fig. 44. Exportando el modelo desde SketchUp

Debido a problemas de compatibilidad entre los modelos generados por SketchUp y UNITY, se ha empelado el programa auxiliar, BLENDER, utilizaremos este programa como puente de adecuación de los modelos.

Partiremos de una escena vacía, en ella importaremos el modelo 3D, para ello accederemos a FILE / IMPORT / COLLADA y seleccionaremos el fichero.

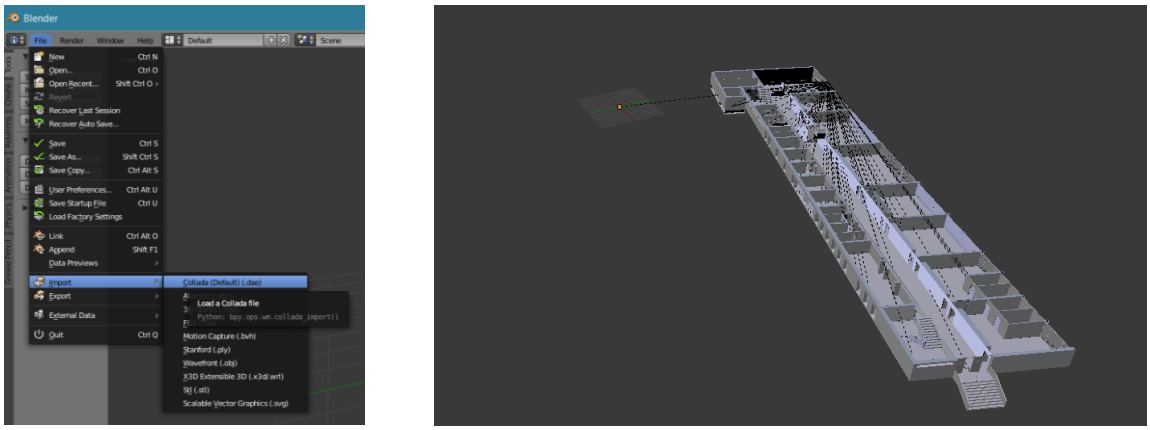


Fig. 45. Modelo en BLENDER

Ahora la exportaremos en la carpeta ASSETS del proyecto de UNITY, esto nos creará los materiales dentro de la carpeta MATERIALS del proyecto. Seguidamente si arrastramos el modelo al espacio de trabajo de UNITY, veremos su representación 3D. Para disponer de una buena organización de los datos, se moverá el asset a la carpeta MODELS tras importarlo a UNITY.

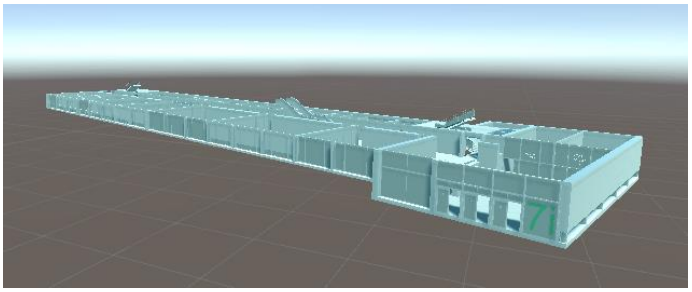


Fig. 46. Modelo en UNITY

3.2.2. MODIFICACIÓN DE LAS TEXTURAS

Como podemos apreciar en la imagen anterior, el modelo no dispone de los materiales correctamente definidos, para solucionarlo accedemos a la carpeta MATERIALS y modificamos las características de los materiales para adecuarlos al modelado. Se ha creado una carpeta para importar las imágenes asociadas a los distintos materiales

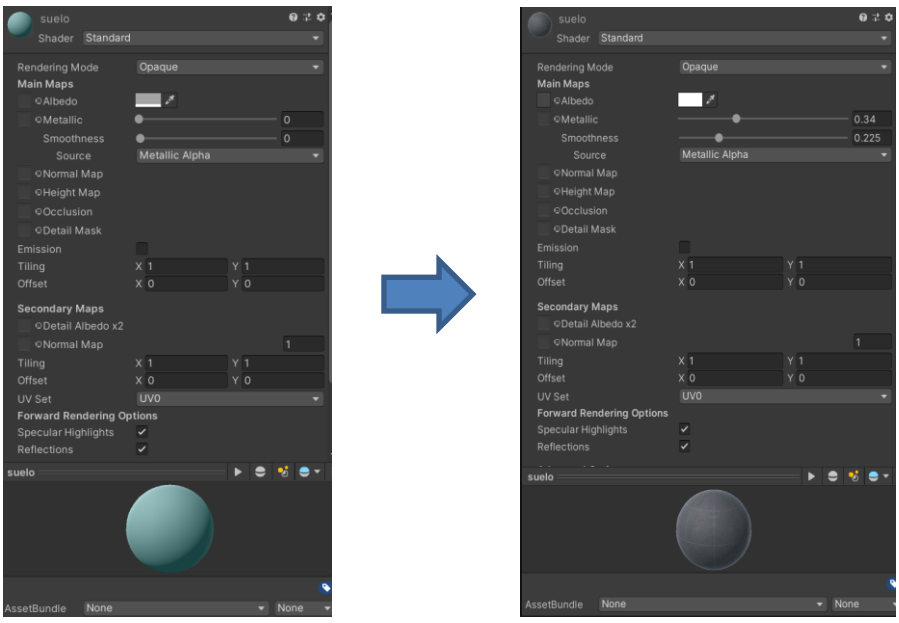


Fig. 47. Modificación de los materiales

Una vez modificados todos los materiales generados, podremos ver como nuestro modelo ya dispone de un aspecto más similar a la realidad.

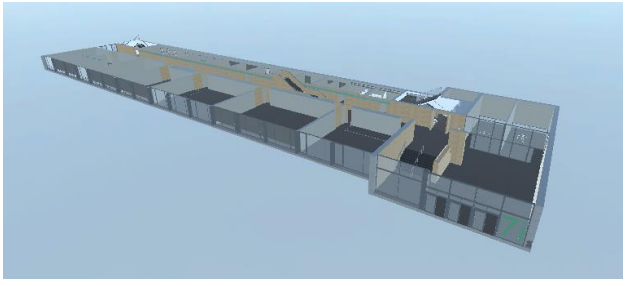


Fig. 48. Modelo en UNITY tras modificación de los materiales

3.2.3. CORRECCIÓN DE LA ESTRUCTURA DEL MODELADO

Si revisamos la estructura de elementos que se han creado en UNITY, se puede observar que no está debidamente ordenada, es por ello que se van a crear GAME OBJECTS vacíos para estructurar correctamente el modelado, separando así elementos en distintos grupos dentro del modelado. Además, se han borrado objetos vacíos, estos se han generado de forma residual en SketchUp.

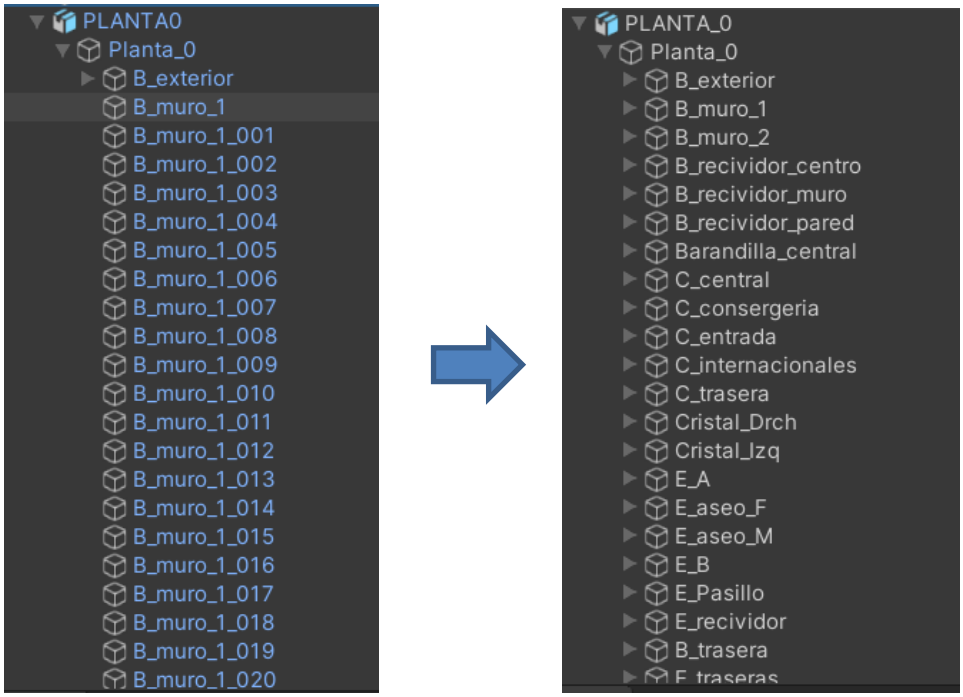


Fig. 49. Estructuración de los modelados

3.2.4. GENERACIÓN DE COLLIDERS

En este apartado generaremos los COLLIDERS, estos elementos dan a las geometrías la parte física de las mismas, haciendo así que una pared no se pueda atravesar o un suelo se pueda pisar. Se pueden crear todo tipo de COLLIDERS, para un mejor rendimiento lo habitual es emplear simplificaciones de estos, utilizando elementos de pocas aristas, pero debido a la complejidad de los elementos existentes de estructura se ha empleado los MESH COLLIDERS. Estos COLLIDERS se adaptan directamente a la forma del modelado, los emplearemos para las estructuras. Para añadirlos, simplemente se seleccionan los elementos y se añade sobre ellos un nuevo componente de tipo MESH COLLIDER.

Fig. 50. Generación de los MESH COLLIDERS

Debido a que estos COLLIDERS requieren mucho tiempo de computación, usaremos otros más sencillos para elementos menos complejos. El más empleado es el BOX COLLIDER, el cual asemeja la física del modelo a un cubo.

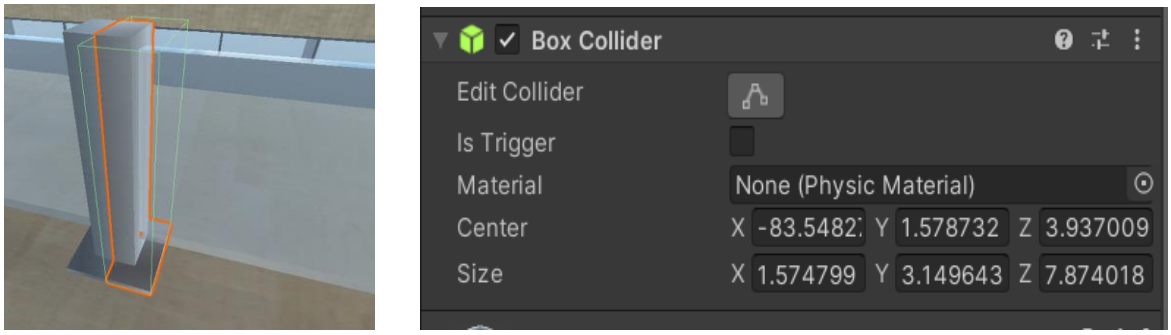


Fig. 51. Generación de los BOX COLLIDERS

3.2.5. ENSAMBLAJE DE LAS PLANTAS DEL EDIFICIO

Finalizaremos la carga del modelado de las plantas, ensamblándolas entre ellas a partir de los distintos puntos de conexión.

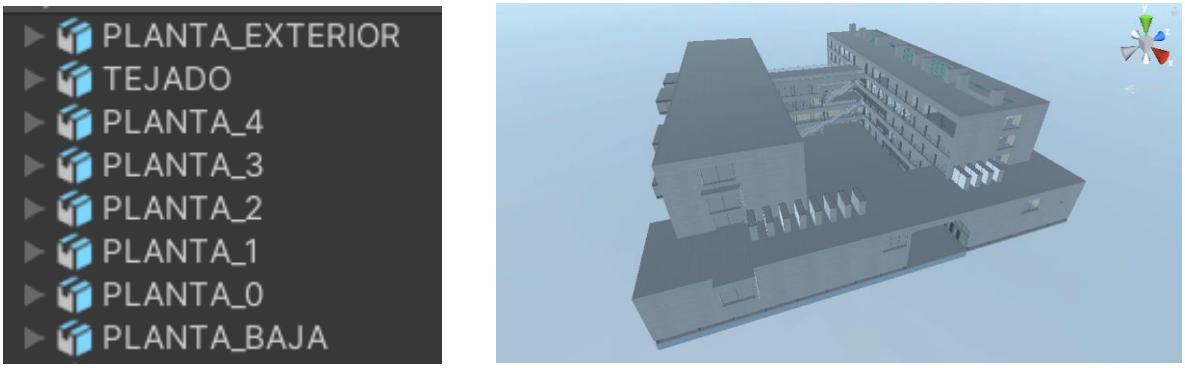


Fig. 52. Resultado de la importación del modelado de las plantas

3.3. CARGA DEL MODELADO DE LOS OBJETOS

La carga de los modelados de los objetos en UNITY es muy similar a la de las estructuras, sin embargo, disponen de pasos adicionales para su correcta integración con el programa. Partiremos de haberlas ya importado, arreglado sus materiales, estructurados sus modelados y generados sus COLLIDERS. El procedimiento descrito se ha empleado para los distintos objetos.

3.3.1. CREACIÓN DE UN PREFAB

Debido a que estos objetos se encuentran repetidos a lo largo de las estancias del edificio, se va a utilizar la creación de PREFABS. Se puede entender un PREFAB como una definición del objeto, siendo así este el original y todos los existentes referencias al este, es por ello que si se modifica el PREFAB todas las instancias del mismo se ven modificadas. El uso de PREFABS mejora el rendimiento de cargas de geometrías, además de permitir modificar sólo una instancia genérica de un objeto y tener actualizados todos los existentes en la escena.

Para crear un PREFAB a partir de un objeto sólo debemos arrastrar el objeto a una carpeta de ASSETS, en nuestro caso dentro de la carpeta PREFABS, dentro de ella generaremos distintos directorios para tener mejor organizados estos PREFABS.

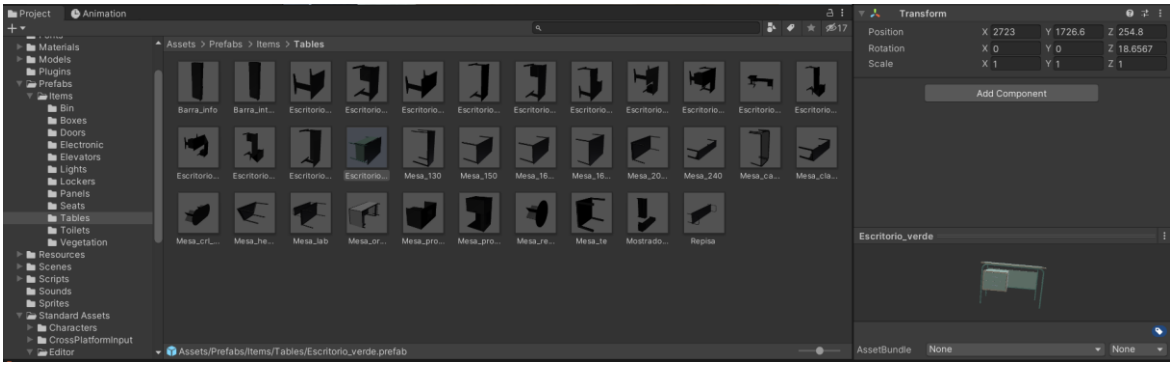


Fig. 53. Ejemplo de PREFAB

3.3.2. POSICIONAMIENTO DE LOS PREFABS

Una vez disponemos de los distintos PREFABS, vamos a posicionarlos correctamente en la escena, para ello nos ayudaremos de los elementos guías que creamos en SketchUp e importamos con las plantas del edificio a UNITY. Para ello, nos ayudaremos de las herramientas de posicionamiento a aristas de geometría que nos proporciona el programa, con ellas iremos posicionando cada instancia de los distintos PREFAB correctamente en la escena.

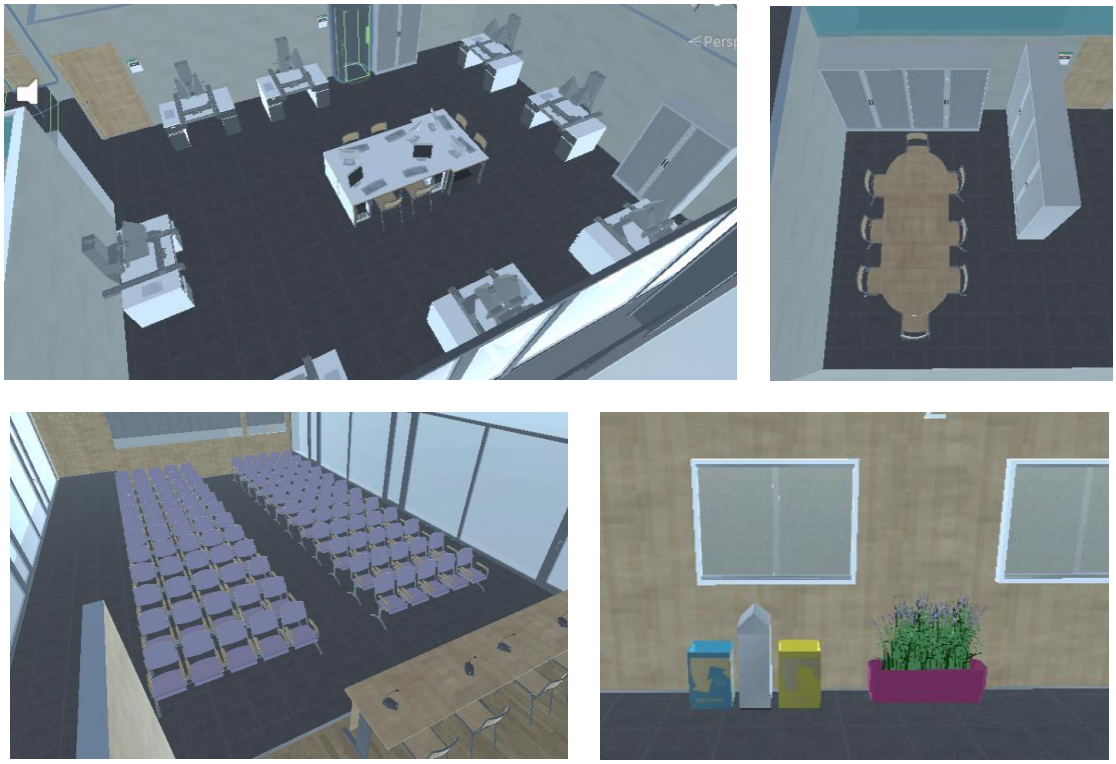


Fig. 54. Posicionamiento de los PREFABS

3.3.3. ESTRUCTURACIÓN DE LOS PREFABS

Una vez disponemos de los distintos PREFABS, los separaremos en distintos grupos por planta del edificio. Teniendo así los siguientes grupos por cada una de las plantas:

- **C&PA**: Dispondrá de varios grupos:
  - **CARTELES**: Carteles existentes al lado de cada una de las estancias de la planta. Cada uno tendrá el código de la habitación a la que hace referencia
  - **PANELES**: Contendrá los paneles verticales que especifican algunas habitaciones. También tendrán el código de la habitación a la que hace referencia.
  - **DIRECTORIOS**: Contendrá los directorios existentes en las plantas.

- **DoorsFloor**: Contendrá las puertas existentes en las plantas.

- **Deco**: Contendrá los distintos objetos de decoración que tiene la planta, estarán agrupados por habitación.

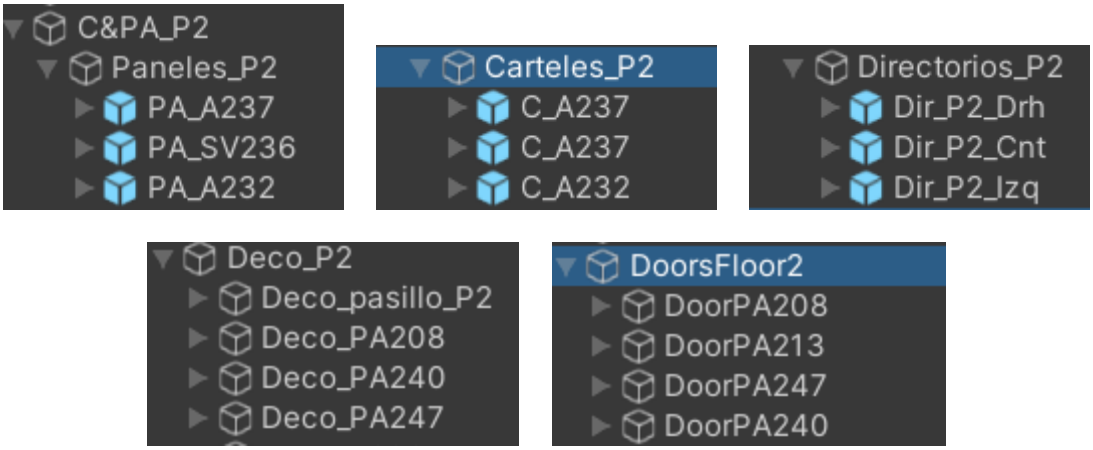


Fig. 55. Estructuración de los PREFABS

3.4. ADICCIÓN DE LOS ASSETS BÁSICOS DE UNITY

UNITY pone a disposición de los desarrolladores una serie de ASSETS básicos, desde el ASSET STORE se puede descargar un pack con scripts, modelados, partículas etc.…

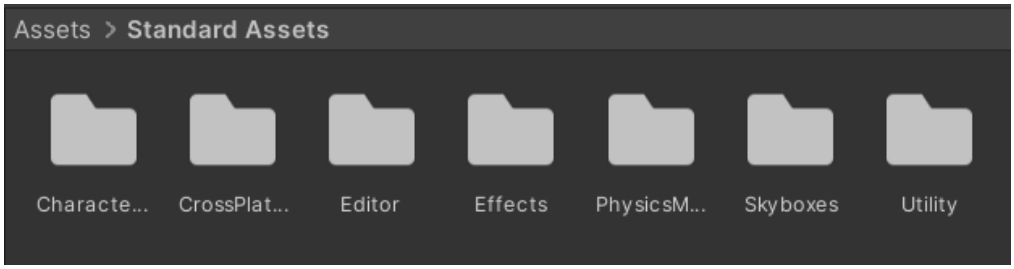


Fig. 56. Standard Assets

Utilizaremos el PREFAB que nos proporciona el FPSController de los nuevos ASSETS. Este ASSET nos proporciona la programación del movimiento del personaje, la cámara y el captador de sonido.

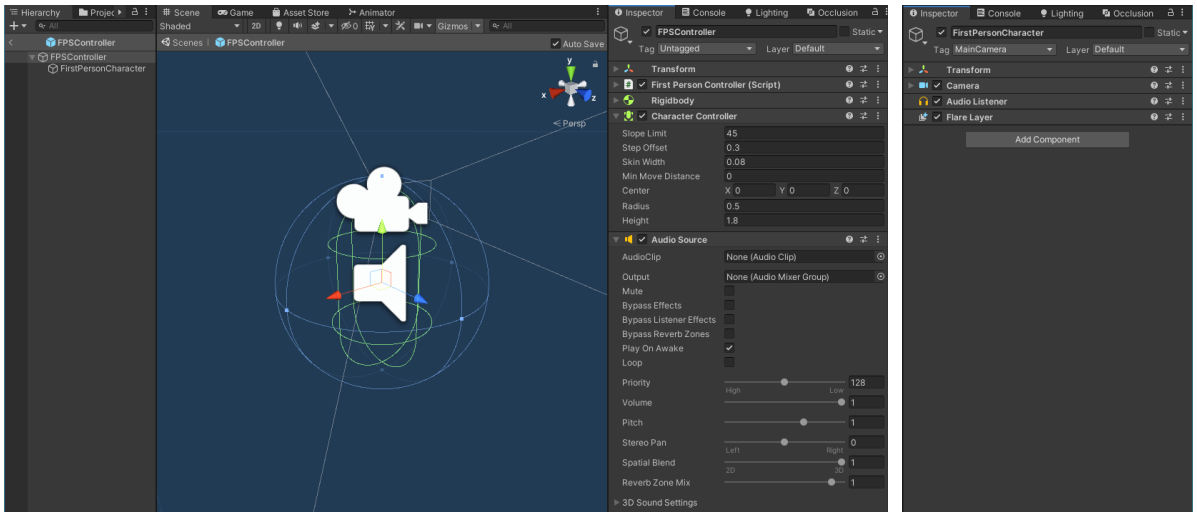


Fig. 57. FPSController

Añadiremos este ASSET a nuestro proyecto, con él podremos ya movernos libremente por el modelado. En él tenemos programado tanto el movimiento, salto y orientación de la cámara. El PREFAB nos permite modificar ciertos parámetros para adecuarlo al proyecto, se modificó la velocidad del personaje, la altura de salto, focales de la cámara, la altura y pendiente máxima que puede subir sin salto.

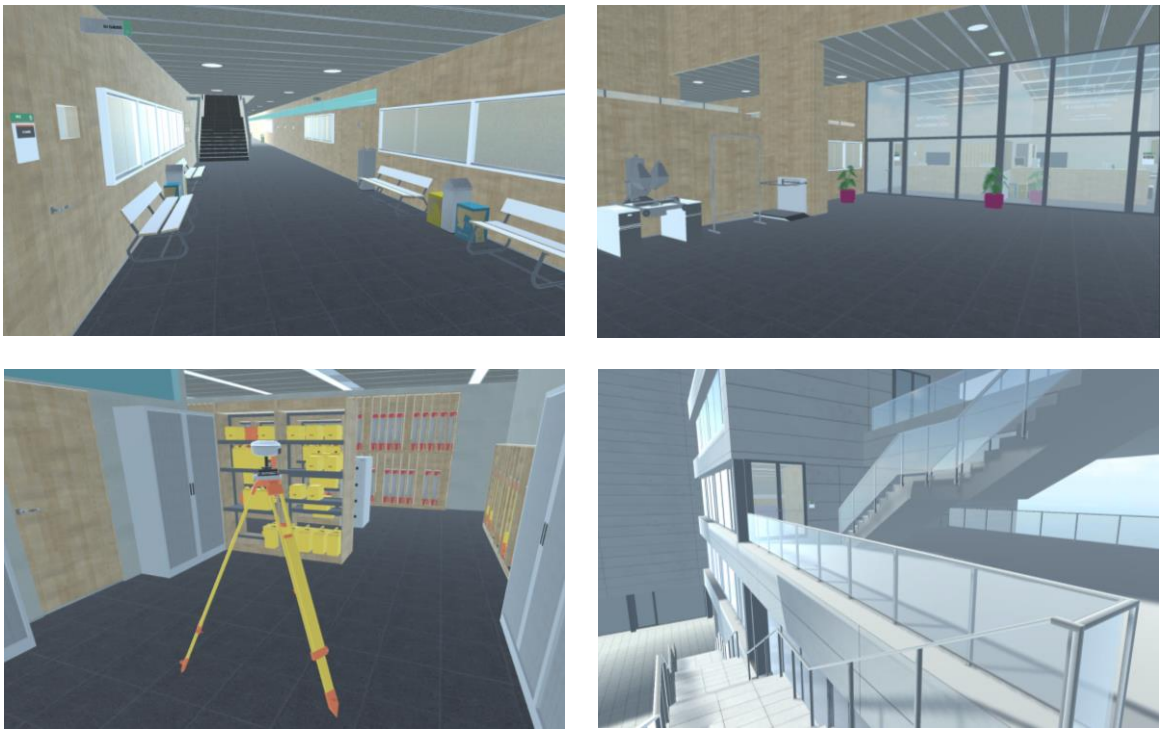


Fig. 58. Vistas con el FPSController

3.5. ADICCIÓN DE BASE DE DATOS

3.5.1. DEFINICIÓN DEL MODELO DE DATOS

En este apartado vamos a añadir la base de datos, el modelo planteado para la aplicación es similar al empleado en el TFG, expuesto en la sección datos de partida, debido a que partimos de esos datos. Sin embargo, se han borrado campos / tablas innecesarias y se ha añadido algún campo / tabla extra.

MODELO DE DATOS

SERVICIOS
ID_SERVICOS (Txt)
NOMBRE (Txt)
LABEL_NAME (Txt)
DESCRIPCIÓN (Txt)
HORARIO (Txt)
URL (Txt)
MÁS INFO (Txt)
TIPO (Int)
PLANTA (Int)

SEMINARIO
ID_SEMINARIO (Txt)
NOMBRE (Txt)
LABEL_NAME (Txt)
DEPARTAMENTO (Txt)
DOTACIÓN (Txt)
UNIDAD DOCENTE (Txt)
PLANTA (Int)

PERSONAL
ID_PERSONAL (Txt)
NOMBRE (Txt)
APELLIDOS (Txt)
DEPARTAMENTO (Txt)
UNIDAD DOCENTE (Txt)
TIPO (Txt)
E-MAIL (Txt)
MÁS INFO (Txt)

AULAS
ID_AULAS (Txt)
NOMBRE (Txt)
LABEL_NAME (Txt)
HORARIO (Txt)
TIPO (Int)
N.º ASIENTOS (Int)
N.º ORDENADORES (Int)
PLANTA (Int)

1 → 1

DESPACHO
ID_DESPACHO (Txt)
ID_PERSONAL (Txt)
PLANTA (Int)
TIPO (Int)

MISIONES
ID_QUEST (Txt)
NOMBRE (Txt)
DESCRIPCION (Txt)
OBJETIVO (Txt)
DESCRIPCION_CORTA
PLANTA (Int)
ID_ROOM (Txt)
X (Real)
Y (Real)
Z (Real)
ROTATE_X (Real)
ROTATE_Y (Real)
ROTATE_Z (Real)
MAP_X (Real)
MAP_Y (Real)

LEYENDA	
CLAVE PRIMARIA	CLAVE AJENA
TIPO CODIGO	HIPERVÍNCULO

Tabla 8. Modelo de datos de la aplicación

Al igual que el modelo de datos de partida, existen dos tablas iguales a SERVICIOS, una para FUNDACIÓN y otra para la DICGF.

En todas las tablas a importar, salvo PERSONAL y DESPACHO, se ha añadido un campo extra denominado LABEL_NAME, este campo dispondrá de la misma información que nombre, su empleo

será el usarlo en los letreros de los distintos espacios de la aplicación, permitiendo acortar o editar el nombre para mejorar su visualización sin afectar al nombre original.

Se añadió una tabla nueva denominada MISIONES, esta dispondrá de toda la información necesaria para disponer de esta funcionalidad, veremos en la sección EXTENSIÓN QUEST más información al respecto.

3.5.2. EXPORTACIÓN DE LA BASE DE DATOS

Para esta aplicación vamos a emplear una base de datos de tipo SQLite, ya que no son muy pesadas, tienen buen rendimiento, son muy portables, son de dominio público y tienen compatibilidad con C#, lenguaje en el que programaremos nuestras funcionalidades para UNITY.

Debido a que no necesitamos todas las tablas y no existe un método directo para pasar de una geodatabase a una base de datos de SQLite, el procedimiento de exportación se traduce en exportar tabla a tabla a formato *CSV* e importarlas en una base de datos creada en SQLite.

Primero accederemos a ArcCatalog y buscaremos nuestra geodatabase, allí seleccionaremos la tabla a exportar y desde *Preview* accedemos a *Export*. En la ventana de exportación seleccionamos *All records* y definimos el fichero de salida, indicando que se trata de tipo *Text file.* Realizaremos este procedimiento con todas las tablas y features que vamos a emplear en el modelo de datos de la aplicación, generando así todos los ficheros *CSV*.

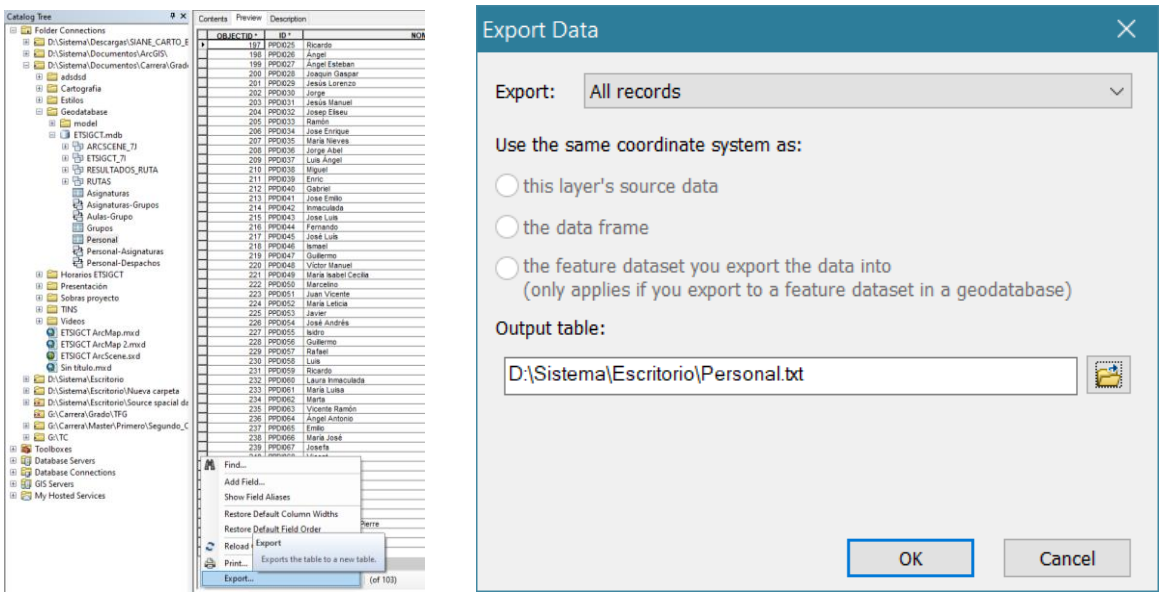


Fig. 59. Exportando la base de datos

Para la gestión de la base de datos emplearemos el programa DB BROWSER FOR SQLITE, programa de código abierto ampliamente utilizado para gestionar bases de datos SQLite. Accedemos al programa y seleccionamos *Nueva base de datos*, la guardaremos dentro del proyecto de UNITY, en la carpeta Resources nos crearemos una nueva llamada *Database*, nombraremos a la base de datos con el nombre de ETSIGCT. Una vez creada, vamos a importar los distintos *CSV* que exportamos de *ArcCatalog*. Para ello accedemos a *Archivo / Importar / Tabla de archivo CSV*, esto nos abrirá una

ventana donde gestionaremos la importación, permitiéndonos modificar algunos parámetros, para nuestro caso con los valores por defecto nos detectará correctamente las cabeceras, separación de campos, registros existentes y la codificación.

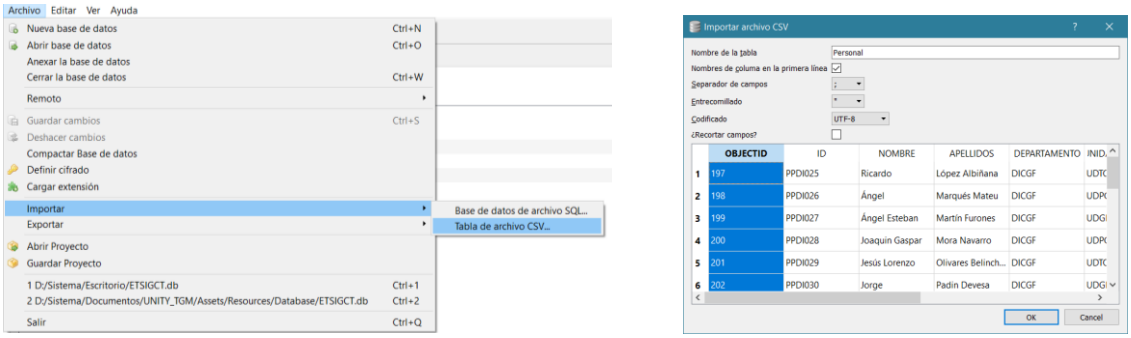


Fig. 60. Importando CSV a la nueva base de datos

Una vez importadas, vamos a modificar y adecuar los datos, ya que por defecto todos los campos se importan como tipo texto. Para ello seleccionamos la tabla a modificar y accedemos a *Modificar Tabla,* desde esta ventana podemos modificar los tipos de los campos, borrar campos, crear campos, establecer claves primarias y ajenas y poner valores por defecto. Modificaremos todas las tablas para adecuarnos al modelo de datos definido.

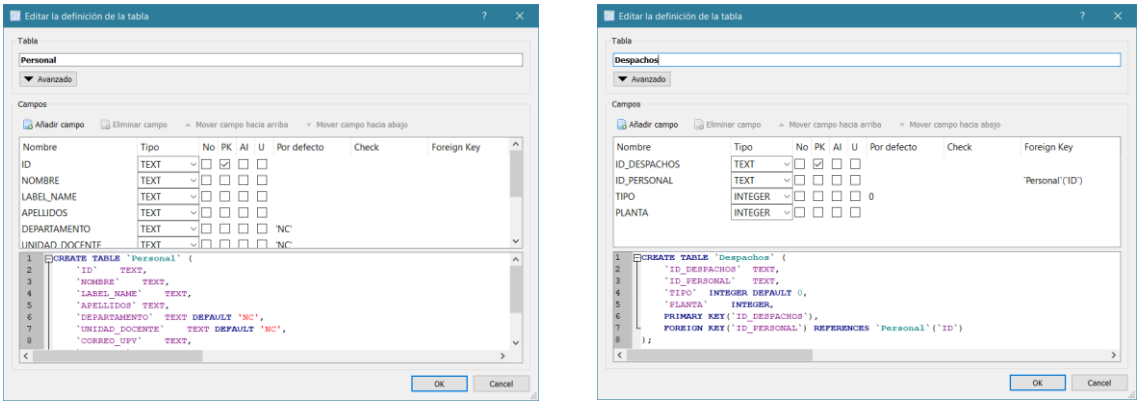


Fig. 61. Adecuando las tablas importadas

Para dejar terminado el modelo de datos sólo necesitaremos definir la nueva tabla de misiones. DB BROWSER FOR SQLITE nos permite añadir de forma sencilla nuevas tablas, para ello accedemos a *Crear Tabla,* una vez allí vamos definiendo los distintos campos y vamos modificándolos de la misma forma que en el proceso anterior.

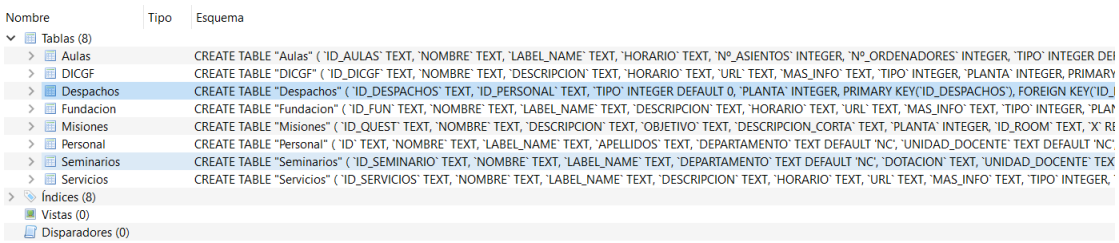


Fig. 62. Resultado final de la importación del modelo de datos

3.5.3. ACCESO A LA BASE DE DATOS DESDE LA APLICACIÓN

Para poder acceder a la base de datos desde la aplicación debemos programar la lógica de acceso. UNITY ofrece la posibilidad de programar en dos lenguajes de programación: C# y UnityScript, optaremos por realizar todos nuestros scripts en C#, debido a que C# es más potente, proporciona mejor rendimiento y está más extendido. Para el acceso a la base de datos vamos a crearnos una clase denominada *DatabaseGame*, esta proporciona los métodos para las conexiones a la base de datos, así como las peticiones directas contra ella.

```
using System;
using Mono.Data.Sqlite;
using UnityEngine;
using System.Data;
using System.Collections.Generic;

class DatabaseGame : MonoBehaviour {

    private string dbPath;
    private SqliteConnection dbConnection;
    private SqliteCommand dbCommand;
    private SqliteDataReader dbReader;

    public void dbOpen (string dbName) {
        dbPath = "URI=file:"+ Application.dataPath + "/Resources/Database/" + dbName;
        dbConnection = new SqliteConnection(dbPath);
        dbConnection.Open();
    }

    public string obtainField(string query) {
        List<string> dataList = obtainSingleElement(query);
        return dataList == null || dataList.Count == 0 ? null : dataList[0];
    }


    public List<string> obtainSingleElement(string query) {
        List<List<string>> dataList = obtainMultipleElements(query);
        return dataList == null || dataList.Count == 0 ? null : dataList[0];
    }

    public List<List<string>> obtainMultipleElements(string query) {
        List<List<string>> dataList = new List<List<string>>();
        dbCommand = dbConnection.CreateCommand();
        dbCommand.CommandText = query;
        dbReader = dbCommand.ExecuteReader();
        if (dbReader != null) {
            while (dbReader.Read()) {
                List<string> dataListRow = new List<string>();
                for (int i = 0; i < dbReader.FieldCount; i++) {
                    dataListRow.Add(dbReader.GetValue(i).ToString());
                }
                dataList.Add(dataListRow);
            }
            return dataList;
        }  else {
            return null;
        }
    }
```

```
    public void dbClose () {
        dbReader.Close();
        dbReader = null;
        dbCommand.Dispose();
        dbCommand = null;
        dbConnection.Close();
        dbConnection = null;
    }
}
```

En el cuadrado rojo se encuentran las variables de la clase, entre ellas el *path* de la base de datos y los tipos propios de SQLite de conexión y acceso a la base de datos, todos ellos privados para no poder acceder desde fuera de la clase.

En los cuadrados azules se encuentran los métodos de conexión y desconexión a la base de datos.

En el cuadrado naranja está el método de acceso a información de la base de datos, donde dado una QUERY la ejecuta en la base de datos y con el resultado compone una lista de listas. La lista superior corresponderá a la cantidad de registros sacados de una tabla, mientras que la segunda será los distintos campos asociados a ese registro. Si lo viésemos con un ejemplo, la primera lista sería cada registro de profesores de la base de datos, por otro lado, la segunda lista haría referencia a los campos de información de cada profesor (nombre, apellidos, correo…).

En los cuadrados verdes podemos ver implementaciones del método anterior, pero devolviendo un único registro de una tabla (Ejemplo un profesor) o un único atributo de un registro (Ejemplo el nombre de un profesor).

3.6. CREACION DE PANTALLAS DE INTERFAZ UI

Antes de adentrarnos en el desarrollo de programación de nuevas funcionalidades, vamos a diseñar y crear las distintas interfaces UI existentes en la aplicación. Dispondremos de las siguiente interfaces UI:

- **PANEL START:** Dispondrá de un menú inicial de la aplicación, permitirá, ver información, iniciar y salir de la aplicación.
- **PANEL GAME:** Pantalla visible en todo momento mientras nos movamos libremente por el modelado. Dispondrá de elementos de ayuda para el usuario.
- **PANEL ROOM:** Mostrará la información relevante de un espacio de la escuela, tendrá diferente información en función del tipo de sala que consultemos.
- **PANEL MAP:** Pantalla que nos proporcionará los mapas de las distintas plantas de la escuela. Permitirá cambiar entre distintas plantas, ver nuestra posición, ver la posición de la misión y emplear los teletransportes.
- **PANEL QUEST:** Dispondrá de todas las misiones disponibles para realizar, permitiendo ver su información asociada y la posibilidad de realizarlas.

- **PANEL MENSSAGE QUEST:** Panel que nos mostrará información de ayuda cuando nos acerquemos al final de una misión.

- **PANEL ELEVATOR:** Pantalla que nos permite seleccionar el piso al que queremos acceder y nos muestra información de interés de la planta.

3.6.1. PANEL START

En este paso vamos a diseñar la pantalla inicial de la aplicación. La intención de esta pantalla UI es disponer de una pantalla inicial, en la que poder iniciar la aplicación, mostrar información y poder cerrar la aplicación.

El primer paso será crearnos un *canvas*, que nos hará de lienzo de todos los elementos de interfaz gráfica de la aplicación, renombramos el game object a UI y dejamos todos los parámetros por defecto salvo el componente *Canvas Scaler,* este indica cómo se reescala la resolución de los objetos en función del tamaño, para hacerlo adaptable ponemos el atributo *UI Scale Mode* con el valor de *Scale With Screen Size.*

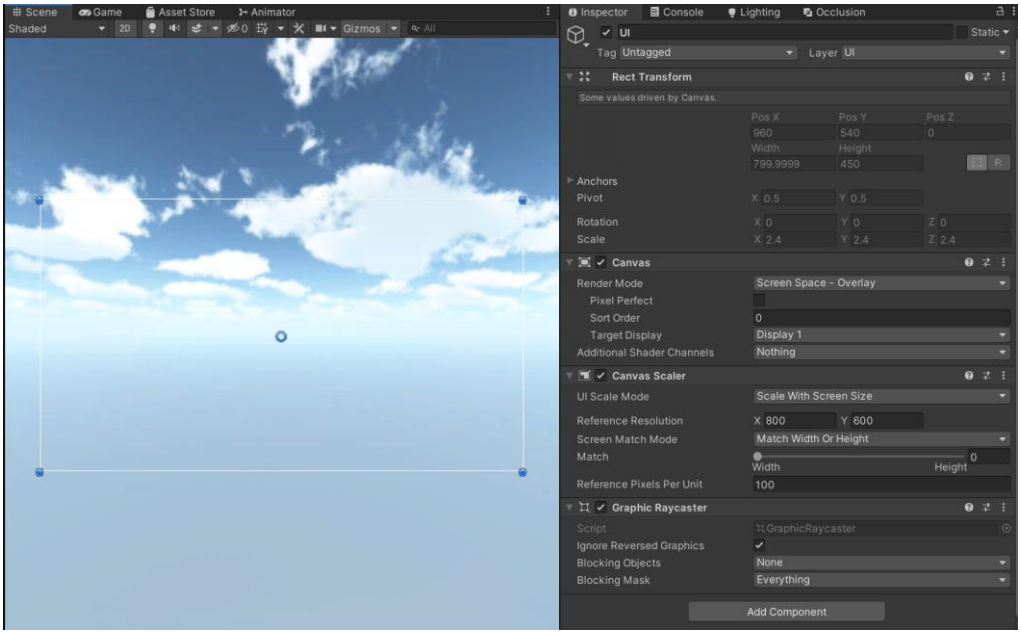


Fig. 63. Creando el canvas UI

Añadiremos como hijo a este canvas un game object de tipo *EventSystem*, este objeto permite que podamos interactuar con todos los elementos de tipo botón que añadamos.

El diseño para la interfaz de inicio parte de una cámara estática, ubicada en la puerta de entrada de la escuela, a la que se le superpone dos secciones. La de la izquierda dispone del logo de la escuela y los botones para interactuar con la interfaz. La de la derecha queda completamente vacía, con el objetivo de mostrar información en caso de ser necesario.

Comenzamos añadiendo un nuevo game object de tipo cámara, le nombraremos como *StartCamara*, la dispondremos en la parte exterior del edificio apuntando a las puertas de la escuela.

Modificaremos los parámetros del componente para adecuar el encuadre. Debido a que no vamos a poder disponer de dos cámaras iniciales, desactivamos el *player*, para no tenerlo del todo desactivado y poder acceder a él por código, creamos un game object vacío y le introducimos player como hijo. Le añadiremos un efecto blur, añadiendo el componente Blur de Standard Assets.

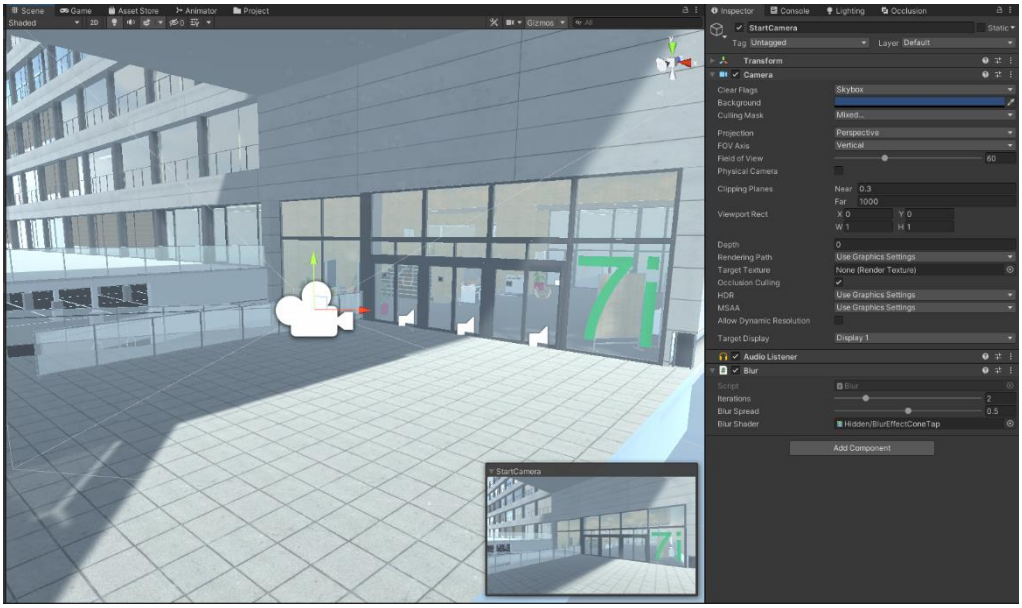


Fig. 64. Posicionando la StartCamera

Ahora es el momento de diseñar y crear la interfaz, para ello crearemos un game object dentro del canvas, será de tipo *Panel*, en él dispondremos de todos los elementos de la pantalla inicial de la aplicación, la nombraremos como *PanelStart*. El elemento panel viene por defecto con un componente *Image*, al cual le quitaremos el *Source Image* asociado, eliminando así los bordes que genera y le modificaremos el color, para darle un toque grisáceo semi transparente.

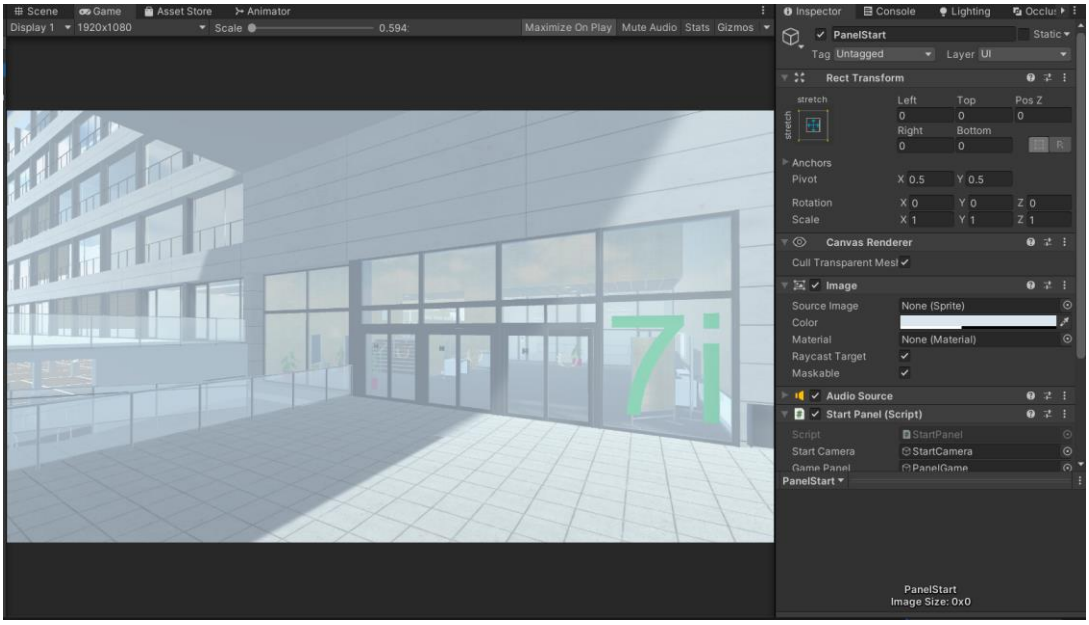


Fig. 65. Añadiendo el PanelStart

El siguiente paso será crear dos nuevos paneles, uno que agrupará los elementos de la zona izquierda y el otro de la derecha, los llamaremos *PanelStartLeftConatiner* y *PanelStartRightContainer*. En ambos paneles les borraremos el componente *image*, no es necesario disponer de él en estos objetos.

Los paneles disponen de un componente llamado Rect Transform, permite restringir su tamaño y añadir distintos tipos de comportamiento en su reescalado. En el panel izquierdo aplicaremos un escalado de tipo *stretch* de altura, lo que indica que ocupará lo máximo posible en altura, mientras que en anchura será de tipo *left*, marcando así que siempre tendrá un fijo máximo, en este caso será de 400 pixeles. Por otro lado, el panel derecho dispondrá de ambos en stretch, para que no superponga al otro panel en anchura, le indicaremos que ocupe todo el ancho salvo 400 pixeles.

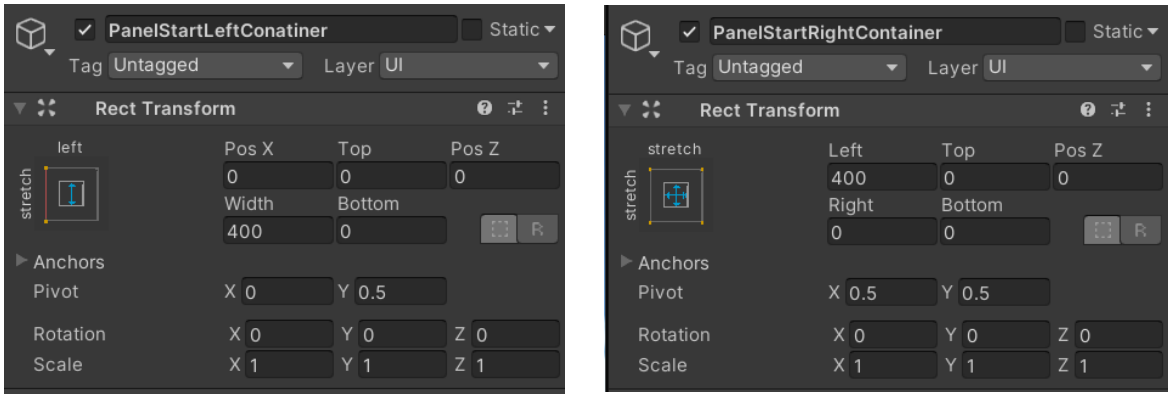


Fig. 66. Modificando el tamaño de los paneles

Debemos destacar el atributo *anchors*, con rango de valores de 0 a 1, este indica desde donde empieza a contar los pixeles introducidos en el componente. En el panel izquierdo, es desde el lado izquierdo y a mitad (0 en X; 0.5 en Y), ya que si contásemos desde el centro el *width* introducido sería de 200 en vez de 400. Para el panel derecho, no es muy importante, ya que la configuración stretch siempre va a posicionarlos desde los extremos.

Continuaremos la interfaz con los elementos del panel izquierdo, aquí disponemos de dos zonas diferenciadas una superior y otra inferior. En la zona superior introduciremos el logo de la escuela. En la zona inferior vendrán los botones para interactuar con el interfaz. Siguiendo el mismo procedimiento dispondremos a cada uno de su espacio.

Vamos a añadir en el panel superior el icono de la escuela, para ello dentro de este añadimos un panel de tipo *stretch* en ambas direcciones y cambiaremos el valor *Source Image* en el componente *image* del panel, allí introduciremos la imagen del logo. Para poder introducirla la debemos importar en nuestra carpeta sprites y al seleccionarla en el inspector, debemos marcarla como *Sprite (2D and UI)*, también debemos marcar en *Advanced* que se trata de un sprite con transparencias (*sRGB y Alpha is Trasparent*), posteriormente la arrastramos al campo *source* image para asignarla. Para que la imagen no se deforme y mantenga su ratio aspecto, marcaremos el campo *Preserve Aspect.*

Fig. 67. Añadiendo el logo de la escuela a la pantalla de inicio

Es la hora de añadir los botones, para ello dentro del panel inferior introducimos tres nuevos game objects de tipo *Button*, los dispondremos uno debajo de otro según el procedimiento visto anteriormente, el de arriba vendrá anclado a la parte superior, el de abajo a la inferior y el central expandido, respetando el hueco de los otros. El primero iniciará la aplicación, el segundo mostrará información acerca de los creadores de la aplicación y el tercero saldrá de la aplicación.

Los botones disponen de un componente (*Button*) que permite modificar su color, se puede modificar el color base, al presionar, al seleccionar, al deshabilitar y realizar hover, se ha optado por una paleta de colores adecuados al logo de la escuela. Este mismo componente permite añadir la funcionalidad al botón, la vincularemos cuando creemos el script para controlar el comportamiento del panel.

Cada *Button* contiene otro game object en el que viene el texto a mostrar en el botón, disponen de un componente de tipo *Text* que permite modificar el texto y las características del mismo (color, tamaño, alineación…). Se ha optado por una paleta de colores acorde al color del botón.

Fig. 68. Añadiendo los botones a la pantalla de inicio

Para terminar, vamos a introducir la información de la autoría de la aplicación, donde aparecerán los datos del alumno y los profesores. Primero crearemos un panel de tipo stretch en ambas direcciones, donde dejaremos un margen en todos los lados de 35 pixeles, menos en el campo left, ya que ese margen lo añade el panel izquierdo. En este panel mantendremos el sprite que nos proporciona UNITY y simplemente modificaremos el color para dejar uno acorde al estilo del resto.

El panel de autoría lo separaremos en dos paneles, el de arriba vendrá con la información del alumno y el de abajo con la de los profesores. Cada panel dispondrá de la misma estructura, en la parte superior vendrá el título y en la parte inferior la información del alumno o profesor. El panel con la información se dividirá en dos secciones, la de la izquierda con una foto y la derecha con la información. La información a mostrar será: Nombre, apellidos y correo electrónico.

Fig. 69. Resultado final de la pantalla de inicio

En el resultado final, podemos ver que hemos añadido en el alumno un texto con LinkedIn, este se trata de un botón al que daremos la funcionalidad posteriormente. Debemos destacar que las imágenes del alumno y los profesores eran cuadradas, se ha añadido un panel padre a cada una de ellas con sprite circular para que haga un recorte de la imagen original, obteniendo así un diseño más moderno y estilizado.

3.6.2. PANEL GAME

En este panel vamos a disponer de todos los elementos de información que tendrá a su disposición para ayudarle durante el recorrido del modelo interactivo. Veamos todas sus características.

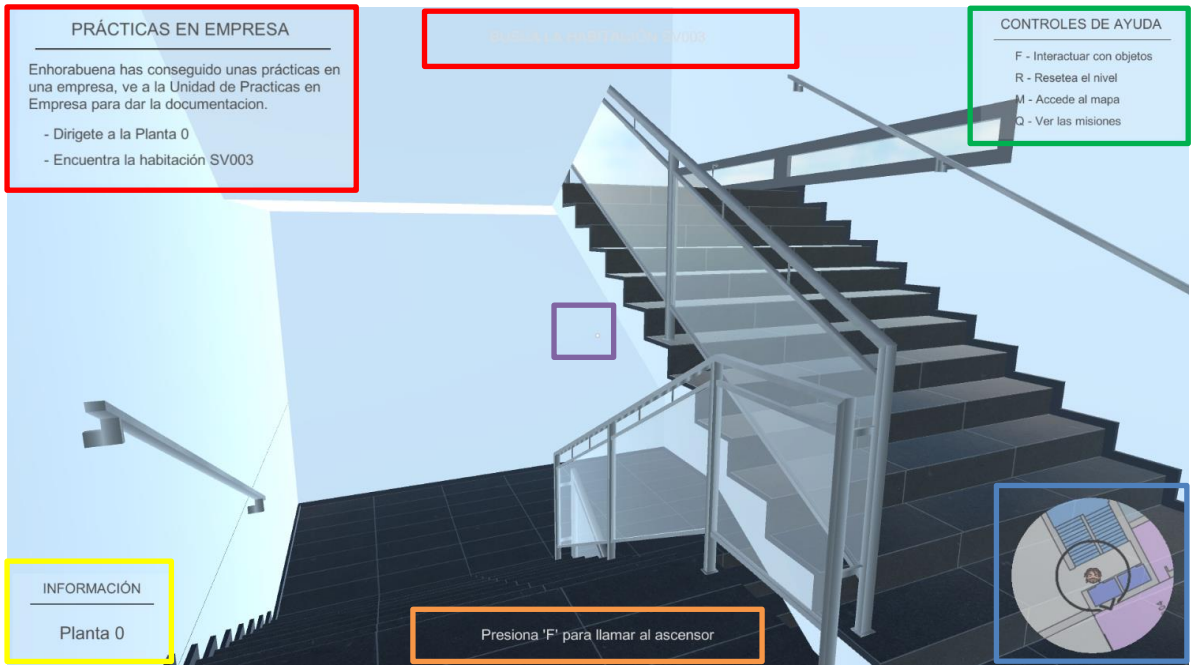


Fig. 70. Información del Panel Game

En el cuadrado verde disponemos de información de interés acerca de los distintos controles que disponemos para el usuario, se han obviado controles de movimiento para no recargar la interfaz. Se tratan de elementos UI vistos en pantallas anteriores, donde encontramos paneles y textos.

En el cuadrado amarillo tenemos información acerca de la localización donde nos encontramos, si accedemos a cualquier habitación de interés nos indicará un nombre. Esta funcionalidad se realizará por código y se verá su actualización en la EXTENSIÓN ROOM. Los elementos UI empleados son textos y paneles.

En el cuadrado naranja aparece un mensaje que nos indica que podemos interactuar con un objeto, este mensaje se mostrará cuando estemos cerca de algún objeto con el que podamos interactuar, su comportamiento se moldeará vía programación. El elemento UI empleado es un texto.

En los cuadrados rojos aparece información relevante de las misiones, todas ellas se actualizan y se muestran vía programación, se verá su comportamiento en la EXTENSIÓN QUEST. El cuadrado a la izquierda indica una pequeña descripción de la misión en curso y sus pasos. El cuadrado a la derecha muestra un mensaje cuando nos acercamos a la misión. Los elementos UI empleados son textos y paneles.

En el cuadrado morado disponemos de un pequeño cursor, se trata de un pequeño círculo blanco que permitirá al usuario tener más claro dónde está su punto de visión. El elemento empleado es un *image*.

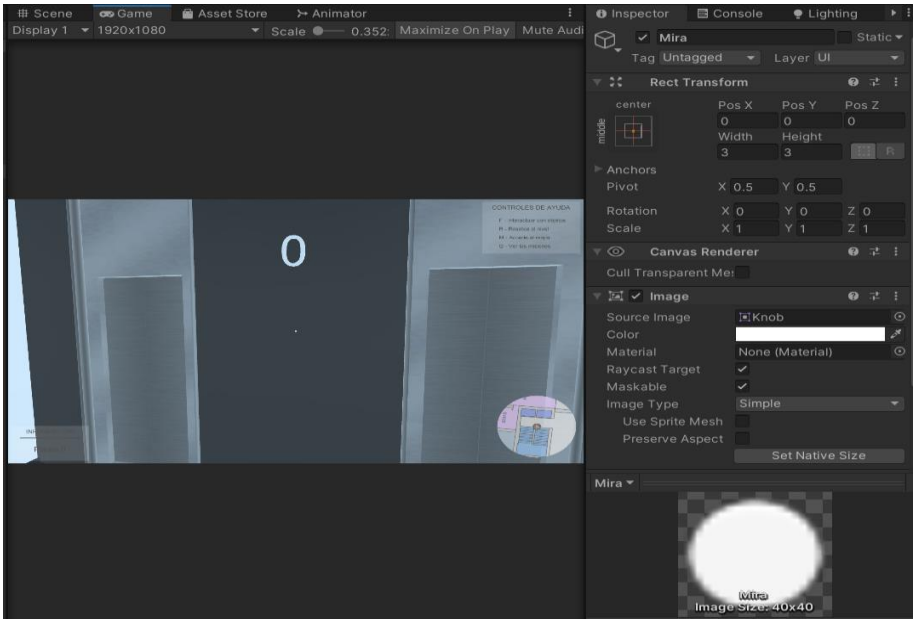


Fig. 71. Mira empleada en el Panel Game

Por último, en el cuadrado azul disponemos de un mini mapa que nos muestra la situación del usuario en el mapa. El concepto base para la realización de este mini mapa es crear una textura de renderizado de una cámara con proyección ortográfica en posición cenital al Player. Vamos a ver paso a paso su realización.

El primer paso para la realización de este mini mapa es añadir un game object vacío dentro de Player, lo llamaremos *PlayerMiniMap*. Seguidamente añadiremos dentro un game object 2D de tipo sprite, lo posicionaremos siguiendo el sentido que tiene la cámara y a una altura superior al edificio. Al estar dentro del objeto Player conseguimos que cuando el personaje se mueva el sprite se mueva con nosotros.

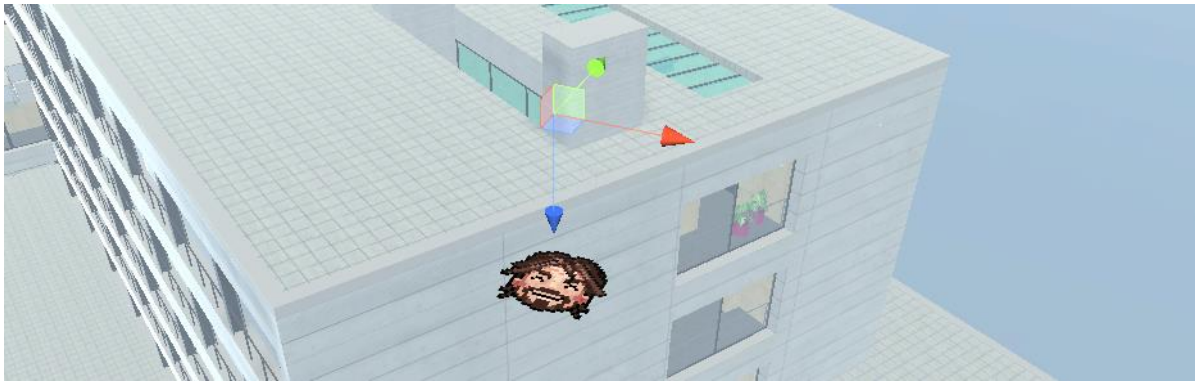


Fig. 72. Sprite del Player

En el segundo paso crearemos un *Render Texture,* para ello desde la carpeta de texturas accedemos a *Create / Render Texture*.

El siguiente paso será añadir un objeto cámara al PlayerMiniMap, está la posicionaremos encima del sprite apuntando en su dirección. Indicaremos que se trata de una cámara de proyección ortográfica con un tamaño de siete, obteniendo así una cámara cenital del sprite válida para nuestro mini mapa.

Vincularemos la textura creada anteriormente en el campo *Target Texture.* Seguidamente añadiremos un *Culling Mask*, esto permite que la cámara sólo pueda ver elementos de una layer concreta, crearemos una con el nombre de Mapa a la que le asignaremos el sprite.

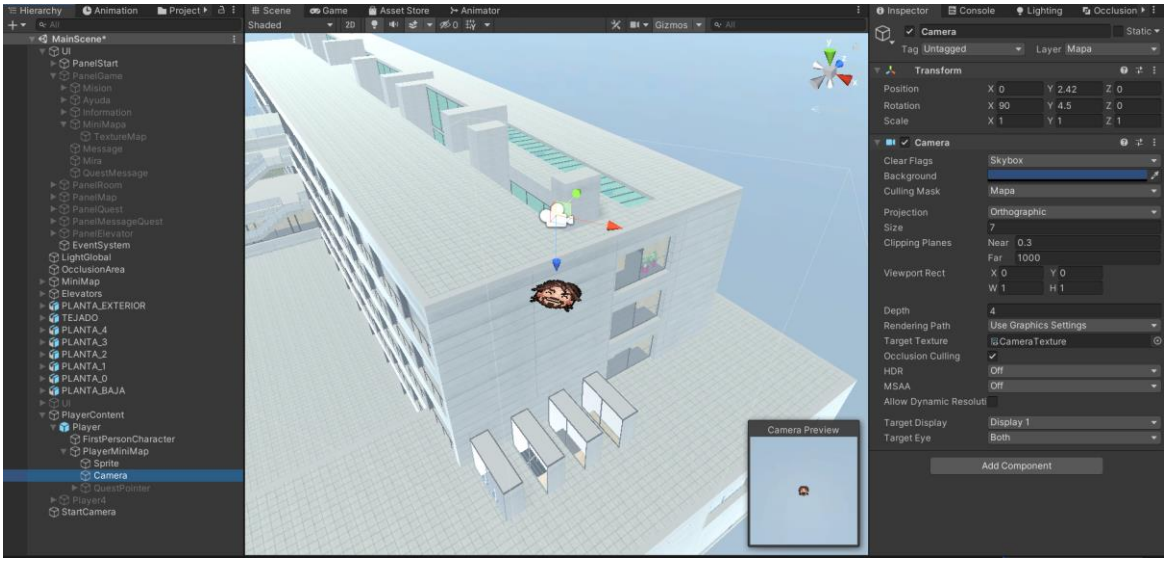


Fig. 73. Cámara del mini mapa

El siguiente paso será crear los elementos del UI, para ello seguiremos el mismo proceso que con las imágenes de *PanelStart*, crearemos una primera capa que servirá de recorte, usaremos el mismo sprite con forma circular. Posteriormente crearemos un elemento hijo de tipo *Raw Image,* en el cual como textura le añadiremos la textura creada.

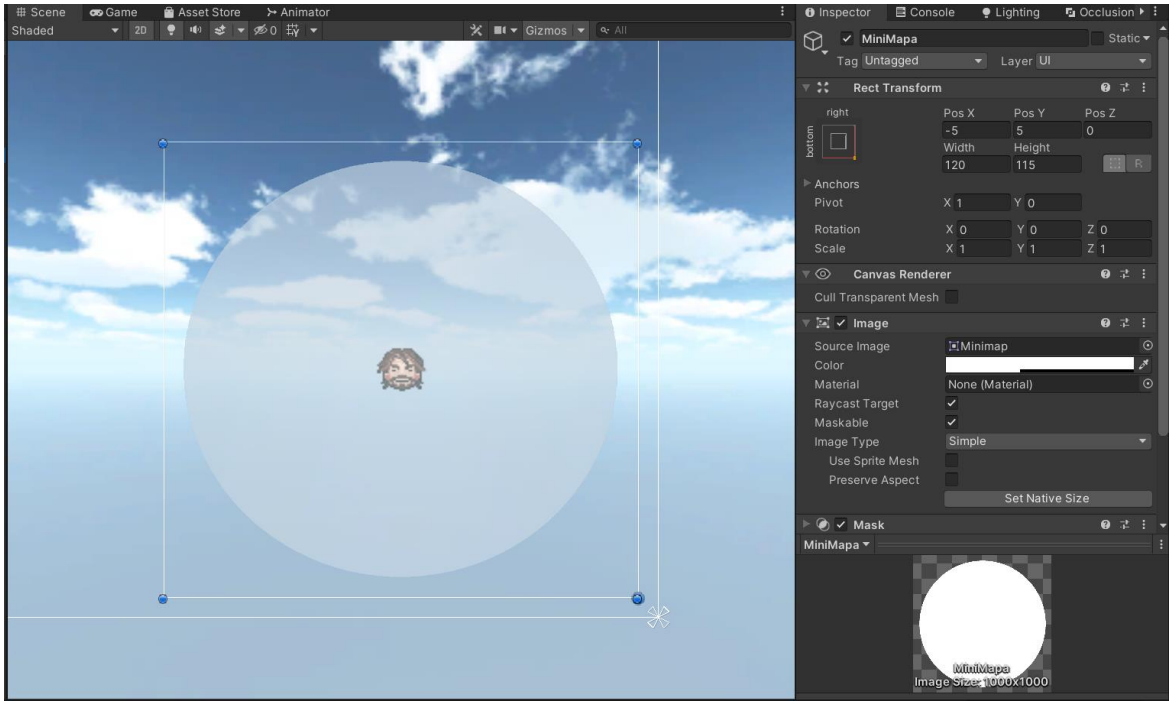


Fig. 74. Añadiendo el render texture al UI

Debido a que sólo hemos indicado que la cámara visualice elemento de la layer Mapa, el mini mapa sólo muestra el sprite. Para solucionarlo vamos a cargar los sprites de las distintas plantas. Partiendo de los mapas del TFG, crearemos con ellos imágenes PNG de los mapas. Las importaremos al proyecto como Sprites dentro de un game object padre denominado *MiniMap*. Una vez importadas las posicionaremos correctamente, para ello nos ayudaremos de los elementos de posicionamiento de UNITY ayudándonos de la ubicación del modelado del edificio. Para que aparezcan en el mini mapa las añadiremos a la layer Mapa.

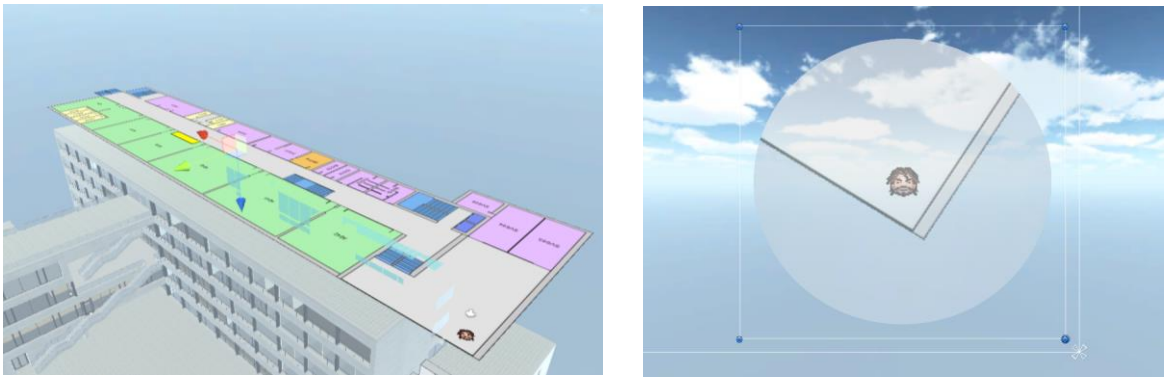


Fig. 75. Añadiendo los sprites de mapas al mini mapa

Con esto disponemos ya de un mini mapa que sigue nuestra orientación y posición, sin embargo, existen dos problemas:

- No se visualizan correctamente el mapa correspondiente al piso actual.
- El sprite de player se mueve con el Player en todas direcciones, esto hace que se desplace con él en altura.

Resolveremos todos estos problemas mediante programación, la cual veremos en EXTENSIÓN MAP, donde crearemos un script que detecte cuando cambiamos de planta y otro que no permita el movimiento en un eje, por ello los marcaremos con *enable* a falso para que no sean visibles.

Para finalizar el mini mapa, añadiremos dos sprites más, estos son para mostrar información de la misión en curso en el mini mapa. El primero lo añadiremos entre las capas de los mapas, este marcará la posición final de la misión. El segundo se pondrá al mismo nivel que el sprite del player, este será un indicador que señalará la dirección al punto final de la misión. Tanto su comportamiento como su visibilidad serán programadas mediante scripts, los cuales veremos en EXTENSIÓN QUEST, los marcaremos con enable a false.

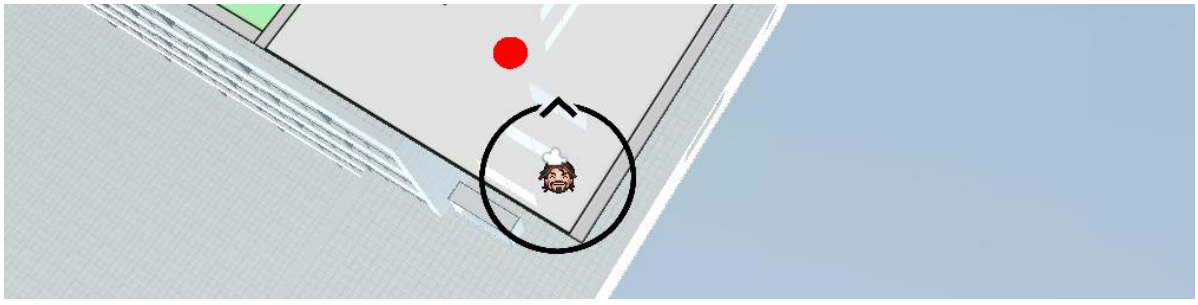


Fig. 76. Añadiendo los sprites de misión al mini mapa

3.6.3. PANEL ROOM

Este panel va a disponer de toda la información de los distintos espacios de interés de la escuela. Tal como pudimos ver en el modelo de datos, disponemos de cuatro tipos diferentes de información: Aulas, despachos, seminarios y servicios. Es por ello que esta interfaz dispondrá de cuatro versiones diferentes en función de la información a mostrar. No obstante, comparten estilo de maquetación, para este panel se ha optado por simular una pizarra de la escuela, donde los elementos se han diseñado mediante paneles.

Cabe destacar que la información de las estancias de la Fundación y DICGF, será modelada igual que las de las estancias de servicios, ya que disponen del mismo modelo de datos.

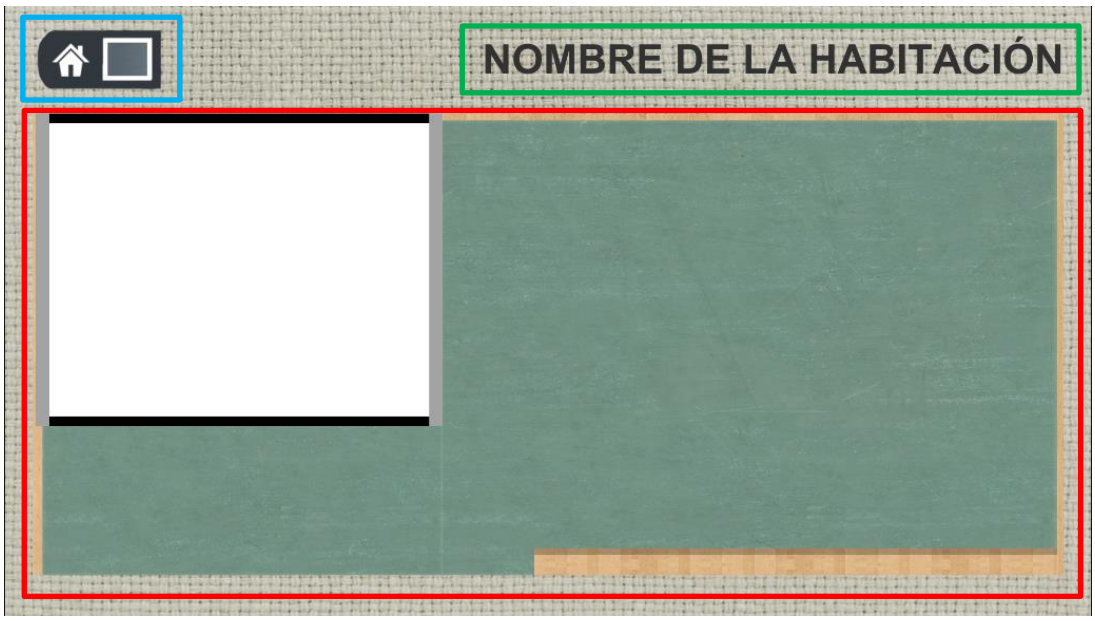


Fig. 77. Diseño común del Panel Room

En el cuadrado verde del diseño, vemos un texto mock, este se rellenará correctamente mediante programación en función de la habitación consultada.

En el cuadrado azul aparece el botón que permitirá regresar al modelo interactivo y poder continuar explorándolo. Se trata de un botón sin texto cuyo sprite fue diseñado simulando los botones del ascensor de la escuela.

En el cuadrado rojo se dispone la zona donde se mostrará la información, es aquí donde la disposición de elementos será diferente en función de la tipología de la habitación.

Se ha empleado texto mock para el diseño de las pantallas, empleando los textos más largos de cada tipo de campo a mostrar, asegurando así una visualización correcta de los campos. Los valores reales vendrán obtenidos por programación, la cual veremos en EXTENSION ROOM.

Para la realización de los distintos tipos se han estructurado los datos empleando la siguiente jerarquía. Se subdivido el interfaz en dos grandes paneles. El superior contiene el botón y el nombre de la instancia y el inferior la información. Dentro del inferior disponemos de un panel con el diseño común (pizarra y proyector) y panel por cada tipología de habitación.

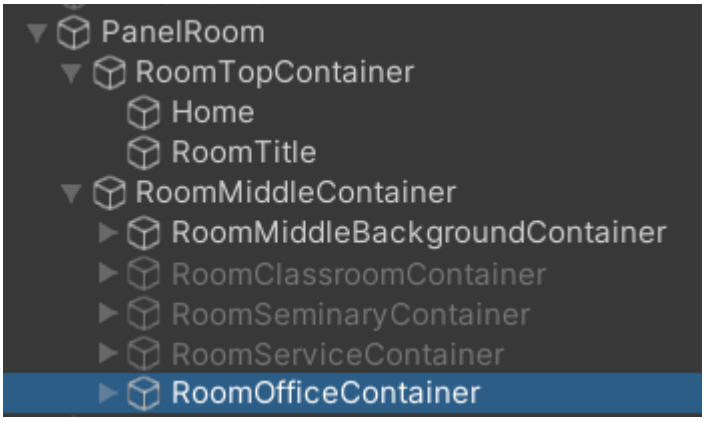


Fig. 78. Estructura del Panel Room

ROOM OFFICE CONTAINER

Este panel dispone de la información de los despachos de los profesores, muestra información cruzada de las tablas PROFESORES y DESPACHOS.

Fig. 79. Diseño de despachos del Panel Room

En el cuadrado rojo, se muestra la información del profesor asociado al despacho. Se han empleado paneles y textos para su diseño.

En el cuadrado azul, aparece la información del despacho. Se han empleado paneles y textos para su diseño.

En el cuadrado naranja, se muestra un botón que redirige al navegador web, para abrir la web de la UPV con más información del profesor.

En el cuadrado verde, aparece la imagen del profesor, obtenidas de la web de la UPV, en caso de no disponer de una muestra la imagen por defecto que se ve en el diseño. Se obtendrá de los resources de la aplicación.

En el cuadrado morado, se muestra el logo del departamento asociado al profesor, en caso de no tener ninguno asociado se muestra el logo de la UPV. Se obtendrá de los resources de la aplicación

ROOM SERVICE CONTAINER

Este panel dispone de la información de los servicios, fundación o DICGF, muestra información de las tablas SERVICIOS, FUNDACIÓN y DICGF.

Fig. 80. Diseño de servicios del Panel Room

En los cuadrados rojos, disponemos de la información referente al servicio mostrado.

En el cuadrado azul, se muestra un botón que redirige al navegador web, para abrir la web de la UPV con más información de ese servicio, en caso de no tener web no aparecerá.

En el cuadrado verde, se muestra el logo de la UPV, en caso de tratarse de un servicio. Por el contrario, en caso de ser una habitación de la Fundación o de DICGF mostrará su logo. Se obtendrá de los resources de la aplicación

ROOM SERVICE CONTAINER

Este panel dispone de la información de los seminarios, muestra información de la tabla SEMINARIOS.

Fig. 81. Diseño de seminarios del Panel Room

En los cuadrados rojos, disponemos de la información referente al seminario mostrado.

En el cuadrado verde, se muestra una fotografía del seminario, en caso de no tener ninguna se muestra la imagen por defecto que se ve en el diseño. Se obtendrá de los resources de la aplicación.

En el cuadrado azul, se muestra el logo del departamento, en caso de no tener ninguno asociado se muestra el logo de la UPV. Se obtendrá de los resources de la aplicación.

ROOM CLASSROOM CONTAINER

Este panel dispone de la información de las aulas, muestra información de la tabla AULAS. Este panel dispone de dos diseños, el primero es el diseño estándar de datos de información de un aula.

Fig. 82. Diseño de aulas del Panel Room

La segunda muestra la información del horario del aula.

Fig. 83. Diseño de horario de aulas del Panel Room

En los cuadrados rojos, disponemos de la información referente al aula mostrada.

En el cuadrado verde, se muestra una fotografía del aula, en caso de no tener ninguna se muestra la imagen por defecto que se ve en el diseño. Se obtendrá de los resources de la aplicación.

En el cuadrado azul, se muestra el botón que permite mostrar el horario del aula, el cual se puede observar en la segunda imagen, al seleccionar cambia la label del botón. En caso de que el aula no disponga de horario no se mostrará el botón.

En el cuadrado amarillo, se muestran los botones que permite mostrar el horario de un cuatrimestre o de otro. Al cambiar se queda marcado en amarillo el seleccionado.

En el cuadrado naranja podemos observar el horario del aula, este cambiara en función del cuatrimestre y aula mostrada. Se obtendrá de los resources de la aplicación.

3.6.4. PANEL MAP

Este panel va a disponer de los mapas de la escuela, donde mostraremos las distintas plantas del edificio, desde la planta baja hasta la cuarta planta, la zona del aparcamiento no se mostrará ya que no contiene información de interés. El diseño es igual para todas las plantas, tan solo cambiará la zona que contiene el mapa, donde se mostrara inicialmente en función de la posición de usuario y posteriormente podremos ir cambiando entre las distintas plantas del edificio.

Debido a que el diseño es el mismo para todas las plantas, veámoslo para la planta baja.

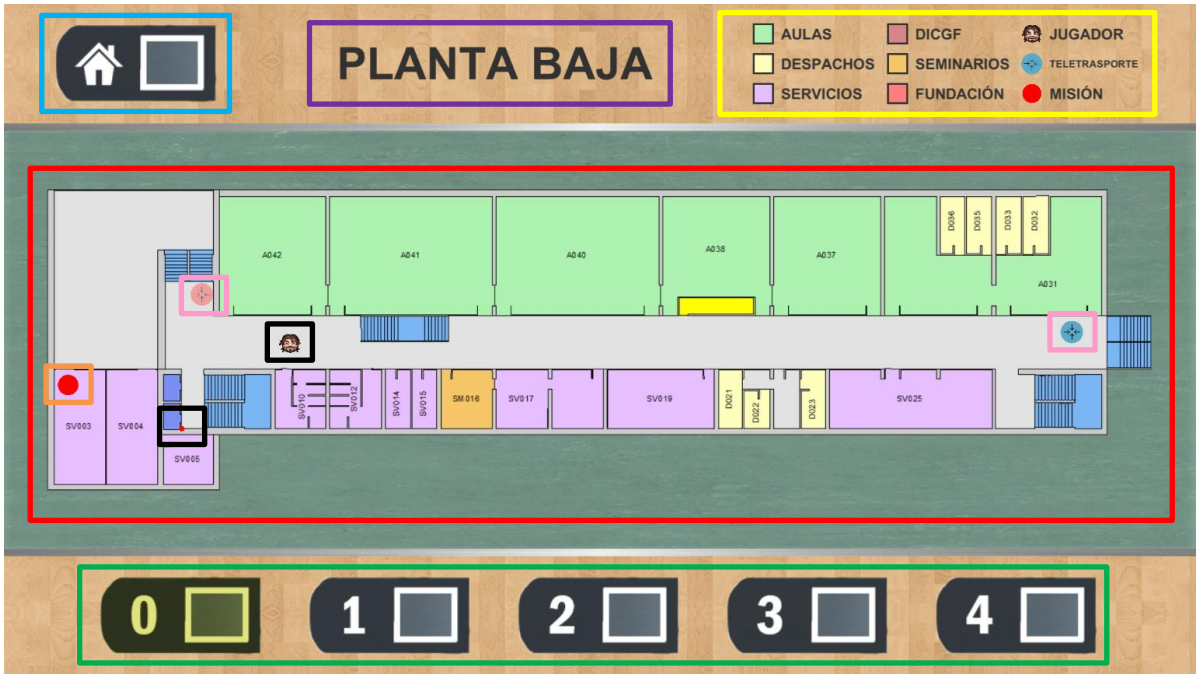


Fig. 84. Diseño del Panel Map

El diseño se basa en una pizarra de la escuela, donde podemos ver el mapa dibujado en ella, los elementos interactivos se encuentran sobre la pared, dichos elementos han sido diseñados mediante paneles. Todas las funcionalidades de los elementos descritos en esta sección, serán programadas mediante scripts C#, los cuáles veremos en EXTENSIÓN MAP.

En el cuadrado azul, aparece el botón que permitirá regresar al modelo interactivo y poder continuar explorándolo. Sigue el diseño aplicado en el *PanelRoom*.

En el cuadrado morado, vemos un texto que indica la planta mostrada, este se rellenará correctamente mediante programación en función de la planta consultada.

En el cuadrado amarillo, aparece una pequeña leyenda de los elementos importantes que aparecen en el mapa. Para su creación se han empleado paneles, imágenes y textos.

En el cuadrado verde, disponemos de los botones que permitirán cambiar entre las distintas plantas. Cuando una de las plantas es activada, este modifica su color ligeramente para indicar que está activo. Sigue el diseño aplicado en los anteriores botones.

En el cuadrado rojo, tenemos el mapa consultado, debido a que no todas las plantas disponen del mismo tamaño en algunas de ellas se verá algo más pequeño.

En el cuadrado naranja, vemos indicado el punto final de la misión, su posición y visualización será controlada por programación, la cual veremos en EXTENSIÓN QUEST.

En los cuadrados rosas, disponemos de los puntos de teletransporte, estos corresponden con las localizaciones de los directorios de la escuela. Estos puntos sólo estarán disponibles si se accede al mapa mediante dichos directorios, el directorio por el que se accedió queda marcado en rojo y deshabilitado, impidiendo así poder moverse al directorio actual.

En los cuadrados negros, tenemos al personaje y el eje de coordenadas que permite posicionarlo correctamente en el mapa. Debido que debemos mostrar al personaje en el mapa se ha de triangular su posición, para ello se han empleado los puntos de teletransporte, de los cuales disponemos de sus coordenadas *inGame* e *inMap*. El punto de coordenadas no lo mostraremos, siendo este sólo algo interno para nuestros cálculos.

3.6.5. PANEL QUEST

Este panel va disponer de las misiones disponibles en la aplicación, aquí encontraremos un listado de misiones disponibles para realizar. La información de las misiones se obtendrá de la base de datos mediante programación, esto se verá con más detenimiento en EXTENSION QUEST.

En cuanto al diseño se ha intentado simular un corcho de los que podemos ver por la escuela, mostrando en uno de ellos las posibles misiones y en otro la información de la misión seleccionada. Para su diseño se han empleado paneles, botones y textos.

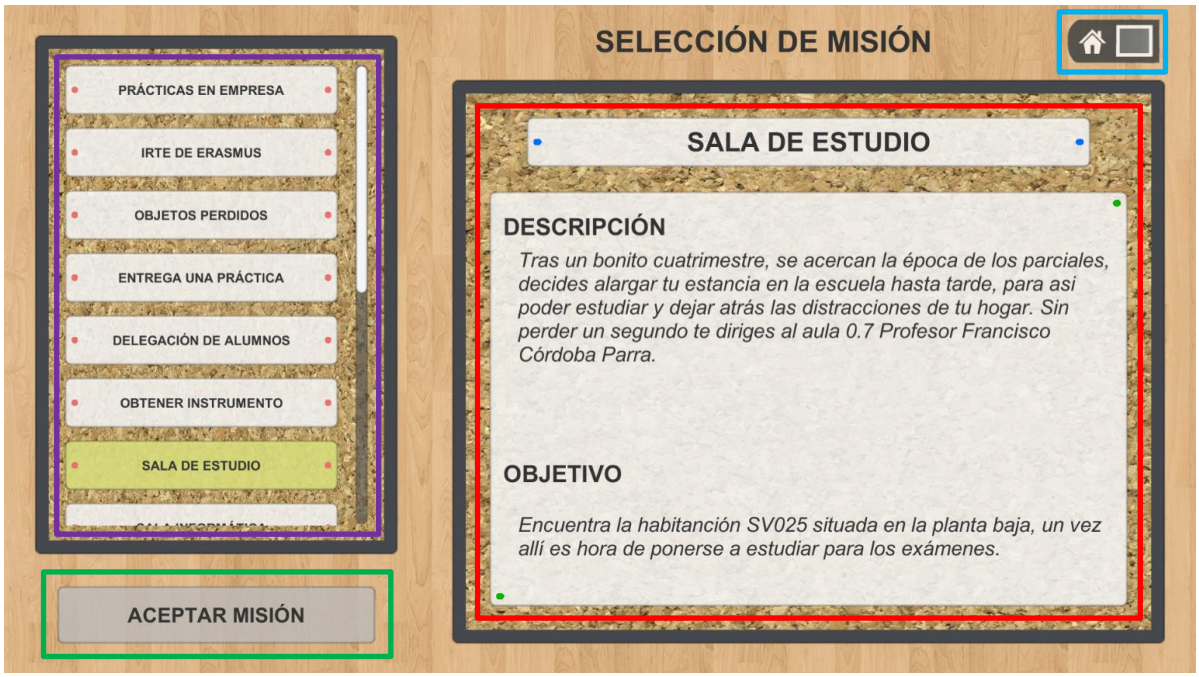


Fig. 85. Diseño del Panel Quest

En la el cuadrado rojo, encontramos la información referente a la misión seleccionada, donde encontramos el nombre, descripción y objetivo de la misión

En el cuadrado verde, disponemos del botón que permitirá aceptar la misión seleccionada. Para este botón se ha optado por un diseño más convencional.

En el cuadrado azul, aparece el botón que permitirá regresar al modelo interactivo y poder continuar explorándolo. Sigue el diseño aplicado en los anteriores paneles.

En el cuadrado morado, podemos encontrar el listado de misiones libres que puede realizar el usuario, cada una de ellas se trata de un botón que permite seleccionar la misión y mostrar su información. Se debe destacar que se ha implementado una *Scrollbar* para poder mostrar correctamente todas las misiones. Veamos con más detenimiento su estructura.

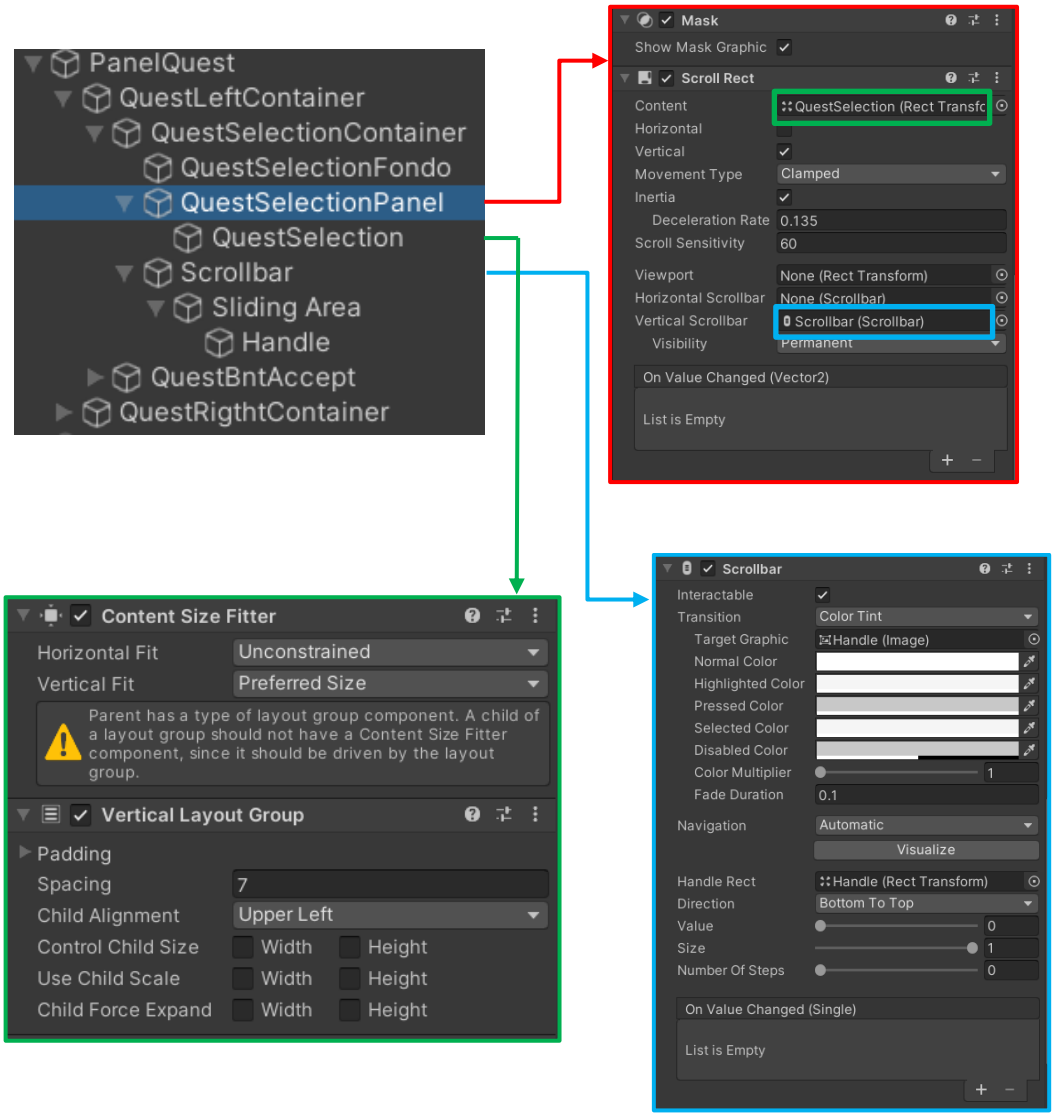


Fig. 86. Añadiendo una Scrollbar al Panel Quest

Como se ilustra en la imagen superior, se ha añadido un objeto con un componente de tipo *Scroll Rect*, este será el objeto padre que contendrá la zona con scroll. A este se le asocia otro objeto que dispone de componente *Content Size Fitter*, el cual será el que contendrá todos los botones de las distintas misiones. Finalmente se crea otro game object con el componente *Scrollbar* y se vincula al *Scroll Rect*, de tal forma que permita controlar el desplazamiento en ese panel. Gracias a esta barra de desplazamiento, podemos disponer de gran cantidad de misiones sin necesidad de disponer de un gran tamaño en la pantalla UI, consiguiendo así la posibilidad de ampliar la cantidad de misiones sin necesidad de modificar el UI en el futuro.

3.6.6. PANEL MESSAGE QUEST

El Panel Message Quest es un panel de ayuda, el cual muestra al usuario indicaciones para finalizar la misión en curso. Este panel aparece cuando el usuario llega al piso donde se encuentra el punto final de la misión.

Fig. 87. Diseño del Panel Message Quest

El diseño cuenta con textos estáticos, paneles y un botón. Este botón permite al usuario continuar explorando el modelo y poder llegar así al punto final de la misión en curso.

3.6.7. PANEL ELEVATOR

Este panel se emplea para activar el desplazamiento del ascensor entre los diferentes pisos del edificio. Para añadirle una funcionalidad extra, al seleccionar una planta muestra las aulas, servicios, despachos y seminarios existentes, esta información se obtiene de la base de datos. Una vez seleccionado el piso permite cerrar las puertas para iniciar así el desplazamiento del ascensor. Todas estas funcionalidades han sido programadas mediante scripts, los cuales podremos ver en EXTENSIÓN ELEVATOR.

En este apartado nos centraremos en el diseño de la pantalla, donde se ha intentado simular una pantalla que nos recuerde al propio ascensor del edificio, para su diseño se han empleado botones, paneles y textos. En el apartado de servicios, aparece también la información de las habitaciones pertenecientes a las tablas Fundación y DICGF, ya que disponen del mismo tipo de datos y permite ahorrar espacio en la visualización de la interfaz UI

Fig. 88. Diseño del Panel Elevator

En la el cuadrado rojo, encontramos la información referente al piso seleccionado, donde encontramos información de aulas, despachos, seminarios y servicios. Para ellos muestra su código y su nombre (Salvo para los despachos, donde muestra los apellidos y el nombre).

En el cuadrado verde, disponemos de los botones que permitirán seleccionar el piso al que queremos desplazarnos, al seleccionarlo nos mostrará la información asociada, al seleccionar uno este se queda en color rojo para diferenciarlo de los demás. Se ha empleado el estilo de botones ya empleado anteriormente.

En el cuadrado azul, aparece el botón que permitirá regresar al modelo interactivo y poder continuar explorándolo. Sigue el diseño aplicado en los anteriores paneles.

En el cuadrado amarillo, aparece el botón que permitirá activar el desplazamiento a la planta seleccionada, sacando al usuario de la pantalla UI y activando el movimiento del ascensor.

4. *IMPLEMETACIÓN DE NUEVAS FUNCIONALIDADES EN UNITY*

En esta etapa del proyecto vamos a darle funcionalidades propias al modelo de UNITY, consiguiendo que este se convierta en un modelo interactivo. UNITY ofrece la posibilidad de programar en dos lenguajes de programación: C# y UnityScript, debido a que C# es más potente, proporciona mejor rendimiento y está más extendido.

Se han dividido las clases C# en distintas extensiones, cada una definiendo una funcionalidad nueva del modelo. Encontramos las siguientes extensiones:

- **CORE:** Encargada de aspectos globales del proyecto.
- **DOORS:** Da funcionalidad a las distintas puertas del proyecto.

- **ROOM:** Permite la visualización de la información de las distintas estancias del edificio.
- **MAP:** Nos proporciona la visualización de un mapa, un mini mapa y la función de teletransporte en el edificio
- **ELEVATOR:** Da funcionalidad a los ascensores del edificio.
- **QUEST:** Encargada de gestionar la realización de misiones.

Dentro de las extensiones tenemos distintos paquetes, diferenciados por la finalidad de cada clase. Disponemos de los siguientes paquetes:

- **MANAGER**: Clases estáticas que disponen de variables y métodos útiles. Son clases globales que no se instancian dentro de ningún game object.
- **DTO**: Clases estáticas que definen elementos de la base de datos (Aula, profesor…).
- **ENUM**: Clases estáticas que definen enumerados que se emplearán en otras clases.
- **CONTROLLER**: Clases asociadas al game object del PLAYER otorgando nuevas funcionalidades.
- **GAME**: Clases que se asocian a game objects dándole nuevas funcionalidades.
- **UI**: Clases que se asocian a elementos de UI, para dotarle de funcionalidad.

4.1. EXTENSIÓN CORE

En ese apartado vamos a ver la extensión CORE, encargada de aspectos globales del proyecto.

4.1.1. GAME POSITION MANAGER

Encargada de disponer de variables y métodos necesarios para el control del posicionamiento del personaje. Veamos su código y diferenciemos las distintas partes importantes:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public static class GamePositionManager {
    //CONSTANTS
    private static float START_X_COORDINATE = -6509.01f;
    private static float START_Y_COORDINATE = 11426.55f;
    private static float START_Z_COORDINATE = -15363.26f;
    private static float START_ALPHA_EULER_ANGLE = 0.0f;
    private static float START_BETA_EULER_ANGLE = -18.21f;
    private static float START_GAMMA_EULER_ANGLE = 0.0f;

    //VARIABLES
    private static Vector3 playerCoordinates = getStartCoordinates();
    private static Vector3 playerOrientation = getStartOrientation();

    private static bool inGame;

    public static void setPlayerPosition(Transform transform) {
        setPlayerCoordinates(transform.position);
        setPlayerOrientation(transform.eulerAngles);
    }
```

```
public static Vector3 getStartCoordinates() {
    return new Vector3(START_X_COORDINATE, START_Y_COORDINATE,
     START_Z_COORDINATE);
}

public static Vector3 getStartOrientation() {
    return new Vector3(START_ALPHA_EULER_ANGLE, START_BETA_EULER_ANGLE,
     START_GAMMA_EULER_ANGLE);
}
```

```
public static Vector3 getPlayerCoordinates() {
    return playerCoordinates;
}

public static void setPlayerCoordinates(Vector3 coordinates) {
    playerCoordinates = coordinates;
}

public static Vector3 getPlayerOrientation() {
    return playerOrientation;
}

public static void setPlayerOrientation(Vector3 orientation) {
    playerOrientation = orientation;
}
```

```
public static bool isInGame() {
    return inGame;
}

public static void setInGame(bool game) {
    inGame = game;
}
```

```
}
```

En los cuadrados rojos, podemos ver las variables y métodos que definen y devuelven la posición inicial del jugador, en UNITY un objeto viene posicionado por su Vector3 de posición (Vector 3D: X, Y, Z) y su Vector3 de orientación (Vector de ángulos Euler: α, β y γ). La posición seleccionada se encuentra en el recibidor de la escuela.

En los cuadros verdes, se observan las variables que indican la posición del jugador. Vemos que inicialmente se inicializan con el valor de las coordenadas y orientación de iniciales. Existen varios métodos para interactuar con estas posiciones, tanto para obtenerlas como para modificarlas. En UNITY los game objects tienen un tipo de dato denominado *Transform*, este dispone tanto la posición como la orientación del objeto, por ello se creó un método que permita actualizarlo a partir de este dato.

En los cuadros azules, disponemos de las variables y métodos que permiten saber si el jugador se encuentra en una pantalla UI (pantalla inicial, pantalla de información…) o bien está moviéndose libremente por el modelo, esto nos permitirá restringir ciertas acciones a cada escenario.

4.1.2. GAME DATABASE MANAGER

Esta clase nos ofrece todos los métodos necesarios para interactuar con la base de datos, es la encargada de orquestar las llamadas a *DatabaseGame* y confeccionar las QUERIES.

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public static class GameDatabaseManager {

    private static string DATABASE_NAME = "ETSIGCT.db";
    private static DatabaseGame db = new DatabaseGame();
```

```
    public static string obtainField(string query) {
        openDatabase();
        string result = db.obtainField(query);
        closeDatabase();
        return result;
    }

    public static List<string> obtainSingleElement(string query) {
        openDatabase();
        List<string> result = db.obtainSingleElement(query);
        closeDatabase();
        return result;
    }

    public static List<List<string>> obtainMultipleElements(string query) {
        openDatabase();
        List<List<string>> result = db.obtainMultipleElements(query);
        closeDatabase();
        return result;
    }
```

```
    public static string obtainBasicQuery(string field, string table, string
condition) {
        string result = "SELECT " + field + " FROM " + table;
        return condition == null ? result : result + " WHERE " + condition;
    }


    public static string obtainBasicQuery(List<string> fieldList, string table,
string condition) {
        return obtainBasicQuery(obtainFields(fieldList), table, condition);
    }


    public static string obtainJoinQuery(string field, string mainTable, string
mainField, string joinTable, string joinField, string condition) {
        string result = "SELECT " + field + " FROM " + mainTable + " M, " + joinTable
+ " J WHERE M." + mainField + " = J." + joinField;
        return condition == null ? result : result + " AND " + condition;
    }
```

```
    public static string obtainJoinQuery(List<string> mainFieldList, string
mainTable, string mainField, List<string> joinFieldList, string joinTable, string
joinField, string condition) {
        string fields = obtainFields(addPrefixToFieldList(mainFieldList, "M"),
addPrefixToFieldList(joinFieldList, "J"));
        return obtainJoinQuery(fields, mainTable, mainField, joinTable, joinField,
condition);
    }
```

```
    private static void openDatabase() {
        db.dbOpen(DATABASE_NAME);
    }

    private static void closeDatabase() {
        db.dbClose();
    }
```

```
    private static string obtainFields(List<string> fieldList) {
        return String.Join(", ", fieldList.ToArray());
    }

    private static List<string> addPrefixToFieldList(List<string> fieldList, string
prefix) {
        List<string> newFieldList = new List<string>();
        foreach (string field in fieldList) {
            newFieldList.Add(String.Format("{0}.{1}", prefix, field));
        }
        return newFieldList;
    }

    private static string obtainFields(List<string> mainFieldList, List<string>
joinFieldList) {
        mainFieldList.AddRange(joinFieldList);
        return obtainFields(mainFieldList);
    }

}
```

En el cuadrado rojo, tenemos las variables de la clase, en ella disponemos del nombre de la base de datos y del objeto de tipo *DatabaseGame*, clase que hemos explicado anteriormente, encargada de hacer las peticiones y conexiones a la base de datos. Ambas son privadas para sólo tener acceso a ellas desde la propia clase.

En el cuadrado verde, podemos ver los métodos que abren y cierran la conexión a la base de datos, estos métodos son privados teniendo sólo acceso a ellos a través en la clase.

En el cuadrado naranja, disponemos de varios métodos privados que proporcionan utilidades para la creación de las queries. El primer método *obtainFields,* permite transformar una lista de campos a obtener en la query, a un string con la nomenclatura correcta para después de un SELECT. El segundo método *obtainFields,* otorga el mismo resultado, pero teniendo en cuenta dos listas, es un método pensado en la generación de una query de tipo JOIN donde tenemos campos de dos tablas. Por último, el método *addPrefixToFieldList*, permite añadir un prefijo delante de cada campo para referenciar la tabla a la que hace referencia, está también pensado para queries de tipo JOIN donde debemos referenciar cada campo a su tabla.

En el cuadrado morado, disponemos de los métodos empleados para obtener valores de la base de datos, hay uno para cada tipo de acción creada en *DatabaseGame*, obtener un campo, un registro o varios registros. La estructura del método es igual en todos ellos, primero abrimos la conexión a la base de datos, seguidamente llamamos al método en cuestión que devuelve los resultados, posteriormente cerramos la conexión y por último devolvemos el resultado. Todos estos métodos son públicos y serán los que dispondrán las otras clases para acceder a la base de datos.

En los cuadrados azules, disponemos de los métodos que nos permiten generar las queries, para posteriormente llamar con ellas a las funciones del cuadrado morado. Estos métodos se apoyan en las funciones del cuadrado naranja para componer partes de la query. Disponemos de cuatro métodos, dos para queries a una única tabla y otras dos para JOINS entre dos tablas. Estos métodos son públicos, ya que serán los que empleen otras clases para componer sus queries.

4.1.3. PLAYER CONTROLLER

En esta clase vamos a definir una serie de comportamientos referentes a player, vamos a definir métodos que modifican la posición del jugador, así como acciones que puede realizar.

```
using UnityEngine;

public class PlayerController : MonoBehaviour {

    public GameObject startPanel;

    void Start () {
        setPosition(GamePositionManager.getPlayerCoordinates(),
GamePositionManager.getPlayerOrientation());
    }

    void Update () {
        if (Input.GetKey("escape") && GamePositionManager.isInGame())
            startPanel.SetActive(true);

        else if (Input.GetKey("r") && GamePositionManager.isInGame())
            setPosition(GamePositionManager.getStartCoordinates(),
GamePositionManager.getStartOrientation());
    }

    public void showCursor() {
        Cursor.visible = true;
        GamePositionManager.setInGame(false);
    }

    public void hideCursor() {
        Cursor.visible = false;
        GamePositionManager.setInGame(true);
    }

    public void moveToPosition(Vector3 coordinates) {
        transform.position = coordinates;
    }

    private void setPosition(Vector3 coordinates, Vector3 orientation) {
        moveToPosition(coordinates);
        transform.eulerAngles = orientation;
    }
```

En el cuadrado azul, tenemos las variables de la clase, la cual corresponde al panel UI de inicio de la aplicación. Al crear variables públicas UNITY permite asociarlas al script una vez introducido este en el objeto.

En el cuadrado naranja, vemos métodos que permiten posicionar al player en cierta posición y orientación. El método *moveToPosition* es accesible fuera de la clase, permitiendo que otras clases accedan a este, esto lo veremos en el apartado de la extensión de Mapa, más concretamente en la clase *MapSceneInit.*

En el cuadrado rojo, disponemos del método *start*, este es propio de las clases *MonoBehaviour* de UNITY, clases C# que asociaremos a game objects. Este método se ejecuta en el arranque de la aplicación, en nuestro caso, se encarga de posicionar al usuario en las coordenadas definidas en *GamePositionManager*.

En el cuadrado verde, vemos el método *update*, se trata de otro propio de las clases *MonoBehaviour* de UNITY. Este método se ejecuta en cada refresco del juego, lo que permite con él, hacer acciones perdurables en el tiempo, en este caso se han desarrollado dos acciones diferentes. La primera permite que cuando se pulse la tecla *escape*, muestre el panel de la pantalla de inicio, la cual le indicamos como parámetro de entrada en el script. La segunda permite que pulsado la tecla R, el personaje regrese a la posición original, definida en *GamePositionManager*, permitiendo que si el personaje se extravía podamos recolocarlo en la posición original. Cabe destacar que ambas acciones sólo son accesibles sin nos encontramos moviéndonos libremente por el modelo, controlado con la variable *inGame.*

En el cuadrado morado, disponemos de métodos que permiten mostrar el cursor, para poder interactuar con elementos en las pantallas de interfaz, o bien ocultarlo, para poder movernos libremente por el modelo. Estos métodos actualizan el estado de la variable *inGame* de la clase *GamePositionManager*.

Una vez creado la clase se la asignamos al player, para ello accedemos al game object denominado FPSController, el cual hemos renombrado a Player para gestionarlo mejor. Dentro del inspector del game object accedemos a Add Component y buscamos el nombre de nuestra clase (*PlayerController*), se lo asignamos. Una vez añadido vemos que nos permite introducir la variable pública definida en el script.

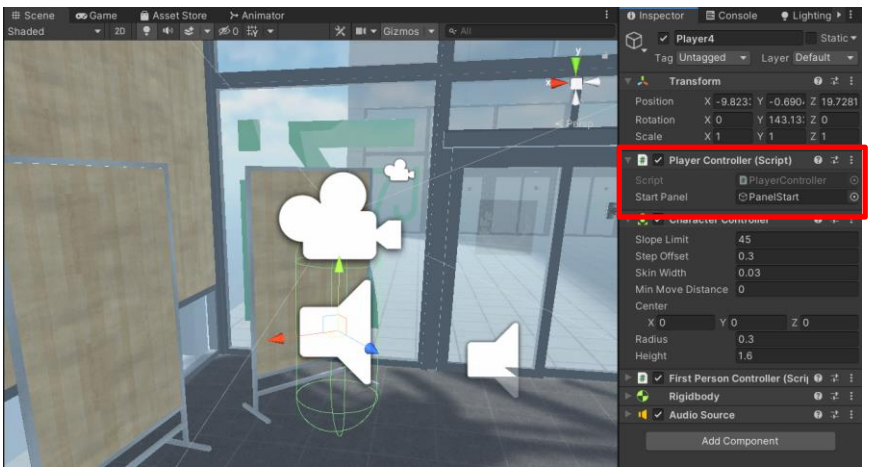


Fig. 89. Añadiendo el script PlayerController

4.1.4. START PANEL

En esta clase vamos a definir el comportamiento para el Start Panel. Tal como vimos anteriormente este panel es el encargado de iniciar, cerrar y mostrar información de la aplicación, se va a crear un script que permita modelar estos comportamientos.

```
using UnityEngine;
using UnityEngine.UI;

public class StartPanel : MonoBehaviour {
    public GameObject startCamera;
    public GameObject gamePanel;
    public GameObject creditsPanel;
    public GameObject player;
    public GameObject startButtonLabel;

    private AudioSource audioSource;
    private bool creditsShowed;

    private static string LINKEDIN_URL = "https://www.linkedin.com/in/can4rio";

    void Start() {
        audioSource = transform.GetComponent<AudioSource>();
    }
    void OnEnable() {
        audioSource.Play();
        stopTime(true);
        player.SetActive(false);
        gamePanel.SetActive(false);
        startCamera.SetActive(true);
        creditsShowed = false;
        changeCreditsPanelVisibility();
        player.GetComponent<PlayerController>().showCursor();
    }
    public void startGame() {
        audioSource.Pause();
        player.GetComponent<PlayerController>().hideCursor();
        startCamera.SetActive(false);
        gameObject.SetActive(false);
        startButtonLabel.GetComponent<Text>().text = "Continuar";
        player.SetActive(true);
        gamePanel.SetActive(true);
        stopTime(false);
    }
    public void showCredits() {
        creditsShowed = !creditsShowed;
        changeCreditsPanelVisibility();
    }
    public void exit() {
        Application.Quit();
    }
```

```
    public void openLinkedin() {
        Application.OpenURL(LINKEDIN_URL);
    }

    private void changeCreditsPanelVisibility() {
        creditsPanel.SetActive(creditsShowed);
    }

    private void stopTime(bool stop) {
        Time.timeScale = stop ? 0 : 1;
    }
}
```

En el cuadrado rosa, introducimos las variables públicas de la clase, las cuales podremos configurar desde el inspector, cuando añadamos el script al panel. Las variables son:

- **StartCamera**: Cámara inicial que apunta a la entrada del edificio.
- **GamePanel**: Hace referencia al interfaz UI que muestra la información de interés mientras nos movemos por el modelado.
- **CreditsPanel**: Es el panel que contiene la información de los autores del proyecto.
- **Player**: Game object con el player.
- **StartButtonLabel**: Elemento UI que tiene el texto del botón que inicia la aplicación.

En el cuadrado naranja, se muestran las variables privadas de la clase. En ella vemos un *boolean* que nos permitirá saber el estado de visibilidad del *CreditsPanel*, un string con la información del enlace de LinkedIn del alumno y un *audio source*. Este *audio source*, nos permitirá hacer sonar un hilo musical mientras estemos en esta pantalla. El valor de esta variable se inicializa en el método *Awake* de la clase, obteniéndolo del propio panel. Este método es propio de las clases *MonoBehaviour* de UNITY, el cual se ejecuta nada más arrancar la aplicación.

En el cuadrado marrón, disponemos de dos métodos de privados para la clase. El primero permite modificar la visibilidad del *CreditsPanel* en función de la variable *creditsShowed* de la clase. El segundo modifica el tiempo del juego, donde se para el transcurso del juego con el valor de 0 y se reanuda con el valor de 1, siempre que estemos en una pantalla UI detendremos el tiempo del juego (salvo en el *panelGame*).

En el cuadrado morado, tenemos el método *OnEnable*, este es propio de las clases *MonoBehaviour* de UNITY, clases C# que asociaremos a game objects. Se aplica siempre que el game object pase a estar activo. Este método realizará todos los procesos necesarios para poder emplear el *startPanel* correctamente, veamos el proceso:

I. Activará el hilo musical.
II. Parará el tiempo del juego.
III. Ocultará al *player* y al *gamePanel*
IV. Mostrará la *startCamera*
V. Ocultará el *creditsPanel*.
VI. Mostrará el cursor.

En el cuadrado rojo, tenemos el método que realizará el *startButton*, este realiza todos los procesos necesarios para poder iniciar o continuar con nuestra exploración del modelo, el proceso es el siguiente:

I. Desactivará el hilo musical.
II. Ocultará el cursor, la cámara y el propio panel
III. Cambiará el texto del *startButton* para que sea a partir de ahora "Continuar".
IV. Mostrará el *player* y el *gamePanel*.
V. Reanudará el tiempo del juego.

En el cuadrado azul, disponemos del método que empleará el botón que muestra a los desarrolladores de la aplicación. Este método se encarga de cambiar la variable *creditsShowed* y actualizar la visualización mediante el método de utilidad. Por tanto, el comportamiento del botón será ocultar o mostrar el *creditsPanel*.

En el cuadrado verde, aparece el método que empleara el *exitButton*, donde se llama a un método existente en UNITY que cierra la aplicación.

En el cuadrado amarillo, tenemos el método empleado por el botón del LinkedIn, donde se llamará a un método existente en UNITY que permite abrir una URL. El enlace a abrir será el correspondiente a la variable *LINKEDIN_URL.*

Una vez visto la lógica, vamos a añadir el script al panel. Para ello, accedemos al panel de inicio de la aplicación y le añadimos un nuevo componente de tipo *Start Panel*, donde referenciaremos todas las variables locales. Posteriormente le añadimos otro componente de tipo *Audio Source* y le añadiremos el clip de audio, se ha marcado como *Loop* para que cuando termine reinicie el audio desde el principio.

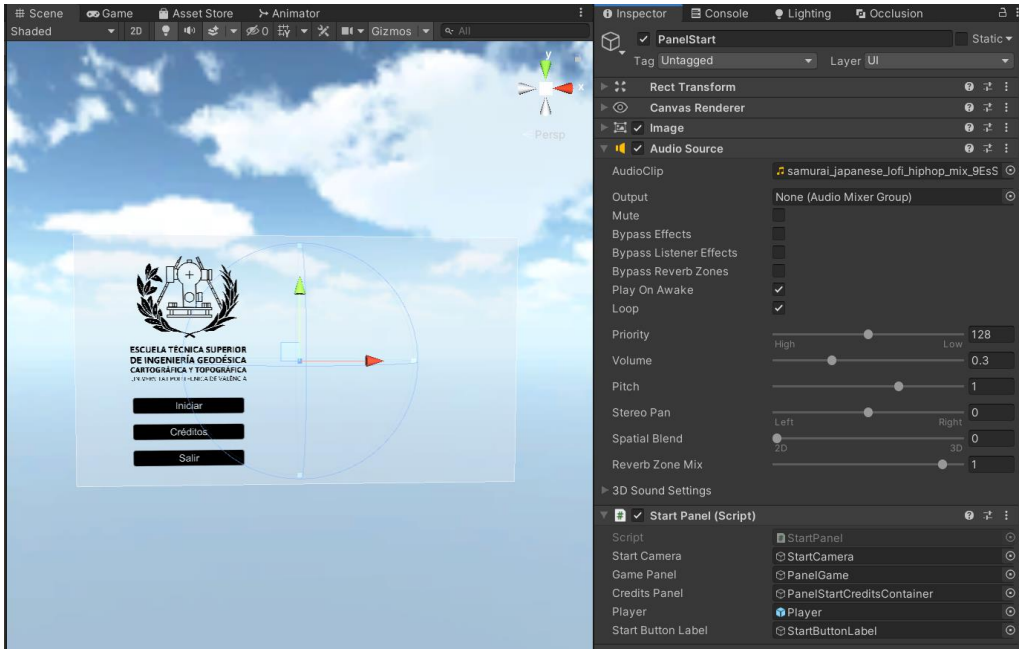


Fig. 90. Añadiendo el script StartPanel

Para finalizar, referenciaremos en el evento *onClick* cada uno de los métodos para cada botón del interfaz UI.

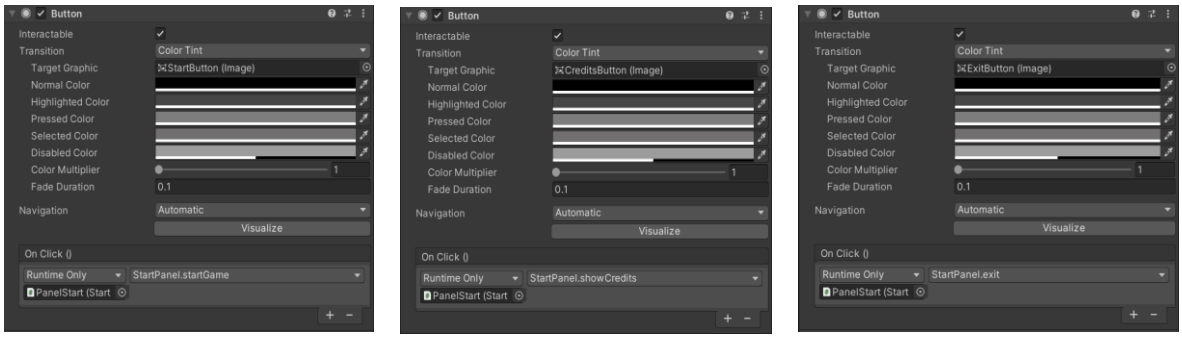


Fig. 91. Referenciando los botones al script StartPanel

4.2. EXTENSIÓN DOORS

En esta sección vamos a modelar el comportamiento del movimiento de las puertas de nuestro modelo interactivo. Dentro del modelo disponemos de tres tipologías de movimientos de puertas:

I. **Puertas de bisagras:** Puertas que se abren desde un lateral, realizando un movimiento de arco, quedando totalmente abiertas en un ángulo de 90 grados.

II. **Puertas correderas:** Puertas con un movimiento horizontal que se desplazan para permitir el acceso.

III. **Puertas giratorias:** Puertas que se desplazan mediante un eje central, permitiéndolas girar y permitiendo mover el hueco disponible para transitar entre dos estancias.

Fig. 92. Tipos de puertas existentes en el modelado

Tanto las puertas de bisagras como las correderas dispondrán de dos versiones. Una automática, que se abre y cierra de manera automática al acercase el player. Otra manual, donde podrá el usuario interactuar con ellas para abrirlas y cerrarlas.

Antes de ver la programación de las puertas vamos a definir una serie de enumerados que se emplearán para la configuración del movimiento de las puertas.

4.2.1. ENUMERADOS DE LA EXTENSIÓN DOORS

DIRECTION

Esta clase define un enumerador que indica el sentido de movimiento de un objeto. Esta nos permitirá en clases posteriores definirla como variable de la clase y aplicar el movimiento en el sentido deseado.

```
public enum Direction {
    Normal,
    Inverse
}
```

AXE DIRECTION

Esta clase define un enumerador que indica la dirección en un eje del espacio 3D, X, Y o Z. Esta nos permitirá en clases posteriores definirla como variable de la clase y aplicar sólo ciertos parámetros de movimiento a dicho eje.

```
public enum AxeDirection {
    X,
    Y,
    Z
}
```

4.2.2. PUERTAS DE BISAGRAS

En este apartado veremos el comportamiento de las puertas de bisagras, tanto las de tipo automático como las de tipo manual.

DOOR OPEN

Este script define el comportamiento de las puertas de bisagra de tipo automático, se trata de las puertas más frecuentes en nuestro modelo. Se optó por el empleo de este tipo de puertas para facilitar al usuario detectar cuáles de las puertas del modelo son accesibles.

```
using System.Collections;
using UnityEngine;

public class DoorOpen : MonoBehaviour {

    public GameObject door;
    public float maxOpenTransition = 90.0f;
    public float degreePerSecond = 35.0f;
    public AxeDirection axe = AxeDirection.Z;
    public Direction direction = Direction.Inverse;
```

```
protected int directionInt;
protected bool isOpen = false;
protected bool isActive = false;
protected float degree = 0;
protected float minCloseTransition = 0.0f;
protected AudioSource audioSource;
```

```
protected void Start() {
    audioSource = GetComponent<AudioSource>();
    switch (direction) {
        case Direction.Normal:
            directionInt = 1;
            break;
        case Direction.Inverse:
            directionInt = -1;
            break;
    }
    maxOpenTransition = Mathf.Abs(maxOpenTransition);
}
```

```
protected void OnTriggerEnter(Collider other) {
    isOpen = true;
    if (!isActive)
        StartCoroutine(transitionDoor());
}

protected void OnTriggerExit(Collider other) {
    isOpen = false;
    if (!isActive)
        StartCoroutine(transitionDoor());
}
```

```
protected IEnumerator transitionDoor() {
    isActive = true;
    float rotatedDegree;
    while (isActive) {
        rotatedDegree = degreePerSecond * Time.deltaTime;
        if (isOpen) {
            openDoor(rotatedDegree);
        } else {
            closeDoor(rotatedDegree);
        }
        yield return null;
    }
}
```

```
private void openDoor(float rotatedDegree) {
    rotateDoor(rotatedDegree, true);
}

private void closeDoor(float rotatedDegree) {
    rotateDoor(rotatedDegree, false);
}
```

```
    private void rotateDoor(float rotatedDegree, bool open) {
        audioSource.pitch = open ? 1 : -1;
        if (open ? degree == minCloseTransition : degree == maxOpenTransition)
            audioSource.Play();
        if (open)
            rotateDoor(rotatedDegree, maxOpenTransition, degree + rotatedDegree >
maxOpenTransition);
        else
            rotateDoor(-rotatedDegree, minCloseTransition, degree - rotatedDegree <
minCloseTransition);
    }
```

```
    private void rotateDoor(float rotatedDegree, float endRotatedDegree, bool
isLastRotate) {
        if (isLastRotate) {
            float diferenceValue = endRotatedDegree - degree;
            degree = endRotatedDegree;
            rotate(diferenceValue);
            audioSource.Stop();
            isActive = false;
        } else {
            degree = degree + rotatedDegree;
            rotate(rotatedDegree);
        }
    }
```

```
    private void rotate(float rotatedDegree) {
        if (axe == AxeDirection.X)
            door.transform.Rotate(new Vector3(rotatedDegree, 0, 0) * directionInt);
        else if (axe == AxeDirection.Y)
            door.transform.Rotate(new Vector3(0, rotatedDegree, 0) * directionInt);
        else if (axe == AxeDirection.Z)
            door.transform.Rotate(new Vector3(0, 0, rotatedDegree) * directionInt);
    }
```

```
}
```

En el cuadrado azul, disponemos de las variables públicas de la clase, estas serán parametrizables en cada script de cada puerta, permitiendo así configurar valores diferentes en función de la puerta a modelar. Disponen de valores por defecto, los cuales serán lo que emplearemos en la mayoría de los casos.

- **Door:** Hace referencia a la puerta que va a realizar el movimiento.
- **MaxOpenTransition:** Indica el ángulo máximo de apertura de la puerta.
- **DegreePerSecond:** Marca el ángulo que se desplaza la puerta por segundo.
- **Axe:** Indica el eje sobre el que rotará la puerta.
- **Direction:** Hace referencia a la dirección en la que la puerta se abre.

En el cuadrado rojo, aparecen las variables protected de la clase, estas serán necesarias para la lógica interna del modelado del movimiento, por ello no serán configurables y almacenarán información durante el proceso.

- **DirectionInt:** Traducción de la variable *Direction,* se empleará en el cálculo de rotación.
- **IsOpen:** Indica si la puerta está abierta.
- **IsActive:** Hace referencia a si la puerta está en movimiento.

- **Degree:** Almacena el ángulo actual de la puerta.
- **MinCloseTransition:** Hace referencia ángulo en el que la puerta está cerrada.
- **AudioSource:** Componente que contiene el sonido de la puerta al moverse.

El cuadrado verde, nos muestra el método *start*, este se encarga de inicializar las variables de *audioSource, maxOpenTransition y directionInt.*

El cuadrado naranja, contiene las funciones propias que *MonoBehaviour* proporciona cuando entramos o salimos del área de acción de un *Trigger*. Un *Trigger* es un *Collider* marcado con *isTrigger* a true, lo que permite que objetos con *Rigidbody* puedan atravesarlo, estos métodos permiten detectar cuando eso ocurre y realizar ahí una funcionalidad. En nuestro caso, queremos que cuando accedamos a cerca de la puerta, esta se abra automáticamente y cuando salgamos de él esta se cierre. Para ello, estas clases actualizan la variable *isOpen* y en el caso de no estar ya en movimiento dan la instrucción de comenzar una corrutina.

En el cuadrado morado, aparece la función de corrutina. Esta función se diferencia de las otras, debido a que se trata de una función que permite añadir interrupciones en su ejecución y ejecutarla así a lo largo de varios frames, estos cortes los definen los *yield return*. En nuestro caso, queremos rotar nuestra puerta una cantidad X de grados por frame, hasta que la variable *isActive* sea falsa y termine así su movimiento. Podríamos realizar este desarrollo en la función *Update*, pero las corrutinas son más eficientes, ya que permiten iniciarlas y terminarlas, mientras que el *Update* siempre se ejecutaría, llamándose pese a que la puerta no se encuentre en movimiento. Mientras la corrutina se encuentre en funcionamiento, calculará la rotación a aplicar en función del tiempo transcurrido y ejecutará el método correspondiente en función a la variable *isOpen.*

En el cuadrado negro, disponemos del método encargado de aplicar la rotación sobre la puerta, este recibe la cantidad de grados a rotar y tiene en cuenta las variables *axe* y *directionInt* para mover correctamente la puerta.

El cuadrado rosa, contiene la función encargada de llamar a la función antes explicada con el valor correcto del ángulo a girar, teniendo en cuenta si se trata del último giro para así calcular la diferencia entre el ángulo final y la variable *maxOpenTransition.* Se ocupará de mantener actualizada la variable *degree* y de parar el sonido y marcar *isActive* a falso en caso de ser el último giro.

En el cuadrado marrón, tenemos el método que llamará a la función anterior. Este se encargará de iniciar el sonido de la puerta, modificar el sentido de ese sonido en función de si se está abriendo o cerrando la puerta y controlar la rotación a hacer (ángulo a rotar, ángulo final a obtener y calcular si se trata de la última rotación).

El cuadrado amarillo, dispone de las clases que invoca la corrutina para abrir o cerrar la puerta. Estas funciones llaman al método anteriormente descrito, indicando si se trata de un movimiento de apertura o de cerrado.

DOOR OPEN BY BUTTON

Este script define el comportamiento de las puertas de bisagra de tipo manual, se trata de una variación del anterior, es por ello que van a compartir lógica. Para ello, se ha empleado la característica de herencia existente en C#, donde una clase puede heredar de otra y así disponer de

sus métodos y lógica. Además, permite sobrescribir métodos de la clase original para adaptarlos en ciertos aspectos en la clase hija.

```
using System;
using System.Collections;
using UnityEngine;
using UnityEngine.UI;

public class DoorOpenByButton : DoorOpen {

    public GameObject message;

    private bool informationShow;

    private new void OnTriggerEnter(Collider other) {
        informationShow = true;
        StartCoroutine(showInformation());
        changeMessage();
    }

    private new void OnTriggerExit(Collider other) {
        informationShow = false;
        StopCoroutine(showInformation());
        changeMessage("");
    }

    IEnumerator showInformation() {
        while (informationShow) {
            if (Input.GetKeyDown("f") && GamePositionManager.isInGame()) {
                isOpen = !isOpen;
                changeMessage();
                if (!isActive)
                    StartCoroutine(transitionDoor());
            }
            yield return null;
        }
    }

    private void changeMessage() {
        state = showOpen;
        changeMessage(String.Format("Presiona 'F' para {0}  la puerta", isOpen ?
"cerrar" : "abrir"));
    }

    private void changeMessage(string newMessage) {
        message.GetComponent<Text>().text = newMessage;
    }

}
```

En el cuadrado negro, aparece la definición de la clase, donde podemos observar que extiende de la clase *DoorOpen*, disponiendo de todos los métodos *public* y *protected* de la clase padre.

En el cuadrado azul, disponemos de la variable pública de la clase, debido a que hereda de otra clase, esta dispondrá de las variables de la clase padre y de la suya propia, permitiendo así parametrizar todas ellas cuando se añada a un game object.

- **Message:** Hace referencia al texto existente en Panel Game que muestra información de los objetos con los que el player puede interactuar.

En el cuadrado rojo, aparece la variable privada de la clase, esta será necesaria para la lógica interna del modelado del movimiento, por ello no será configurable y almacenará información durante el proceso.

- **InformationShow:** Booleano que indica si ha de mostrar el mensaje indicando al usuario que puede interactuar con la puerta.

El cuadrado verde, nos muestran dos métodos de utilidad que permitirá modificar el texto de game object instanciado en la variable *message.*

El cuadrado naranja, contiene las funciones de los triggers sobrescritas, en este caso lo que queremos es que dichos triggers activen una nueva corrutina que sea la encargada de disponer al usuario de la posibilidad de interactuar con las puertas. Por ello, al entrar interactuar con el trigger modificaremos la variable *showMessage*, iniciaremos o pararemos la corrutina y mostraremos u ocultaremos el texto de interacción con la puerta.

En el cuadrado morado, aparece la función de corrutina, esta permite al usuario arrancar la corrutina de *DoorOpen* si pulsa la tecla F. Este método modifica el texto de interacción con la puerta y modifica la variable *isOpen*.

4.2.3. PUERTAS CORREDERAS

En este apartado veremos el comportamiento de las puertas correderas, tanto las de tipo automático como las de tipo manual.

DOOR TRASITION OPEN

Este script define el comportamiento de las puertas correderas de tipo automático, serán empleados para las puertas existentes en la entrada de la escuela.

```
using System.Collections;
using UnityEngine;

public class DoorTrasitionOpen : MonoBehaviour {
    public GameObject door;
    public float transitionPerSecond = 1f;
    public float maxTransition = 35.60497f;
    public AxeDirection axe = AxeDirection.Y;
    public Direction direction = Direction.Normal;
```

```
protected bool isActive = false;
protected bool isOpen = false;
protected Vector3 defaultPos;
protected Vector3 openPos;
```

```
protected void Start() {
    if (door == null)
        door = gameObject;
    defaultPos = door.transform.localPosition;
    maxTransition = getMaxTransition();
    openPos = getOpenPos();
}

private float getMaxTransition() {
    float result = Mathf.Abs(maxTransition);
    int directionValue = 1;
    switch (direction) {
        case Direction.Normal:
            directionValue = 1;
            break;
        case Direction.Inverse:
            directionValue = -1;
            break;
    }
    return result * directionValue;
}

private Vector3 getOpenPos() {
    Vector3 result = Vector3.zero;
    switch (axe) {
        case AxeDirection.X:
            result = new Vector3(maxTransition, 0.0f, 0.0f);
            break;
        case AxeDirection.Y:
            result = new Vector3(0.0f, maxTransition, 0.0f);
            break;
        case AxeDirection.Z:
            result = new Vector3(0.0f, 0.0f, maxTransition);
            break;
    }
    return defaultPos + result;
}
```

```
protected void OnTriggerEnter(Collider other) {
    open();
}

protected void OnTriggerExit(Collider other) {
    close();
}
```

```
public void open() {
    isOpen = true;
    StartCoroutine(transitionDoor());
}
```

```
    public void close() {
        isOpen = false;
        StartCoroutine(transitionDoor());
    }
```

```
    protected IEnumerator transitionDoor() {
        isActive = true;
        while (isActive) {
            if (isOpen) {
                moveDoor(openPos);
            }
            else {
                moveDoor(defaultPos);
            }
            yield return null;
        }
    }
```

```
    protected void moveDoor(Vector3 finalPosition) {
        door.transform.localPosition = Vector3.Lerp(door.transform.localPosition,
finalPosition, Time.deltaTime * transitionPerSecond);
        if (door.transform.localPosition == finalPosition)
            isActive = false;
    }
```

```
}
```

En el cuadrado azul, disponemos de las variables públicas de la clase, estas serán parametrizables en cada script de cada puerta, permitiendo así configurar valores diferentes en función de la puerta a modelar. Disponen de valores por defecto, los cuales serán lo que emplearemos en la mayoría de los casos.

- **Door:** Hace referencia a la puerta que va a realizar el movimiento.
- **MaxTransition:** Indica el desplazamiento máximo de apertura de la puerta.
- **TransitionPerSecond:** Marca el desplazamiento de la puerta por segundo.
- **Axe:** Indica el eje sobre el que se mueve la puerta.
- **Direction:** Hace referencia a la dirección en la que la puerta se abre.

En el cuadrado rojo, aparecen las variables protected de la clase, estas serán necesarias para la lógica interna del modelado del movimiento, por ello no serán configurables y almacenarán información durante el proceso.

- **IsOpen:** Indica si la puerta está abierta.
- **IsActive:** Hace referencia a si la puerta está en movimiento.
- **OpenPos:** Almacena la posición final de la puerta, cuando está abierta.
- **DefaultPos:** Hace referencia a la posición inicial de la puerta, cuando está cerrada.

El cuadrado verde, nos muestra el método *start* y métodos auxiliares, estos se encargan de inicializar las variables de *door, defaultPos, maxTransition y openPos.*

El cuadrado naranja, contiene las funciones propias que *MonoBehaviour* proporciona cuando entramos o salimos del área de acción de un *Trigger*. Al igual que en *DoorOpen,* queremos que cuando nos acerquemos a la puerta, esta se abra automáticamente y cuando nos alejemos de ella se cierre.

Los cuadrados amarillos, disponen de las clases que invocan los triggers para abrir o cerrar la puerta. Estas funciones modifican el valor de la variable *isOpen* e inician la corrutina.

En el cuadrado morado, aparece la función de corrutina, donde queremos mover nuestra puerta una cantidad X por frame, hasta que la variable *isActive* sea falsa y termine así su movimiento. Mientras la corrutina se encuentre en funcionamiento, tratará de mover la puerta hacia *defaultPos u openPos,* esto dependerá del valor de la variable *isOpen.*

En el cuadrado negro, disponemos del método encargado de aplicar la translación sobre la puerta, este recibe la posición final donde desplazar la puerta y mediante el tiempo transcurrido y el movimiento por segundo desplaza la puerta a la ubicación deseada. Cuando llega a la posición final, se modifica la variable *isActive,* terminando así su movimiento.

DOOR TRASITION OPEN BY BUTTON

Este script define el comportamiento de las puertas correderas de tipo manual, al igual que ocurría con las puetas de bisagras manuales, estas heredarán de la clase DoorTrasitionOpen.

```
using System;
using System.Collections;
using UnityEngine;
using UnityEngine.UI;

public class DoorTrasitionOpenByButton : DoorTrasitionOpen {

    public GameObject message;

    private bool informationShow;

    protected new void OnTriggerEnter(Collider other) {
        informationShow = true;
        StartCoroutine(showInformation());
        changeMessage();
    }

    protected new void OnTriggerExit(Collider other) {
        informationShow = false;
        StopCoroutine(showInformation());
        changeMessage("");
    }
```

```
    IEnumerator showInformation() {
        while (informationShow) {
            if (Input.GetKeyDown("f") && GamePositionManager.isInGame()) {
                if (isOpen)
                    close();
                else
                    open();
                changeMessage();
            }
            yield return null;
        }
    }
```

```
    private void changeMessage() {
        changeMessage(String.Format("Presiona 'F' para {0}  la puerta", isOpen ?
"cerrar" : "abrir"));
    }

    private void changeMessage(string newMessage) {
        message.GetComponent<Text>().text = newMessage;
    }
```

```
}
```

En el cuadrado rojo, aparece la definición de la clase, donde podemos observar que extiende de la clase *DoorTrasitionOpen*, disponiendo de todos los métodos públicos y protected de la clase padre.

En el cuadrado azul, disponemos de la variable pública y privada de la clase, como se puede observar se ha empleado el mismo tipo de añadido visto en *DoorOpenByButton*.

El cuadrado verde, nos muestran dos métodos de utilidad iguales a los descritos en la clase *DoorOpenByButton.*

El cuadrado naranja, contiene las funciones de los triggers sobrescritas, como se puede observar dispone del mismo código descrito en *DoorOpenByButton*.

En el cuadrado morado, aparece la función de corrutina, esta permite al usuario arrancar los métodos de open o close de *DoorTrasitionOpen* si pulsa la tecla F. Además, este método modifica el texto de interacción con la puerta.

4.2.4. PUERTAS GIRATORIAS

En este apartado veremos el comportamiento de las puertas giratorias, en este caso sólo dispondremos de puertas de tipo automático, concretamente este script estará destinado a la puerta que da acceso a la sala de rebelado.

DOOR ROTATE

```
using System.Collections;
using UnityEngine;
```

```
public class DoorRotate : MonoBehaviour {
    public GameObject door;
    public float degreePerSecond = 35.0f;
    public float maxRotation = 160.0f;

    private int directionInt = 1;
    private bool isOpen = false;
    private bool isFinish = false;
    private float degree = 0.0f;

    private void OnTriggerEnter(Collider other) {
        degree = 0.0f;
        isFinish = false;
        if (isOpen) {
            directionInt = 1;
            isOpen = false;
        } else {
            directionInt = -1;
            isOpen = true;
        }
        StartCoroutine(rotateDoor());
    }

    IEnumerator rotateDoor() {
        while (!isFinish) {
            if (degree + (degreePerSecond * Time.deltaTime) > maxRotation) {
                rotate(maxRotation - Mathf.Abs(degree));
                isFinish = true;
            }
            else {
                rotate(degreePerSecond * Time.deltaTime);
            }
            yield return null;
        }
    }

    private void rotate(float rotatedDegree) {
        degree = degree + rotatedDegree;
        door.transform.Rotate(0.0f, 0.0f, rotatedDegree * directionInt);
    }
}
```

En el cuadrado azul, disponemos de las variables públicas de la clase, estas serán parametrizables en cada script de cada puerta, permitiendo así configurar valores diferentes en función de la puerta a modelar. Disponen de valores por defecto, los cuales serán lo que emplearemos en la mayoría de los casos.

- **Door:** Hace referencia a la puerta que va a realizar el movimiento.
- **MaxRotation:** Indica el ángulo máximo de apertura de la puerta.
- **DegreePerSecond:** Marca el ángulo que se desplaza la puerta por segundo.

En el cuadrado rojo, aparecen las variables privadas de la clase, estas serán necesarias para la lógica interna del modelado del movimiento, por ello no serán configurables y almacenarán información durante el proceso.

- **IsOpen:** Indica si la puerta está abierta.
- **IsFinish:** Hace referencia a si la puerta ha terminado su movimiento.
- **DirectionInt:** Indica el sentido de movimiento de la puerta.
- **Degree:** Almacena el ángulo de giro actual de la puerta.

El cuadrado naranja, contiene la función asociada al trigger. En este caso, debido a que nos encontramos en una puerta giratoria, es al acceder al interior de la puerta cuando esta comenzará su movimiento, por ello sólo disponemos de la función de acceso al trigger. Al acceder, restablecerá el valor de *degree* a cero y pondrá *isFinish* a false. Posteriormente, en función del valor *isOpen*, modificará la dirección del movimiento y actualizará *isOpen.* Finalmente ejecutará la corrutina.

En el cuadrado morado, aparece la función de corrutina, donde queremos mover nuestra puerta una cantidad X por frame, hasta que la variable *isFinish* sea falsa y termine así su movimiento. Mientras la corrutina se encuentre en funcionamiento, tratará de girar la puerta la cantidad definida en *degreePerSecond,* en caso de que el aumento de ese valor supere el valor de *MaxRotation* calculará la diferencia.

En el cuadrado verde, disponemos del método encargado de aplicar la rotación sobre la puerta, este recibe la cantidad a girar y mediante lo almacenado en *degree,* rota a la posición deseada. Además, mantiene actualizado la variable *degree* para las futuras iteraciones.

4.2.5. ADICCIÓN Y CONFIGURACIÓN DE LAS PUERTAS

Para finalizar, vamos a añadir y configurar estos scripts en nuestras puertas. Para ello accedemos a los PREFABS creados con las distintas puertas existentes en nuestro modelo.

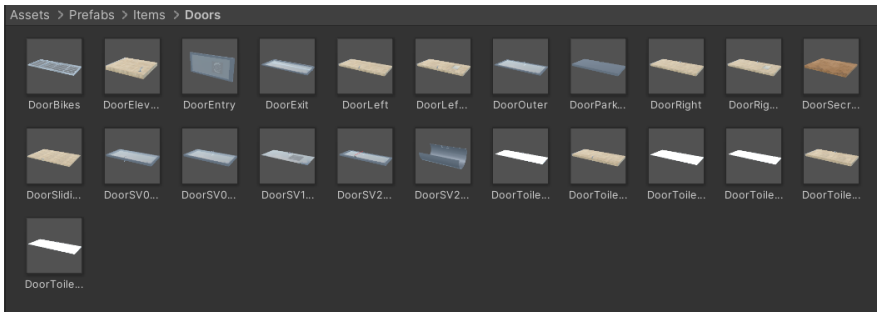


Fig. 93. PREFABS de puertas

Editamos cada uno de ellos, donde les añadiremos un game object vacío, a este le añadiremos un componente de tipo *Box Collaider*, el cual marcaremos como trigger y modificaremos sus dimensiones para que sobresalgan por delante y por detrás de la puerta, permitiendo así poder activar las funciones de los triggers. Para las puertas de tipo bisagra añadiremos un componente *AudioSource* con un archivo de audio que simule el sonido de apertura de una puerta. Para terminar, añadiremos el componente del script desarrollado y referenciaremos sus variables locales.

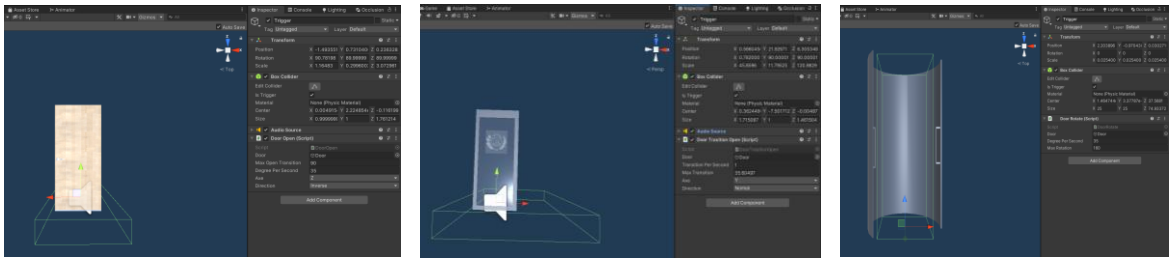


Fig. 94. Añadiendo los scripts a los PREFABS de puertas

Con esto conseguiremos actualizar todos nuestros PREFRABS, posteriormente revisaremos cada uno de ellos para limitar las posibles colisiones entre los BOX COLLIDERS, así como adaptarlos un poco según su situación.

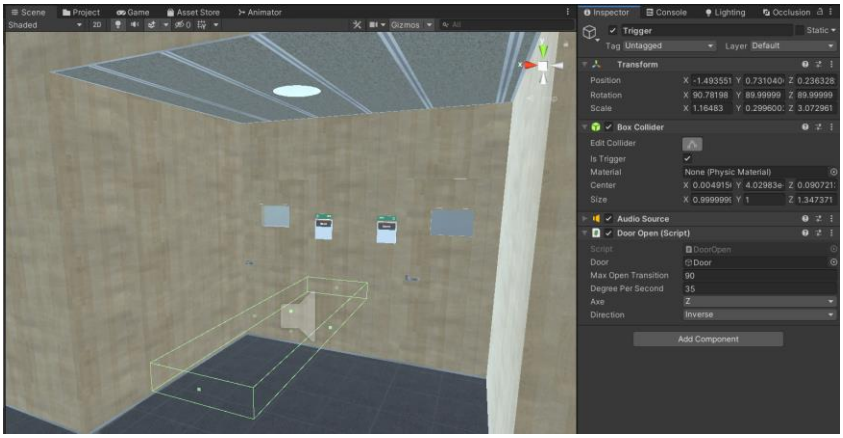


Fig. 95. Modificando los Box Colliders de las puertas

4.3. EXTENSIÓN ROOM

En esta sección, vamos a modelar el acceso a la información de las distintas salas del edificio, permitiendo obtenerla para visualizar información en la exploración del modo interactivo (carteles y *PanelGame*) o mostrar información en el *PanelRoom*.

4.3.1. ENUMERADOS DE LA EXTENSIÓN ROOM

ROOM TABLE

Esta clase define un enumerador, el cual indica la tabla a la que hace referencia la instancia.

```
public enum RoomTable {
    Seminarios,
    Servicios,
    Aulas,
    Fundacion,
    DICGF,
    Despachos,
}
```

4.3.2. CLASES DTO DE LA EXTENSIÓN ROOM

En esta extensión se han creado una serie clases DTO que definirán los elementos de la base de datos. Dispondrán de todos los datos necesarios para su empleo en la aplicación.

ROOM DATA

Se trata de una clase que define una habitación genérica de la escuela.

```
using System;

public class RoomData {

    private string code;
    private string name;
    private RoomTable table;

    public RoomData(string codeValue, string nameValue, RoomTable tableValue) {
        code = codeValue;
        name = nameValue;
        table = tableValue;
    }

    public string getCode() {
        return code;
    }

    public string getName() {
        return name.Replace("[0]", "\n");
    }

    public string getNameUi() {
        return name.Replace("[0]", Environment.NewLine + Environment.NewLine);
    }

    public RoomTable getTable() {
        return table;
    }
}
```

En el cuadrado rojo, aparecen las variables de un objeto tipo *RoomData*. Este queda definido por el código de la habitación, el nombre y la tabla a la que pertenece.

El cuadrado verde, define el constructor de la clase. Este método es el empleado para generar un objeto de este tipo, necesita todas sus variables para poder inicializarlas.

En el cuadrado azul, disponemos de todos los métodos para poder obtener la información almacenada en el objeto. Debemos destacar el campo *name*, el cual dispone de dos métodos. El primero se empleará para mostrar el texto en elementos 3D, donde el salto de línea viene definido por */n*. El segundo, se utilizará para mostrarlo en elementos UI, aquí el salto de línea se define por *Environment.NewLine*.

CLASSROOM DATA

Se trata de una clase que define un aula de la escuela.

```
using System;
using System.Collections.Generic;

public class ClassroomData : RoomData {

    private int floor;
    private string type;
    private int computers;
    private int seats;
    private string schudeleUrl;
    private string equipament;

    private Dictionary<string, string> typeDictionary = new Dictionary<string,
string>();

    public ClassroomData(string codeValue, string nameValue)
        : base(codeValue, nameValue, RoomTable.Aulas) {}

    public ClassroomData(string codeValue, string nameValue, string floorValue,
string typeCode, string seatsValue, string computersValue, string schudeleUrlValue)
        : base(codeValue, nameValue, RoomTable.Aulas) {
        typeDictionary = createTypeDictionary();
        floor = int.Parse(floorValue);
        type = typeDictionary[typeCode];
        seats = int.Parse(seatsValue);
        computers = int.Parse(computersValue);
        schudeleUrl = schudeleUrlValue;
    }

    public string getFloor() {
        return String.Format("Planta {0}", floor);
    }

    public string getType() {
        return type;
    }

    public int getComputers() {
        return computers;
    }

    public int getSeats() {
        return seats;
    }

    public bool hasSchudele() {
        return schudeleUrl != null  && schudeleUrl != "";
    }
```

```
    public string getEquipament() {
        List<string> equipamentList = new List<string>();
        if (seats > 0)
            equipamentList.Add(String.Format("Numero de asientos: {0}", seats));
        if (computers > 0)
            equipamentList.Add(String.Format("Numero de ordenadores: {0}",
computers));
        return String.Join(Environment.NewLine + Environment.NewLine,
equipamentList.ToArray());
    }
```

```
    private Dictionary<string, string> createTypeDictionary() {
        Dictionary<string, string> result = new Dictionary<string, string>();
        result["0"] = "Sin Clasificar";
        result["1"] = "Aula no informática";
        result["2"] = "Aula informática";
        return result;
    }
}
```

En el cuadrado negro, tenemos la definición de la clase, se puede observar que hereda de *RoomData.* Disponiendo así de todos los métodos y variables definidas en el padre.

En el cuadrado rojo, encontramos las variables que dispone la clase, estas serán accesibles mediante los métodos existentes en los cuadrados azules. Algunos campos adecuan su resultado para poder emplearlos directamente en textos o elementos.

El cuadrado verde, contiene los constructores de la clase. El primero, define un objeto con muy poca información, este se empleará en elementos del modelo interactivo. El segundo, dispone de mucha más información, está pensado para nutrir la información descrita en *PanelRoom*.

El cuadrado naranja, dispone de una función de utilidad que permite traducir el campo codificado que se encuentra en la base de datos. Se emplea en el constructor para disponer de él ya traducido.

OFFICE DATA

Se trata de una clase que define un despacho de la escuela.

```
using System;
using System.Collections.Generic;

public class OfficeData : RoomData {
    private string code;
    private string type;
    private string firstname;
    private string lastname;
    private int floor;
    private string email;
    private string typeTeacher;
    private string departamentCode;
    private string departament;
    private string docentUnit;
```

```
    private string url;

    private Dictionary<string, string> typeDictionary = new Dictionary<string,
string>();
    private Dictionary<string, string> typeTeacherDictionary = new Dictionary<string,
string>();
    private Dictionary<string, string> departamentDictionary = new Dictionary<string,
string>();
    private Dictionary<string, string> docentUnitDictionary = new Dictionary<string,
string>();
```

```
    public OfficeData(string codeValue, string firstnameValue, string lastnameValue)
        : base(codeValue, String.Format("{0}, {1}", lastnameValue, firstnameValue),
RoomTable.Despachos) {
        code = codeValue;
        firstname = firstnameValue;
        lastname = lastnameValue;
    }

    public OfficeData(string codeValue, string typeCode, string floorValue, string
firstnameValue, string lastnameValue, string emailValue, string typeTeacherCode,
string departamentCodeValue, string docentUnitCode, string urlValue)
        : base(codeValue, String.Format("{0}, {1}", lastnameValue, firstnameValue),
RoomTable.Despachos) {
        typeDictionary = createTypeDictionary();
        typeTeacherDictionary = createTypeTeacherDictionary();
        departamentDictionary = createDepartamentDictionary();
        docentUnitDictionary = createDocentUnitDictionary();
        code = codeValue;
        type = typeDictionary[typeCode];
        floor = int.Parse(floorValue);
        firstname = firstnameValue;
        lastname = lastnameValue;
        email = emailValue;
        typeTeacher = typeTeacherDictionary[typeTeacherCode];
        departamentCode = departamentCodeValue;
        departament = departamentDictionary[departamentCodeValue];
        docentUnit = docentUnitDictionary[docentUnitCode];
        url = urlValue;
    }
```

```
    public string getType() { return type; }

    public string getFirstname() { return firstname; }

    public string getLastname() { return lastname; }

    public string getFloor() { return String.Format("Planta {0}", floor); }

    public string getEmail() { return email; }

    public string getTypeTeacher() { return typeTeacher; }

    public string getDepartamentCode() { return departamentCode; }

    public string getDepartament() { return departament; }

    public string getDocentUnit() { return department;}
```

```
    public string getUrl() { return url;}

    private Dictionary<string, string> createTypeDictionary() {
        Dictionary<string, string> result = new Dictionary<string, string>();
        result["0"] = "Sin clasificar";
        result["1"] = "Despacho de personal docente";
        result["2"] = "Despacho de personal técnico";
        result["3"] = "Despacho de personal invitado";
        result["4"] = "Despacho vacío";
        return result;
    }

    private Dictionary<string, string> createTypeTeacherDictionary() {
        Dictionary<string, string> result = new Dictionary<string, string>();
        result["NC"] = "No Clasificado";
        result["PDI"] = "Personal Docente e Investigador";
        result["PAS"] = "Personal de Administración y Servicios";
        return result;
    }

    private Dictionary<string, string> createDepartamentDictionary() {
        Dictionary<string, string> result = new Dictionary<string, string>();
        result["NC"] = "No Clasificado";
        result["NA"] = "No Aplicable";
        result["DEGA"] = "Departamento de Expresión Gráfica Arquitectónica";
        result["DIG"] = "Departamento de Ingeniería Gráfica";
        result["DFA"] = "Departamento de Física Aplicada";
        result["DICGF"] = "Departamento de Ingeniería Cartográfica, Geodesia y
Fotogrametría";
        result["DIT"] = "Departamento de Ingeniería del Terreno";
        result["DMAA"] = "Departamento de Matemática Aplicada";
        result["DSIC"] = "Departamento de Sistemas Informáticos y Computación";
        result["DU"] = "Departamento de Urbanismo";
        result["DLA"] = "Departamento de Lingüística Aplicada";
        result["DECS"] = "Departamento de Economía y Ciencias Sociales";
        result["DIHMA"] = "Departamento de Hidráulica y Medio Ambiente";
        result["DOE"] = "Departamento de Organización de Empresas";
        return result;}

    private Dictionary<string, string> createDocentUnitDictionary() {
        Dictionary<string, string> result = new Dictionary<string, string>();
        result["NC"] = "No Clasificado";
        result["NA"] = "No Aplicable";
        result["UDCTG"] = "Unidad docente de Cartografía, Teledetección y Geografía
Física";
        result["UDF"] = "Unidad docente de Fotogrametría";
        result["UDGTGNSS"] = "Unidad docente de Geodesia y Tecnologías GNSS";
        result["UDGFA"] = "Unidad docente de Geofísica y Astronomía";
        result["UDIT"] = "Unidad docente de Instrumentos Topográficos";
        result["UDPCSIG"] = "Unidad docente de Producción Cartográfica y SIG";
        result["UDTOP"] = "Unidad docente de Topografía de Obras y Proyectos";
        result["UDTEC"] = "Unidad docente de Topografía Escuela de Caminos";
        result["UDTA"] = "Unidad docente de Topografía Agroforestal";
        result["UDTGEPSG"] = "Unidad docente de Topografía y Geografía Escuela
Politécnica Superior de Gandía";
        return result;
    }
}
```

En el cuadrado negro, tenemos la definición de la clase, se puede observar que al igual que ocurría con las aulas, hereda de *RoomData*.

Los cuadrados rojos, definen las variables de las que dispondrá la clase. Estas serán accesibles a partir de los métodos que podemos encontrar en los cuadrados azules.

El cuadrado verde, contiene los constructores de la clase. Uno se empleará para los objetos del modelo interactivo y el otro para elementos del UI.

El cuadrado naranja, dispone varias funciones de utilidad que permiten traducir los campos codificados que se encuentran en la base de datos. Se emplea en el constructor para almacenar los campos ya traducidos.

SEMINARY DATA

Se trata de una clase que define un seminario de la escuela.

```
using System;
using System.Collections.Generic;

public class SeminaryData : RoomData {

    private int floor;
    private string departamentCode;
    private string departament;
    private string docentUnit;
    private string equipament;
    private Dictionary<string, string> departamentDictionary = new Dictionary<string,
string>();
    private Dictionary<string, string> docentUnitDictionary = new Dictionary<string,
string>();

    public SeminaryData(string codeValue, string nameValue)
        : base(codeValue, nameValue, RoomTable.Seminarios) {}

    public SeminaryData(string codeValue, string nameValue, string floorValue, string
departamentCodeValue, string docentUnitCode, string equipamentValue) :
base(codeValue, nameValue, RoomTable.Seminarios) {
        departamentDictionary = createDepartamentDictionary();
        docentUnitDictionary = createDocentUnitDictionary();
        floor = int.Parse(floorValue);
        departamentCode = departamentCodeValue;
        departament = departamentDictionary[departamentCodeValue];
        docentUnit = docentUnitDictionary[docentUnitCode];
        equipament = equipamentValue;
    }

    public string getFloor() {
        return String.Format("Planta {0}", floor);
    }

    public string getDepartamentCode() {
        return departamentCode;
    }
```

```
    public string getDepartament() {
        return departament;
    }

    public string getDocentUnit() {
        return departament;
    }

    public string getEquipament() {
        return String.Format(equipament, Environment.NewLine + Environment.NewLine);
    }

    private Dictionary<string, string> createDepartamentDictionary() {
        Dictionary<string, string> result = new Dictionary<string, string>();
        result["NC"] = "No Clasificado";
        result["NA"] = "No Aplicable";
        result["DEGA"] = "Departamento de Expresión Gráfica Arquitectónica";
        result["DIG"] = "Departamento de Ingeniería Gráfica";
        result["DFA"] = "Departamento de Física Aplicada";
        result["DICGF"] = "Departamento de Ingeniería Cartográfica, Geodesia y
Fotogrametría";
        result["DIT"] = "Departamento de Ingeniería del Terreno";
        result["DMAA"] = "Departamento de Matemática Aplicada";
        result["DSIC"] = "Departamento de Sistemas Informáticos y Computación";
        result["DU"] = "Departamento de Urbanismo";
        result["DLA"] = "Departamento de Lingüística Aplicada";
        result["DECS"] = "Departamento de Economía y Ciencias Sociales";
        result["DIHMA"] = "Departamento de Hidráulica y Medio Ambiente";
        result["DOE"] = "Departamento de Organización de Empresas";
        return result;
    }

    private Dictionary<string, string> createDocentUnitDictionary() {
        Dictionary<string, string> result = new Dictionary<string, string>();
        result["NC"] = "No Clasificado";
        result["NA"] = "No Aplicable";
        result["UDCTG"] = "Unidad docente de Cartografía, Teledetección y Geografía
Física";
        result["UDF"] = "Unidad docente de Fotogrametría";
        result["UDGTGNSS"] = "Unidad docente de Geodesia y Tecnologías GNSS";
        result["UDGFA"] = "Unidad docente de Geofísica y Astronomía";
        result["UDIT"] = "Unidad docente de Instrumentos Topográficos";
        result["UDPCSIG"] = "Unidad docente de Producción Cartográfica y SIG";
        result["UDTOP"] = "Unidad docente de Topografía de Obras y Proyectos";
        result["UDTEC"] = "Unidad docente de Topografía Escuela de Caminos";
        result["UDTA"] = "Unidad docente de Topografía Agroforestal";
        result["UDTGEPSG"] = "Unidad docente de Topografía y Geografía Escuela
Politécnica Superior de Gandía";
        return result;
    }
}
```

En el cuadrado negro, tenemos la definición de la clase, se puede observar que al igual que ocurría con otras clases hereda de *RoomData*.

Los cuadrados rojos, definen las variables de las que dispondrá la clase. Estas serán accesibles a partir de los métodos que podemos encontrar en los cuadrados azules.

El cuadrado verde, contiene los constructores de la clase. Uno se empleará para los objetos del modelo interactivo y el otro para elementos del UI.

El cuadrado naranja, dispone varias funciones de utilidad que permiten traducir los campos codificados que se encuentran en la base de datos. Se emplea en el constructor para almacenar los campos ya traducidos.

SERVICE DATA

Se trata de una clase que define un servicio de la escuela. Debido a que los elementos de la tabla Fundación y DICGF disponen del mismo modelo de datos, serán definidas por esta clase.

```
using System;
using System.Collections.Generic;

public class ServiceData : RoomData {
    private int floor;
    private string type;
    private string description;
    private string moreInformation;
    private string openHours;
    private string url;
    private Dictionary<string, string> typeDictionary = new Dictionary<string,
string>();

    public ServiceData(RoomTable tableValue, string codeValue, string nameValue)
        : base(codeValue, nameValue, tableValue) {}

    public ServiceData(RoomTable tableValue, string codeValue, string nameValue,
string floorValue, string typeCode, string descriptionValue, string
moreInformationValue, string openHoursValue, string urlValue)
        : base(codeValue, nameValue, tableValue) {
        typeDictionary = createTypeDictionary();
        floor = int.Parse(floorValue);
        type = typeDictionary[typeCode];
        description = descriptionValue;
        moreInformation = moreInformationValue;
        openHours = openHoursValue;
        url = urlValue;
    }

    public string getFloor() {
        return String.Format("Planta {0}", floor);
    }

    public string getType() {
        return type;
    }

    public string getDescription() {
        return description;
    }
```

```
    public string getMoreInformation() {
        return String.Format(moreInformation, Environment.NewLine +
Environment.NewLine);
    }

    public string getOpenHoursValue() { return openHours; }

    public string getUrl() { return url; }
```

```
    private Dictionary<string, string> createTypeDictionary(){
        Dictionary<string, string> result = new Dictionary<string, string>();
        result["0"] = "Otros";
        result["1"] = "Aseos o Vestuarios";
        result["2"] = "Salas de Reunión";
        result["3"] = "Despachos";
        result["4"] = "Almacenes";
        result["5"] = "Aulas";
        result["6"] = "Salas de Personal";
        return result;
    }
```

```
}
```

En el cuadrado negro, tenemos la definición de la clase, se puede observar que al igual que ocurría con otras clases hereda de *RoomData*.

Los cuadrados rojos, definen las variables de las que dispondrá la clase. Estas serán accesibles a partir de los métodos que podemos encontrar en los cuadrados azules.

El cuadrado verde, contiene los constructores de la clase. Uno se empleará para los objetos del modelo interactivo y el otro para elementos del UI. En el caso de servicios, el campo *table* nos vendrá en el constructor, ya que estos elementos pueden ser de tablas diferentes.

El cuadrado naranja, dispone una función de utilidad que permite traducir el campo codificado que se encuentra en la base de datos. Se emplea en el constructor para almacenar el campo ya traducido.

4.3.3. GAME ROOM MANAGER

Encargada de disponer de variables y métodos necesarios para gestionar la información de la habitación consultada, así como de disponer de métodos para la llamada y transformación de elementos de la base de datos a objetos DTO.

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public static class GameRoomManager {
    private static string FIELD_ROOM_LABEL_INFO = "LABEL_NAME";
    private static string FIELD_ROOM_DICGF_LABEL_INFO = "NOMBRE";
```

```
    private static List<string> FIELDS_CLASSROOM_INFO = new List<string>() {
"NOMBRE", "PLANTA", "TIPO", "Nº_ASIENTOS", "Nº_ORDENADORES", "HORARIO" };
    private static string TABLE_CLASSROOM = "Aulas";
    private static string WHERE_CLASSROOM_CONDITION = "ID_AULAS = '{0}'";

    private static List<string> FIELDS_SEMINARY_INFO = new List<string>() { "NOMBRE",
"PLANTA", "DEPARTAMENTO", "UNIDAD_DOCENTE", "DOTACION" };
    private static string TABLE_SEMINARY = "Seminarios";
    private static string WHERE_SEMINARY_CONDITION = "ID_SEMINARIO = '{0}'";

    private static List<string> FIELDS_SERVICE_INFO = new List<string>() { "NOMBRE",
"PLANTA", "TIPO", "DESCRIPCION", "MAS_INFO", "HORARIO", "URL" };
    private static string TABLE_SERVICE = "Servicios";
    private static string WHERE_SERVICE_CONDITION = "ID_SERVICIOS = '{0}'";
    private static string TABLE_FUN = "Fundacion";
    private static string WHERE_FUN_CONDITION = "ID_FUN = '{0}'";
    private static string TABLE_DICGF = "DICGF";
    private static string WHERE_DICGF_CONDITION = "ID_DICGF = '{0}'";

    private static List<string> FIELDS_OFFICE_MAIN_INFO = new List<string>() {
"TIPO", "PLANTA" };
    private static List<string> FIELDS_OFFICE_JOIN_INFO = new List<string>() {
"NOMBRE", "APELLIDOS", "CORREO_UPV", "TIPO", "DEPARTAMENTO", "UNIDAD_DOCENTE",
"MAS_INFO" };
    private static string FIELD_MAIN_OFFICE = "ID_PERSONAL";
    private static string FIELD_JOIN_OFFICE = "ID";
    private static string TABLE_MAIN_OFFICE = "Despachos";
    private static string TABLE_JOIN_OFFICE = "Personal";
    private static string WHERE_OFFICE_CONDITION = "ID_DESPACHOS = '{0}'";
```

```
    public static string roomID = "SV140";
    public static RoomTable roomTable = RoomTable.Servicios;
```

```
    public static RoomData getRoomByID(string roomCode, RoomTable roomTable) {
        switch (roomTable) {
            case RoomTable.Seminarios:
                return getRoomSeminaryByID(roomCode, roomTable);
            case RoomTable.Servicios:
                return getRoomServiceByID(roomCode, roomTable);
            case RoomTable.Aulas:
                return getRoomClassroomByID(roomCode, roomTable);
            case RoomTable.Fundacion:
                return getRoomFunByID(roomCode, roomTable);
            case RoomTable.DICGF:
                return getRoomDICGFByID(roomCode, roomTable);
            case RoomTable.Despachos:
                return getRoomOfficeByID(roomCode, roomTable);
        }
        return null;
    }
```

```
    public static ClassroomData getClassroomByID(string classroomCode) {
        string query = GameDatabaseManager.obtainBasicQuery(FIELDS_CLASSROOM_INFO,
TABLE_CLASSROOM, String.Format(WHERE_CLASSROOM_CONDITION, classroomCode));
        List<string> dataList = GameDatabaseManager.obtainSingleElement(query);
```

```
        return dataList == null || dataList.Count == 0 ? null : new
ClassroomData(classroomCode, dataList[0], dataList[1], dataList[2], dataList[3],
dataList[4], dataList[5]);
    }

    public static SeminaryData getSeminaryByID(string seminaryCode) {
        string query = GameDatabaseManager.obtainBasicQuery(FIELDS_SEMINARY_INFO,
TABLE_SEMINARY, String.Format(WHERE_SEMINARY_CONDITION, seminaryCode));
        List<string> dataList = GameDatabaseManager.obtainSingleElement(query);
        return dataList == null || dataList.Count == 0 ? null : new
SeminaryData(seminaryCode, dataList[0], dataList[1], dataList[2], dataList[3],
dataList[4]);
    }

    public static ServiceData getServiceByID(string serviceCode) {
        return getServiceByIDAndServiceType(serviceCode, RoomTable.Servicios);
    }

    public static ServiceData getFunByID(string funCode) {
        return getServiceByIDAndServiceType(funCode, RoomTable.Fundacion);
    }

    public static ServiceData getDICGFByID(string DICGFCode) {
        return getServiceByIDAndServiceType(DICGFCode, RoomTable.DICGF);
    }

    public static OfficeData getOfficeByID(string officeCode) {
        string query = GameDatabaseManager.obtainJoinQuery(FIELDS_OFFICE_MAIN_INFO,
TABLE_MAIN_OFFICE, FIELD_MAIN_OFFICE, FIELDS_OFFICE_JOIN_INFO, TABLE_JOIN_OFFICE,
FIELD_JOIN_OFFICE, "M." + String.Format(WHERE_OFFICE_CONDITION, officeCode));
        List<string> dataList = GameDatabaseManager.obtainSingleElement(query);
        return dataList == null || dataList.Count == 0 ? null : new
OfficeData(officeCode, dataList[0], dataList[1], dataList[2], dataList[3],
dataList[4], dataList[5], dataList[6], dataList[7], dataList[8]);
    }
```

```
    private static RoomData getRoomClassroomByID(string roomCode, RoomTable
roomTable) {
        return getRoomByID(roomCode, roomTable, FIELD_ROOM_LABEL_INFO,
TABLE_CLASSROOM, WHERE_CLASSROOM_CONDITION);
    }

    private static RoomData getRoomSeminaryByID(string roomCode, RoomTable roomTable)
{
        return getRoomByID(roomCode, roomTable, FIELD_ROOM_LABEL_INFO,
TABLE_SEMINARY, WHERE_SEMINARY_CONDITION);
    }

    private static RoomData getRoomServiceByID(string roomCode, RoomTable roomTable)
{
        return getRoomByID(roomCode, roomTable, FIELD_ROOM_LABEL_INFO, TABLE_SERVICE,
WHERE_SERVICE_CONDITION);
    }

    private static RoomData getRoomFunByID(string roomCode, RoomTable roomTable) {
        return getRoomByID(roomCode, roomTable, FIELD_ROOM_LABEL_INFO, TABLE_FUN,
WHERE_FUN_CONDITION);
    }
```

```
    private static RoomData getRoomDICGFByID(string roomCode, RoomTable roomTable) {
        return getRoomByID(roomCode, roomTable, FIELD_ROOM_DICGF_LABEL_INFO,
TABLE_DICGF, WHERE_DICGF_CONDITION);
    }

    private static RoomData getRoomByID(string roomCode, RoomTable roomTable, string
field, string table, string where) {
        string query = GameDatabaseManager.obtainBasicQuery(field, table,
String.Format(where, roomCode));
        return getRoomByID(roomCode, roomTable, query);
    }

    private static RoomData getRoomOfficeByID(string roomCode, RoomTable roomTable) {
        string query = GameDatabaseManager.obtainJoinQuery(FIELD_ROOM_LABEL_INFO,
TABLE_MAIN_OFFICE, FIELD_MAIN_OFFICE, TABLE_JOIN_OFFICE, FIELD_JOIN_OFFICE, "M." +
String.Format(WHERE_OFFICE_CONDITION, roomCode));
        return getRoomByID(roomCode, roomTable, query);
    }

    private static RoomData getRoomByID(string roomCode, RoomTable roomTable, string
query) {
        string roomName = GameDatabaseManager.obtainField(query);
        return roomName == null || roomName == "" ? null : new RoomData(roomCode,
roomName, roomTable);
    }
```

```
    private static ServiceData getServiceByIDAndServiceType(string code, RoomTable
roomTable) {
        string query = "";
        if (roomTable == RoomTable.Servicios)
            query = GameDatabaseManager.obtainBasicQuery(FIELDS_SERVICE_INFO,
TABLE_SERVICE, String.Format(WHERE_SERVICE_CONDITION, code));
        else if (roomTable == RoomTable.Fundacion)
            query = GameDatabaseManager.obtainBasicQuery(FIELDS_SERVICE_INFO,
TABLE_FUN, String.Format(WHERE_FUN_CONDITION, code));
        else if (roomTable == RoomTable.DICGF)
            query = GameDatabaseManager.obtainBasicQuery(FIELDS_SERVICE_INFO,
TABLE_DICGF, String.Format(WHERE_DICGF_CONDITION, code));
        if (query == "")
            return null;
        else {
            List<string> dataList = GameDatabaseManager.obtainSingleElement(query);
            return dataList == null || dataList.Count == 0 ? null : new
ServiceData(roomTable, code, dataList[0], dataList[1], dataList[2], dataList[3],
dataList[4], dataList[5], dataList[6]);
        }
    }
```

```
    public static string getRoomID() { return roomID; }

    public static void setRoomID(string roomIDValue) { roomID = roomIDValue; }

    public static RoomTable getRoomTable() { return roomTable;}

    public static void setRoomTable(RoomTable roomTableValue) {
        roomTable = roomTableValue;
    }
}
```

En el cuadrado rojo, aparecen las variables privadas de la clase, las cuales serán utilizadas para la generación de las queries que se realizarán a la base de datos mediante *GameDatabaseManager*.

El cuadrado azul, contiene las variables públicas que dispone la clase, estas se emplearán para almacenar la información del aula que se va a consultar en el panel *PanelRoom*.

En los cuadrados verdes, disponemos de los métodos que permiten obtener la información básica de una habitación, obteniendola a partir del código y el tipo de tabla, el resultado es un *RoomData.* Estos métodos se apoyan en las variables privadas y en las clases *RoomData* y *GameDatabaseManager* para obtener el resultado. Serán empleados por los scripts que asociaremos a los objetos existentes en el modelo interactivo.

Los cuadrados naranjas, muestran los métodos que permiten obtener la información completa de una habitación obteniéndolo a partir del código de la sala. Estos métodos se apoyan en las variables privadas, en las clases *DTO* y en *GameDatabaseManager* para obtener el resultado. Serán empleado por los scripts que asociaremos al *PanelRoom*.

En el cuadrado morado, se definen los métodos que nos permitirán almacenar y consultar la información de la habitación a mostrar. Desde los objetos con los que podemos interactuar en el modelo, guardaremos la información de la habitación consultada. Esta información la recuperaremos posteriormente para con ella obtener la información a mostrar en el *PanelRoom*.

4.3.4. CLASES GAME DE LA EXTENSIÓN ROOM

En esta extensión se han creado una serie clases que extienden de *MonoBehaviour,* destinadas a añadirlas a elementos existentes en el modelo interactivo. Dentro de nuestro modelo disponemos de tres tipos de elementos con información de las habitaciones:

- **Paneles en los pasillos:** Paneles horizontales que muestran información de aulas.
- **Carteles en las puertas:** Carteles cercanos a las puertas de las habitaciones.
- **Triggres en las habitaciones:** Triggers dispuestos en las habitaciones accesibles.

Fig. 96. Elementos que dispondrán de scripts en la extensión Room

ROOM PANEL INFORMATION

Encargada de rellenar la información visible en los Paneles de los pasillos.

```
using System;
using System.Collections.Generic;
using UnityEngine;

public class RoomPanelInformation : MonoBehaviour {

    public RoomTable roomTable;
    public string roomNumberValue;

    protected List<GameObject> roomNameLabelList = new List<GameObject>();
    protected RoomData room;
    private Dictionary<RoomTable, string> roomCodeDictionary = new
Dictionary<RoomTable, string>();

    void Awake() {
        roomCodeDictionary[RoomTable.Seminarios] = "SM{0}";
        roomCodeDictionary[RoomTable.Servicios] = "SV{0}";
        roomCodeDictionary[RoomTable.Aulas] = "A{0}";
        roomCodeDictionary[RoomTable.Fundacion] = "FUN{0}";
        roomCodeDictionary[RoomTable.DICGF] = "DICGF{0}";
        roomCodeDictionary[RoomTable.Despachos] = "D{0}";
    }


    protected virtual void Start () {
        updateRoomNameLabelList();
        string roomCode = String.Format(roomCodeDictionary[roomTable],
roomNumberValue);
        room = GameRoomManager.getRoomByID(roomCode, roomTable);
        if (room == null)
            room = new RoomData(roomCode, "", roomTable);
        populateNameRoom(room);
    }


    protected virtual void updateRoomNameLabelList() {
        roomNameLabelList.Add(transform.Find("Name_1").gameObject);
        roomNameLabelList.Add(transform.Find("Name_2").gameObject);
    }

    protected virtual void populateNameRoom(RoomData room) {
        foreach (GameObject roomNameLabel in roomNameLabelList)
            populateNameRoom(room, roomNameLabel);
    }

    protected void populateNameRoom(RoomData room, GameObject label) {
        if (hasInformation())
            populate(room.getName(), label);
        else
            notFindNameRoom(room, label);
    }
```

```
    protected void populate(string roomName, GameObject label) {
        if (nameHasMoreLines())
            label.GetComponent<TextMesh>().characterSize = getSmallFontSize();
        else
            label.GetComponent<TextMesh>().characterSize = getBigFontSize();
        label.GetComponent<TextMesh>().text = roomName;
    }

    protected virtual void notFindNameRoom(RoomData room, GameObject label) {
        label.GetComponent<TextMesh>().text = room.getTable().ToString().ToUpper();
    }

    protected virtual float getSmallFontSize() {
        return 0.7f;
    }

    protected virtual float getBigFontSize() {
        return 0.9f;
    }

    protected bool hasInformation() {
        string roomName = room.getName();
        return roomName != null && roomName != "";
    }

    protected bool nameHasMoreLines() {
        return room.getName().IndexOf("\n") != -1;
    }
}
```

En el cuadrado azul, aparecen las variables públicas de la clase. La primera, indica la tabla en la que se encuentra la información de ese panel. La segunda, hace referencia al número de la habitación, este coincide con la numeración del id en la base de datos (Ejemplo: El servicio SV003 tendría de roomNumbreValue 003)

El cuadrado rojo, contiene la definición de las variables protegidas y privadas de la clase.

- **RoomNameLabelList:** Contiene el listado de textos que tiene que actualizar con la información de la base de datos.
- **Room:** Almacena un objeto de tipo *RoomData* con la información de la habitación
- **RoomCodeDictionary:** Contiene un diccionario que permite relacionar una *RoomTable* con el prefijo que dispone el id de su tabla.

En el cuadrado verde, tenemos los métodos que permite rellenar las variables de la clase y añadir la información de la habitación. En el método *Awake* se rellena los valores del diccionario. En el método *Start* se rellena los valores de la lista de textos a actualizar, la información del *RoomData* y se pone esta información en los distintos textos almacenados en *RoomNameLabelList*.

El cuadrado morado, define unos métodos auxiliares que permiten definir el tamaño de la fuente, validar si dispone la variable *room* del atributo *name* y comprobar si este contiene saltos de línea.

Los cuadrados naranjas, definen los métodos que permiten rellenar la información de *RoomData* en los distintos textos. En ellos vemos que en caso de no disponer de información se muestra el valor

del método *notFindNameRoom.* También se puede observar que se pone el tamaño de la fuente a un valor u otro en función de la existencia de saltos de línea en la variable *name* del *RoomData*

ROOM INFORMATION

Encargada de rellenar la información visible en los carteles de las puertas. Esta clase extenderá de la clase anterior y sobrescribirá ciertos métodos.

```
using System.Collections;
using UnityEngine;

public class RoomInformation : RoomPanelInformation {
    public GameObject roomPanel;

    private GameObject roomMoreInformation;
    private bool informationShow;

    protected override void Start() {
        GameObject roomNumber = transform.Find("Num").gameObject;
        roomNumber.GetComponent<TextMesh>().text = roomNumberValue;
        roomMoreInformation = transform.Find("Info").gameObject;
        base.Start();
    }

    private void OnTriggerEnter(Collider other){
        if (hasInformation()) {
            roomMoreInformation.SetActive(true);
            informationShow = true;
            StartCoroutine(showInformation());
        }
    }

    private void OnTriggerExit(Collider other){
        if (hasInformation()) {
            roomMoreInformation.SetActive(false);
            informationShow = false;
            StopCoroutine(showInformation());
        }
    }

    IEnumerator showInformation() {
        while (informationShow) {
            if (Input.GetKeyDown("f") && GamePositionManager.isInGame()) {
                GameRoomManager.setRoomID(room.getCode());
                GameRoomManager.setRoomTable(room.getTable());
                roomPanel.gameObject.SetActive(true);
            }
            yield return null;
        }
    }

    protected override void updateRoomNameLabelList() {
        roomNameLabelList.Add(transform.Find("Name").gameObject);
    }
```

```
    protected override void notFindNameRoom(RoomData room, GameObject label) {
        label.GetComponent<TextMesh>().characterSize = getBigFontSize();
        if (room.getTable() == RoomTable.Despachos)
            label.GetComponent<TextMesh>().text = "Despacho: \n VACÍO";
        else
            base.notFindNameRoom(room, label);
    }

    protected override float getSmallFontSize() {
        return 0.5f;
    }

    protected override float getBigFontSize() {
        return 0.6f;
    }
}
```

En el cuadrado azul, aparece la variable pública que contiene la clase, esta hace referencia al panel donde se mostrará la información.

El cuadrado rojo, aparecen las variables privadas que contiene la clase. La primera hace referencia a la zona del cartel donde se mostrará que tecla ha de pulsar el usuario para ver la información asociada. La segunda indica si se ha de mostrar dicha información, esta se empleará en la corrutina que espera a que el usuario interactúe con el cartel.

En el cuadrado verde, se muestra cómo se rellena la información referente al número que aparece en el cartel. También permite obtener el valor para la variable *roomMoreInformation.* Para finalizar realiza la lógica que vemos en la clase *RoomPanelInformation.*

El cuadrado naranja, contiene las clases que se ejecutan cuando se interactúa con el trigger. El objetivo de estas será mostrar el texto que permite interactuar con el botón e iniciar la corrutina. En caso de salir de la zona de acción del trigger, ocultará la información y detendrá la corrutina. En ambos casos sólo lo realizará si dicho cartel dispone de información, es decir si dicho cartel hace referencia a una instancia en la base de datos.

En el cuadrado morado, tenemos el método de la corrutina. Este permite interactuar pulsando la tecla F del teclado, al realizarlo almacenamos el id de la habitación y la tabla a la que pertenece. Posteriormente se muestra el panel UI con la información de la sala.

El cuadrado amarillo, contiene los métodos que se han sobrescritos de la clase padre, definiendo donde se encuentra y el tamaño del texto que contendrá el nombre de la habitación. Además, se redefine el método *notFindNameRoom*, donde se muestra un texto especial en caso de que se trate de un despacho.

ROOM INFORMATION

Encargada de rellenar la información visible en el *PanelGame*, con los datos de la habitación en la que hemos entrado. Esta clase extenderá de la clase *RoomPanelInformation* y sobrescribirá ciertos métodos.

```
using System;
using UnityEngine;
using UnityEngine.UI;

public class RoomUpdateName : RoomPanelInformation {

    public GameObject informationContent;

    private void OnTriggerEnter(Collider other) {
        if (hasInformation()) {
            int fontSize = nameHasMoreLines() ? 8 : 10;
            foreach (GameObject roomNameLabel in roomNameLabelList)
                updateUiRoomNameLabel(roomNameLabel, fontSize,
roomNameLabel.GetComponent<Text>().text + Environment.NewLine + Environment.NewLine +
room.getNameUi());
        }
    }

    private void OnTriggerExit(Collider other) {
        if (hasInformation()) {
            foreach (GameObject roomNameLabel in roomNameLabelList)
                updateUiRoomNameLabel(roomNameLabel, 15, String.Format("Planta {0}",
GameMapManager.getFloor()));
        }
    }

    protected override void updateRoomNameLabelList() {
        roomNameLabelList.Add(informationContent);
    }

    protected override void populateNameRoom(RoomData room) {
        //Not is necessary this method
    }

    private void updateUiRoomNameLabel(GameObject label, int fontSize, string
labelValue) {
        label.GetComponent<Text>().fontSize = fontSize;
        label.GetComponent<Text>().text = labelValue;
    }
}
```

En el cuadrado azul, aparece la variable pública que contiene la clase, esta hace referencia al texto existente en el *PanelGame*.

El cuadrado rojo, tenemos los métodos sobrescritos de la clase padre. En el primero indicamos que objeto contendrá el texto a mostrar. La segunda, se deja sin implementación, ya que para este caso no deseamos que rellene la información según la lógica del elemento padre, la cual rellena la información al iniciar la aplicación. En este caso deseamos que se realice en las clases de interacción con los triggers.

El cuadrado naranja, contiene las clases que se ejecutan cuando se interactúa con el trigger. Al entrar él se actualizará la información del *PanelGame*, mediante el método que se muestra en el cuadrado verde. Al salir, se cambiará la información, indicando la planta actual en la que se encuentra el usuario.

ADICCIÓN Y CONFIGURACIÓN DE LOS GAME OBJECTS

Una vez realizados los scripts, vamos a asignárselos a los distintos objetos de nuestro proyecto.

En primer lugar, añadiremos los paneles de los pasillos, donde agregaremos el componente RoomPanelInformation y le configuraremos su id y la tabla a la que hacen referencia.

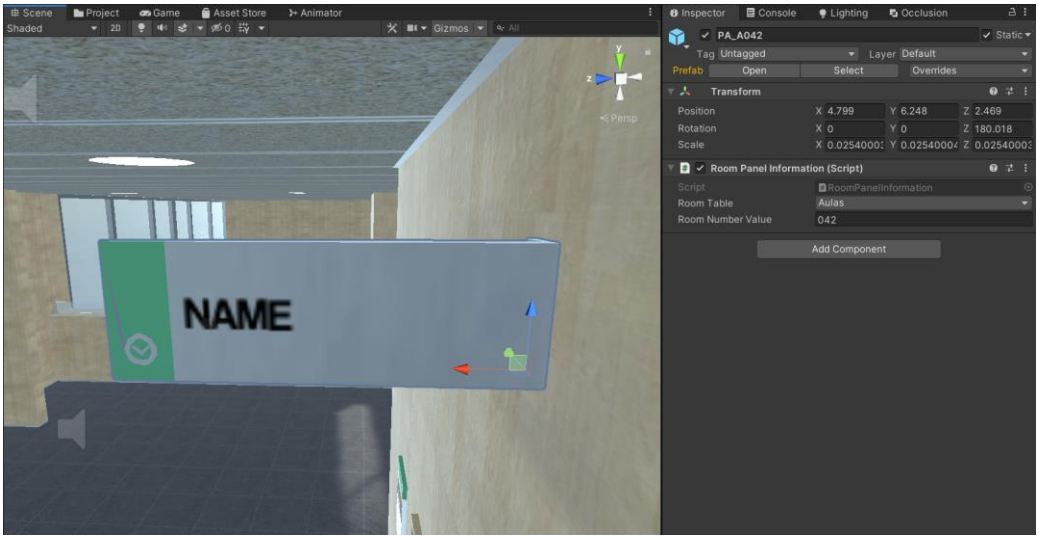


Fig. 97. Añadiendo scripts a los paneles de los pasillos

En segundo lugar, añadiremos los scripts a los carteles de las puertas, antes de hacerlo editaremos los PREFABS para añadir los elementos comunes. Añadiremos los triggers, el componente *RoomInformation* y los tres elementos *TextMesh* que contendrán la información, número, nombre y texto de interacción con el cartel.

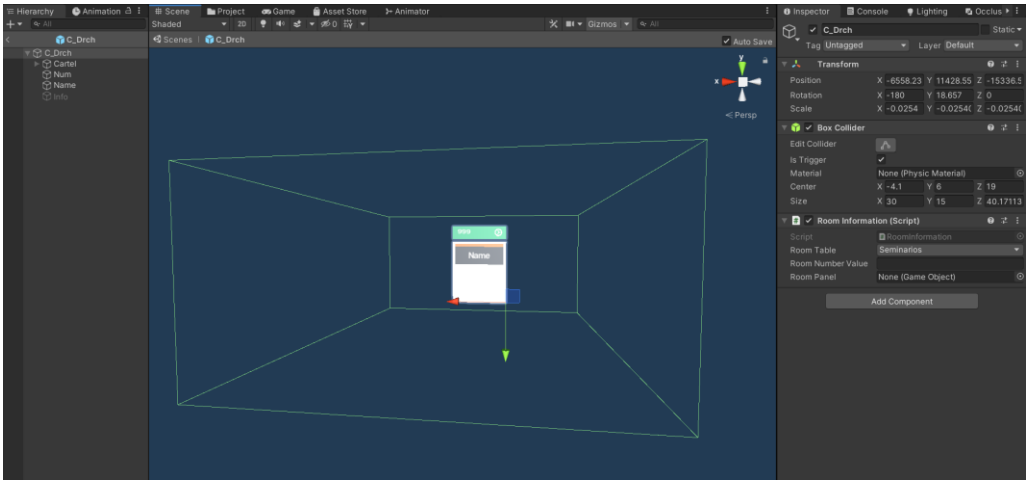


Fig. 98. Añadiendo elementos a los carteles de las puertas

Una vez modificado del PREFAB, pasaremos a modificar cada uno de ellos, introduciendo la información pertinente. También adecuaremos los triggers para que se encuentren bien distribuidos en función de la localización del cartel.

Fig. 99. Adecuando los triggers y variables a los carteles de las puertas

En último lugar, crearemos todos los elementos de los triggers de las habitaciones accesibles. Crearemos un PREFAB que contendrá un game object vacío al que le añadiremos un *Box Collider y RoomUpdateName.* Iremos instanciando estos PREFABS por nuestro modelo adecuando su trigger a las dimensiones de la habitación.

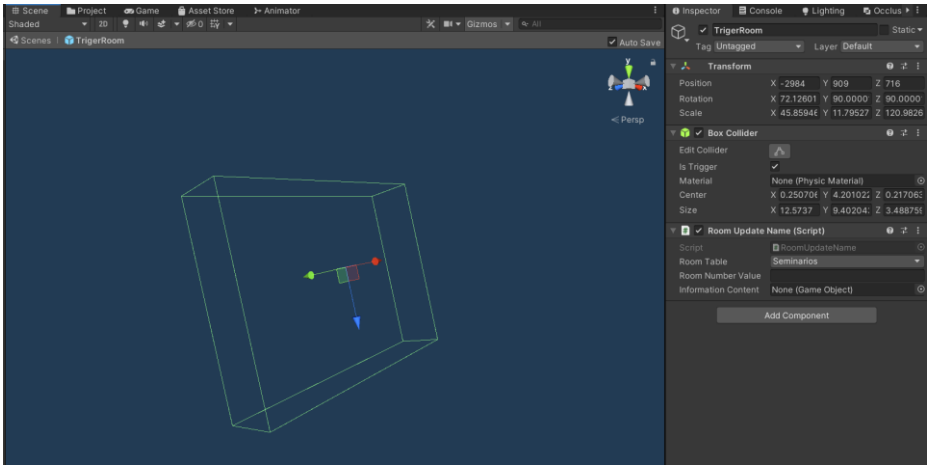


Fig. 100. PREFAB de los triggers de las habitaciones.

4.3.1. CLASES UI DE LA EXTENSIÓN ROOM

En este apartado vamos a ver las clases creadas para dar la funcionalidad al RoomPanel.

ROOM SCENE INIT

Clase que se asignará directamente al RoomPanel, está será la encargada mostrar el panel correspondiente al tipo de habitación.

```
using UnityEngine;
using UnityEngine.UI;

public class RoomSceneInit : MonoBehaviour {

    public GameObject classroomTable;
    public GameObject seminaryTable;
    public GameObject serviceTable;
    public GameObject officeTable;
    public GameObject roomTitle;

    private GameObject player;

    void OnEnable() {
        player = GameObject.Find("Player");
        player.GetComponent<PlayerController>().showCursor();
        stopTime(true);
        hideTablesAndTitle();
        selectTable();
    }

    public void closeRoomInformation() {
        stopTime(false);
        player.GetComponent<PlayerController>().hideCursor();
        hideTablesAndTitle();
        gameObject.SetActive(false);
    }

    private void stopTime(bool stop) { Time.timeScale = stop ? 0 : 1;}

    private void hideTablesAndTitle() {
        classroomTable.SetActive(false);
        seminaryTable.SetActive(false);
        serviceTable.SetActive(false);
        officeTable.SetActive(false);
        roomTitle.GetComponent<Text>().text = "";}

    private void selectTable() {
        switch (GameRoomManager.getRoomTable()) {
            case RoomTable.Aulas:
                classroomTable.SetActive(true);
                break;
            case RoomTable.Despachos:
                officeTable.SetActive(true);
                break;
            case RoomTable.Seminarios:
                seminaryTable.SetActive(true);
                break;
            case RoomTable.Servicios:
                serviceTable.SetActive(true);
                break;
            case RoomTable.Fundacion:
                serviceTable.SetActive(true);
                break;
            case RoomTable.DICGF:
                serviceTable.SetActive(true);
                break;
        }}
}
```

En el cuadrado rojo, se definen las variables públicas de la clase, donde tenemos los distintos paneles de las tipologías de las habitaciones y el título del *PanelRoom*.

El cuadrado azul, muestra la variable privada de la clase, esta hace referencia al usuario.

En el cuadrado verde, se especifican dos métodos. El primero, adecua el cursor y el tiempo de juego para poder interactuar con el UI. Posteriormente oculta y restaura la visualización de los elementos del *PanelRoom* al valor inicial. Finalmente muestra el panel correspondiente a la tipología de habitación a mostrar. El segundo, retorna a los valores originales el cursor, el tiempo y el *PanelRoom.*

El cuadrado naranja, dispone de los métodos auxiliares que emplea la clase. El primero, modifica el tiempo de juego. El segundo, restaura la visualización de los elementos del UI. El tercero, determina el panel a mostrar en función de la tipología de habitación consultada.

ROOM ABSTRACT TABLE INIT

Clase abstracta que determina la estructura y métodos que ha de implementar las clases definidas en cada panel de las distintas tipologías de habitación.

```
using System;
using UnityEngine;
using UnityEngine.UI;

public abstract class RoomAbstractTableInit : MonoBehaviour {

    public GameObject roomImage;
    public GameObject roomTitle;

    protected Sprite roomSprite;

    protected void setRoomSprite(string roomCode) {
        roomSprite = Resources.Load<Sprite>(String.Format(getFolder(), roomCode));
        if (roomSprite == null)
            roomSprite = Resources.Load<Sprite>(String.Format(getFolder(), "ERROR"));
    }

    protected void populateRoomTitle(string roomTitleValue) {
        roomTitle.GetComponent<Text>().text = roomTitleValue.ToUpper();
    }

    protected void populateRoomImage() {
        roomImage.GetComponent<Image>().sprite = roomSprite;
    }

    protected abstract string getFolder();
}
```

En el cuadro verde, vemos la definición de la clase, donde se observa que se trata de una clase de tipo abstracta. Una clase abstracta no se puede instanciar y sirve para disponer de métodos comunes, definir métodos a implementar y definir variables de la clase a los objectos que van a heredar de ella.

El cuadrado rojo, dispone de las variables públicas de la clase. Aquí se hace referencia al título del *PanelRoom* y a la imagen definida en el panel de la propia tipología.

En el cuadrado azul, aparece una variable privada de la clase. Esta hace referencia al *sprite* que se pondrá en la imagen del panel de la propia tipología.

El cuadrado naranja, contiene los métodos auxiliarles que dispone la clase y son utilizables por las clases que heredaran de ella. El primero, rellena el valor del sprite en función del roomCode proporcionado, en caso de no encontrarlo, mostrará el archivo con el nombre ERROR localizado en la carpeta del tipo de habitación. El segundo, permite actualizar el nombre del título del *PanelRoom.* El tercero, modifica la imagen definida en el panel de la propia tipología con el valor del *sprite.*

El cuadrado morado, define un método para obtener el *path* de la carpeta creada en *Resources* del tipo de habitación. Debido a que este es diferente para cada tipología se define como abstracto y se deja sin implementar, serán sus hijas las que lo implementarán.

ROOM CLASSROOM TABLE INIT

Clase que permite mostrar la información de un aula en el *PanelRoom.* Esta clase hereda de *RoomAbstractTableInit.*

```
using UnityEngine;
using UnityEngine.UI;

public class RoomClassroomTableInit : RoomAbstractTableInit{

    public GameObject classroomName;
    public GameObject classroomFloor;
    public GameObject classroomType;
    public GameObject classroomEquipment;
    public GameObject schudeleButton;

    void OnEnable() {
        ClassroomData classroom =
GameRoomManager.getClassroomByID(GameRoomManager.getRoomID());
        if (classroom != null) {
            populateClassroomInformation(classroom);
            showSchudeleButton(classroom);
        } else {
            removeClassroomInformation();
            showSchudeleButton(false);
        }
    }

    protected override string getFolder() {
        return "Sprites/Aulas/{0}";
    }

    private void populateClassroomInformation(ClassroomData classroom) {
        setRoomSprite(classroom.getCode());
        populateClassroomInformation(classroom.getName(), classroom.getFloor(),
classroom.getType(), classroom.getEquipament());
    }
```

```
    private void removeClassroomInformation() {
        setRoomSprite("ERROR");
        populateClassroomInformation("", "", "", "");
    }

    private void populateClassroomInformation(string name, string floor, string type,
string equipament) {
        populateRoomImage();
        classroomName.GetComponent<Text>().text = name;
        populateRoomTitle(name);
        classroomFloor.GetComponent<Text>().text = floor;
        classroomType.GetComponent<Text>().text = type;
        classroomEquipment.GetComponent<Text>().text = equipament;
    }

    private void showSchudeleButton(ClassroomData classroom) {
        showSchudeleButton(classroom.hasSchudele());
    }

    private void showSchudeleButton(bool visible) {
        schudeleButton.SetActive(visible);
    }
}
```

En el cuadrado rojo, aparecen las variables públicas de la clase, son todas las zonas donde añadiremos la información del aula consultada.

El cuadrado verde, muestra el método que se ejecuta al mostrar el panel con la información de aulas. Este método recupera el *ClassroomData* asociado a la variable *roomID* de *GameRoomManager*. Si dispone de información la introduce en el UI y determina si ha de mostrar el botón del horario. En caso de no disponer de información, mostrará el *sprite* de ERROR, dejará la información vacía y ocultará el botón que permite mostrar el horario.

En el cuadrado morado, vemos la definición del método abstracto para obtener el folder de *Resources* con información de las aulas.

Los cuadrados naranjas, definen los métodos auxiliares que emplea la clase para mostrar u ocultar la información del aula.

Antes de continuar con los scripts de las otras tipologías de habitaciones, vamos a ver los scripts existentes para la visualización de los horarios del aula.

ROOM SCHUDELE BUTTON

Clase que permite modelar el comportamiento del botón que muestra la información del horario del aula.

```
using UnityEngine;
using UnityEngine.UI;

public class RoomSchudeleButton : MonoBehaviour {
```

```
    public GameObject classroomContainerGameObjects;
    public GameObject schudeleContainerGameObjects;
    public GameObject firstSemesterButton;
    public GameObject schudeleButtonlabel;

    private bool active = false;

    private void OnEnable() {
        active = false;
        disableSchudele();
    }

    public void showSchudele() {
        active = !active;
        if (active)
            enableSchudele();
        else
            disableSchudele();
    }

    public void disableSchudele() {
        classroomContainerGameObjects.SetActive(true);
        schudeleContainerGameObjects.SetActive(false);
        schudeleButtonlabel.GetComponent<Text>().text = "MOSTRAR HORARIO";
    }

    public void enableSchudele() {
        classroomContainerGameObjects.SetActive(false);
        schudeleContainerGameObjects.SetActive(true);
        schudeleButtonlabel.GetComponent<Text>().text = "OCULTAR HORARIO";

firstSemesterButton.GetComponent<RoomSchudeleSemesterButton>().selectSemester();
    }
}
```

En el cuadrado rojo, aparecen las variables públicas de la clase.

- **ClassroomContainerGameObjects:** Panel que contiene la información del aula.
- **ClassroomContainerGameObjects:** Panel que contiene la información del horario.
- **FirstSemesterButton:** Botón que muestra el horario del primer cuatrimestre.
- **SchudeleButtonlabel:** Texto del botón de mostrar horario.

En el cuadrado azul, se define la variable privada de la clase, esta define si el botón ha sido pulsado y está mostrando la información del horario.

El cuadrado verde, muestra el método que se ejecuta al mostrar el botón, donde se pone la variable active a *false* y se oculta el panel que contiene la información del horario.

En el cuadrado morado, vemos el método que utilizará el botón de mostrar/ocultar el horario. Al pulsar el botón modificará el valor de la variable *active* y en función de su valor resultante mostrará u ocultará el horario. Para ello, hará uso de los métodos auxiliares que dispone la clase, estos ocultan o muestran los paneles y modifican el texto del botón. En caso de mostrar la información del horario, ejecutará el método *selectSemester* del botón asociado al primer cuatrimestre.

ROOM SCHUDELE SEMESTER BUTTON

Clase que permite modelar el comportamiento del botón que muestra la información del horario del aula de un semestre u otro.

```
using System;
using UnityEngine;
using UnityEngine.UI;

public class RoomSchudeleSemesterButton : MonoBehaviour {

    public GameObject schudeleGameObject;

    private Sprite schudeleSprite;

    void OnEnable() {
        schudeleSprite =
Resources.Load<Sprite>(String.Format("Sprites/Aulas/Horarios/{0}/{1}",
GameRoomManager.getRoomID(), name));
    }

    public void selectSemester() {
        schudeleGameObject.GetComponent<Image>().sprite = schudeleSprite;
        schudeleGameObject.SetActive(true);
    }

}
```

En el cuadrado rojo, aparece la variable pública de la clase, esta hace referencia al elemento que contendrá el *sprite* del horario.

En el cuadrado azul, se define la variable privada de la clase, esta define el *sprite* que dispone de la información del horario de la clase.

El cuadrado verde, muestra el método que se ejecuta al mostrar el botón, este obtiene el valor del sprite que mostrará.

En el cuadrado morado, vemos el método que utilizará el botón. Al pulsar el botón modificará el valor del componente *sprite* con el valor almacenado en la variable *schudeleSprite*. Finalmente mostrará el game object.

ROOM OFFICE TABLE INIT

Clase que permite mostrar la información de un despacho en el *PanelRoom.* Esta clase hereda de *RoomAbstractTableInit.*

```
using System;
using UnityEngine;
using UnityEngine.UI;
```

```
public class RoomOfficeTableInit : RoomAbstractTableInit {

    public GameObject officeFloor;
    public GameObject officeType;
    public GameObject officeMoreInformationButton;
    public GameObject teacherFirstname;
    public GameObject teacherLastname;
    public GameObject teacherEmail;
    public GameObject teacherType;
    public GameObject teacherDepartament;
    public GameObject teacherDocentUnit;
    public GameObject teacherDepartamentImage;

    private Sprite teacherDepartamentSprite;

    void OnEnable() {
        OfficeData office =
GameRoomManager.getOfficeByID(GameRoomManager.getRoomID());
        if (office != null) {
            populateOfficeInformation(office);
        } else {
            removeOfficeInformation();
        }
    }

    protected override string getFolder() {
        return "Sprites/Despachos/{0}";
    }

    private void populateOfficeInformation(OfficeData office){
        setRoomSprite(office.getCode());
        setDepartamentSprite(office.getDepartamentCode());
        populateOfficeInformation(office.getName(), office.getFirstname(),
office.getLastname(), office.getEmail(), office.getTypeTeacher(),
office.getDepartament(), office.getDocentUnit(), office.getUrl(), office.getFloor(),
office.getType());
    }

    private void removeOfficeInformation() {
        setRoomSprite("ERROR");
        setDepartamentSprite(null);
        populateOfficeInformation("", "", "", "", "", "", "", "", "", "");
    }

    private void setDepartamentSprite(string departamentCode) {
        if (departamentCode != null && departamentCode != "" && departamentCode !=
"NA" && departamentCode != "NC") {
            teacherDepartamentSprite =
Resources.Load<Sprite>(String.Format("Sprites/Comunes/Departamentos/{0}",
departamentCode));
        } else {
            teacherDepartamentSprite = Resources.Load<Sprite>("Sprites/Comunes/UPV");
        }
    }
```

```
    private void populateOfficeInformation(string name, string firstname, string
lastname, string email, string typeTeacher, string departament, string docentUnit,
string url, string floor, string type) {
        populateRoomImage();
        populateRoomTitle(name);
        officeFloor.GetComponent<Text>().text = floor;
        officeType.GetComponent<Text>().text = type;
        teacherFirstname.GetComponent<Text>().text = firstname;
        teacherLastname.GetComponent<Text>().text = lastname;
        teacherEmail.GetComponent<Text>().text = email;
        teacherType.GetComponent<Text>().text = typeTeacher;
        teacherDepartament.GetComponent<Text>().text = departament;
        teacherDocentUnit.GetComponent<Text>().text = docentUnit;
        teacherDepartamentImage.GetComponent<Image>().sprite =
teacherDepartamentSprite;

officeMoreInformationButton.GetComponent<RoomShowMoreInformationButton>().setUrl(url)
;
        officeMoreInformationButton.SetActive(url != "" && url != null);
    }
}
```

En el cuadrado rojo, aparecen las variables públicas de la clase, son todas las zonas donde añadiremos la información del despacho consultado.

El cuadrado azul, indica la variable privada de la clase, hace referencia al *sprite* con la imagen del logo del departamento al que pertenece el profesor.

El cuadrado verde, muestra el método que se ejecuta al mostrar el panel con la información de despachos. Este método recupera el *OfficeData* asociado a la variable *roomID* de *GameRoomManager*. Si dispone de información la introduce en el UI. En caso de no disponer de información, mostrará el *sprite* de ERROR y dejará la información vacía.

En el cuadrado morado, vemos la definición del método abstracto para obtener el folder de *Resources* con información de los profesores.

Los cuadrados naranjas, definen los métodos auxiliares que emplea la clase para mostrar u ocultar la información del despacho. A la hora de mostrar el sprite del logo del departamento, comprueba que tiene valor y es distinto de NA y NC, en ese caso muestra el sprite del logo de la UPV. Debemos destacar que es el encargado de definir la URL para el botón de más información y determina si es visible o no.

Antes de continuar con los scripts de las otras tipologías de habitaciones, vamos a ver el script existente para la visualización de más información.

ROOM SHOW MORE INFORMATION BUTTON

Clase que permite modelar el comportamiento del botón que muestra más información de una habitación. El botón permite redirigir al navegador y mostrar allí una web.

```
using UnityEngine;

public class RoomShowMoreInformationButton : MonoBehaviour {

    private string url;

    public void goToUrl() {
        Application.OpenURL(url);
    }

    public void setUrl(string newUrl) {
        url = newUrl;
    }

}
```

En el cuadrado rojo, aparece la variable pública de la clase, hace referencia a la URL a la que redirigirá al usuario de la aplicación. El valor de esta variable será definido mediante el método público existente en el cuadrado verde.

El cuadrado morado, define el método que empleará el botón, este empleará el método *OpenURL*, tal y como ya vimos en el *StartPanel*.

ROOM SEMINARY TABLE INIT

Clase que permite mostrar la información de un seminario en el *PanelRoom.* Esta clase hereda de *RoomAbstractTableInit.*

```
using System;
using UnityEngine;
using UnityEngine.UI;

public class RoomSeminaryTableInit : RoomAbstractTableInit {

    public GameObject seminaryName;
    public GameObject seminaryFloor;
    public GameObject seminaryDepartament;
    public GameObject seminaryDocentUnit;
    public GameObject seminaryEquipment;
    public GameObject seminaryDepartamentImage;

    private Sprite seminaryDepartamentSprite;

    void OnEnable() {
        SeminaryData seminary =
GameRoomManager.getSeminaryByID(GameRoomManager.getRoomID());
        if (seminary != null) {
            populateSeminaryInformation(seminary);
        } else {
            removeSeminaryInformation();
        }
    }

    protected override string getFolder() {
        return "Sprites/Seminarios/{0}";
    }
```

```
    private void populateSeminaryInformation(SeminaryData seminary){
        setRoomSprite(seminary.getCode());
        setDepartamentSprite(seminary.getDepartamentCode());
        populateSeminaryInformation(seminary.getName(), seminary.getFloor(),
seminary.getDepartament(), seminary.getDocentUnit(), seminary.getEquipament(),
seminary.getDepartamentCode());
    }

    private void removeSeminaryInformation() {
        setRoomSprite("ERROR");
        setDepartamentSprite(null);
        populateSeminaryInformation("", "", "", "", "", "");
    }

    private void setDepartamentSprite(string departamentCode) {
        if (departamentCode != null && departamentCode != "" && departamentCode !=
"NA" && departamentCode != "NC") {
            seminaryDepartamentSprite =
Resources.Load<Sprite>(String.Format("Sprites/Comunes/Departamentos/{0}",
departamentCode));
        } else {
            seminaryDepartamentSprite =
Resources.Load<Sprite>("Sprites/Comunes/UPV");
        }
    }

    private void populateSeminaryInformation(string name, string floor, string
departament, string docentUnit, string equipament, string departamentCode) {
        populateRoomImage();
        seminaryName.GetComponent<Text>().text = name;
        populateRoomTitle(name);
        seminaryFloor.GetComponent<Text>().text = floor;
        seminaryDepartament.GetComponent<Text>().text = departament;
        seminaryDocentUnit.GetComponent<Text>().text = docentUnit;
        seminaryEquipment.GetComponent<Text>().text = equipament;
        seminaryDepartamentImage.GetComponent<Image>().sprite =
seminaryDepartamentSprite;
    }
}
```

En el cuadrado rojo, aparecen las variables públicas de la clase, son todas las zonas donde añadiremos la información del seminario consultado.

El cuadrado azul, indica la variable privada de la clase, hace referencia al *sprite* con la imagen del logo del departamento al que pertenece el seminario.

El cuadrado verde, muestra el método que se ejecuta al mostrar el panel con la información de seminarios. Este método recupera el *SeminaryData* asociado a la variable *roomID* de *GameRoomManager*. Si dispone de información la introduce en el UI. En caso de no disponer de información, mostrará el *sprite* de ERROR y dejará la información vacía.

En el cuadrado morado, vemos la definición del método abstracto para obtener el folder de *Resources* con información de los seminarios.

El cuadrado naranja, define los métodos auxiliares que emplea la clase para mostrar u ocultar la información del seminario. Dispone de la misma lógica mostrada en despachos con respecto al icono del departamento.

ROOM SERVICE TABLE INIT

Clase que permite mostrar la información de un servicio en el *PanelRoom.* Esta clase hereda de *RoomAbstractTableInit.* Esta clase la emplearemos tanto para elementos de la tabla SERVICIOS, FUNDACION y DICGF.

```
using UnityEngine;
using UnityEngine.UI;

public class RoomServiceTableInit : RoomAbstractTableInit {

    public GameObject serviceName;
    public GameObject serviceFloor;
    public GameObject serviceType;
    public GameObject serviceOpenHours;
    public GameObject serviceDescription;
    public GameObject serviceMoreInformation;
    public GameObject serviceMoreInformationButton;

    void OnEnable() {
        ServiceData service = null;
        string roomImageName = "UPV";
        switch (GameRoomManager.getRoomTable()) {
            case RoomTable.Servicios:
                roomImageName = "UPV";
                service =
GameRoomManager.getServiceByID(GameRoomManager.getRoomID());
                break;
            case RoomTable.Fundacion:
                roomImageName = "FUN";
                service = GameRoomManager.getFunByID(GameRoomManager.getRoomID());
                break;
            case RoomTable.DICGF:
                roomImageName = "DICGF";
                service = GameRoomManager.getDICGFByID(GameRoomManager.getRoomID());
                break;
        }
        setRoomSprite(roomImageName);
        if (service != null) {
            populateServiceInformation(service);
        } else {
            removeServiceInformation();
        }
    }

    protected override string getFolder() { return "Sprites/Comunes/{0}"; }

    private void populateServiceInformation(ServiceData seminary){
        populateServiceInformation(seminary.getName(), seminary.getFloor(),
seminary.getType(), seminary.getOpenHoursValue(), seminary.getDescription(),
seminary.getMoreInformation(), seminary.getUrl());
    }
```

```
    private void removeServiceInformation() {
        populateServiceInformation("", "", "", "", "", "", "");
    }

    private void populateServiceInformation(string name, string floor, string type,
string openHours, string description, string moreInformation, string url) {
        populateRoomImage();
        serviceName.GetComponent<Text>().text = name;
        populateRoomTitle(name);
        serviceFloor.GetComponent<Text>().text = floor;
        serviceType.GetComponent<Text>().text = type;
        serviceOpenHours.GetComponent<Text>().text = openHours;
        serviceDescription.GetComponent<Text>().text = description;
        serviceMoreInformation.GetComponent<Text>().text = moreInformation;

serviceMoreInformationButton.GetComponent<RoomShowMoreInformationButton>().setUrl(url
);
        serviceMoreInformationButton.SetActive(url != "" && url != null);
    }
}
```

En el cuadrado rojo, aparecen las variables públicas de la clase, son todas las zonas donde añadiremos la información del servicio consultado.

El cuadrado verde, muestra el método que se ejecuta al mostrar el panel con la información de seminarios. Este método recupera el *ServiceData* asociado a la variable *roomID* de *GameRoomManager*, en función del RoomTable empleará un método u otro para obtener la información. Si dispone de información la introduce en el UI. En caso de no disponer de información, dejará la información vacía. La imagen a mostrar en esta pantalla será un logo, este dependerá de la tipología de la habitación.

En el cuadrado morado, vemos la definición del método abstracto para obtener el folder de *Resources* con información de los servicios.

Los cuadrados naranjas, definen los métodos auxiliares emplea la clase para mostrar u ocultar la información del servicio. Debemos destacar que es el encargado de definir la URL para el botón de más información y determina si es visible o no.

ASOCIANDO LOS SCRIPTS AL UI

Una vez definidos los scripts, los asociamos a los distintos paneles y botones, al igual que vimos en el StartPanel.

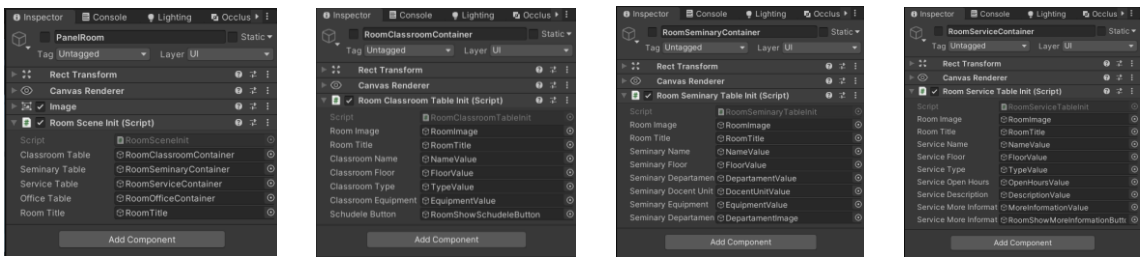


Fig. 101. Asociando scripts al PanelRoom.

4.4. EXTENSIÓN MAP

En esta extensión se van a definir todos los scripts que disponemos para añadir la funcionalidad del mapa en el proyecto.

4.4.1. GAME MAP MANAGER

Encargada de disponer de variables y métodos necesarios para gestionar la información de la planta en la que se encuentra el usuario y de la información del teletransporte consultado.

```
public static class GameMapManager {

    //CONSTANTS
    public static float FLOOR_0_Y_COORDINATE = 11425.5f;
    public static float FLOOR_1_Y_COORDINATE = 11430f;
    public static float FLOOR_2_Y_COORDINATE = 11434.5f;
    public static float FLOOR_3_Y_COORDINATE = 11439f;
    public static float FLOOR_4_Y_COORDINATE = 11443.5f;
    public static float ROOF_Y_COORDINATE = 114485f;
    public static float MINI_MAP_Y_COORDINATE = 11452.82f;
    //VARIABLES
    public static int floor = 0;
    public static bool teleportActive = false;
    public static string nameTeleport = null;

    public static bool isTeleportInUse(string name) {
        return name == getNameTeleport();
    }

    public static bool isPlayerInFloor(int floorToCheck){
        return floorToCheck == getFloor();
    }

    public static bool isTeleportActive() {
        return teleportActive;
    }

    public static void setTeleport(bool active) {
        teleportActive = active;
    }

    public static int getFloor() {
        return floor;
    }

    public static void setFloor(int newFloor) {
        floor = newFloor;
    }

    public static string getNameTeleport() {
        return nameTeleport;
    }

    public static void setNameTeleport(string name) {
        nameTeleport = name;
    }}
```

En el cuadrado rojo, se muestran las constantes que definen las cotas de las distintas plantas.

El cuadrado azul, contiene las variables que contendrán la información de la planta en la que se encuentra el usuario y la información del punto de teletransporte consultado.

En el cuadrado verde, dispone de los métodos que permiten editar y consultar la información de las variables del manager.

4.4.2. MAP PLAYER CONTROLLER

En esta clase vamos a definir una serie de comportamientos referentes a la planta en la en la que se encuentra el player, así como acciones que puede realizar.

```
using System.Collections;
using UnityEngine;
using UnityEngine.UI;

public class MapPlayerController : MonoBehaviour {

    public GameObject panelMap;

    public float refresh_time = 0.5f;
    public GameObject decorationP0;
    public GameObject decorationP1;
    public GameObject decorationP2;
    public GameObject decorationP3;
    public GameObject decorationP4;
    public GameObject mapParking;
    public GameObject mapP0;
    public GameObject mapP1;
    public GameObject mapP2;
    public GameObject mapP3;
    public GameObject mapP4;
    public GameObject message;

    void OnEnable() {
        loadFloor();
        StartCoroutine(decorationController()); }

    void Update () {
        if (Input.GetKey("m") && GamePositionManager.isInGame()) {
            GameMapManager.setTeleport(false);
            panelMap.gameObject.SetActive(true);
            GameMapManager.setNameTeleport(null);
    }}

    private void loadFloor() {
        changeFloorLabel();
        decorationShow(true);
    }

    void changeFloorLabel() {
        int floor = GameMapManager.getFloor();
        message.GetComponent<Text>().text = floor == -1 ? "Aparcamiento" : "Planta "
+ floor;
        message.GetComponent<Text>().fontSize = 15;}
```

```
    void decorationShow(bool status) {
        switch (GameMapManager.getFloor()) {
            case -1:
                mapParking.SetActive(status);
                break;
            case 0:
                decorationP0.SetActive(status);
                mapP0.SetActive(status);
                break;
            case 1:
                decorationP1.SetActive(status);
                mapP1.SetActive(status);
                break;
            case 2:
                decorationP2.SetActive(status);
                mapP2.SetActive(status);
                break;
            case 3:
                decorationP3.SetActive(status);
                mapP3.SetActive(status);
                break;
            case 4:
                decorationP4.SetActive(status);
                mapP4.SetActive(status);
                break;
    }}

    private void changeFloorTo(int floor) {
        decorationShow(false);
        GameMapManager.setFloor(floor);
        loadFloor();
    }
```

```
    IEnumerator decorationController() {
        while (true) {
            int floor = GameMapManager.getFloor();
            float y = transform.position.y;
            if (y <= GameMapManager.FLOOR_0_Y_COORDINATE && floor != -1)
                changeFloorTo(-1);
            else if (y > GameMapManager.FLOOR_0_Y_COORDINATE && y <=
GameMapManager.FLOOR_1_Y_COORDINATE && floor != 0)
                changeFloorTo(0);
            else if (y > GameMapManager.FLOOR_1_Y_COORDINATE && y <=
GameMapManager.FLOOR_2_Y_COORDINATE && floor != 1)
                changeFloorTo(1);
            else if (y > GameMapManager.FLOOR_2_Y_COORDINATE && y <=
GameMapManager.FLOOR_3_Y_COORDINATE && floor != 2)
                changeFloorTo(2);
            else if (y > GameMapManager.FLOOR_3_Y_COORDINATE && y <=
GameMapManager.FLOOR_4_Y_COORDINATE && floor != 3)
                changeFloorTo(3);
            else if (y > GameMapManager.FLOOR_4_Y_COORDINATE && y <=
GameMapManager.ROOF_Y_COORDINATE && floor != 4)
                changeFloorTo(4);
            else if (y > GameMapManager.ROOF_Y_COORDINATE)
                decorationShow(false);
            yield return new WaitForSeconds(refresh_time);
    }}
}
```

El cuadrado azul, muestra la variable donde se instanciará el panel que mostrará la información del mapa.

En el cuadrado rojo, disponemos de las variables que contendrán elementos del modelado que se mostrarán u ocultarán en función de la planta en la que se encuentre el usuario. Las primeras, harán referencia los grupos de elementos de decoración de las habitaciones existentes en la planta, cabe destacar que los elementos decorativos de los pasillos serán excluidos, ya que se notaría más su carga al cambiar entre plantas. Las segundas, son las correspondientes a los distintos mapas que se tienen en el mini mapa.

El cuadrado verde, contiene la función *OnEnable* de la clase. Esta se encarga de cargar la información de la planta inicial, que comprende de mostrar el mapa correcto en el mini mapa, mostrar la decoración interior de la planta y modificar el *PanelGame* indicando el piso en el que se encuentra el usuario. Finalmente inicia la función de la corrutina, encargada de detectar los cambios de planta del usuario.

En el cuadrado amarillo, aparece la función *Update* de la clase. Esta es la que permitirá al usuario acceder al panel con la información del mapa. Mediante la tecla M, se mostrará el *PanelMap*, almacenado que se ha accedido a él sin la función de teletransporte entre directorios.

Los cuadrados naranjas, definen los métodos auxiliares empleados por la clase para modificar el elemento UI del *PanelGame* con la información del piso en el que se encuentra y la visibilidad de la decoración y de los mapas de mini mapa. Además, se almacenará la planta en la que se encuentra el usuario en la variable *floor* de *GameMapManager*.

En el cuadrado morado, tenemos la función de la corrutina, esta comprueba la posición del usuario en cota y en función de su rango determina la planta en la que se encuentra. Una vez detectada, se encarga de llamar a las funciones auxiliares, encargadas de mostrar el mapa correcto en el mini mapa, mostrar la decoración interior de la planta y modificar el *PanelGame* indicando el piso en el que se encuentra el usuario.

Una vez desarrollado el script, se lo asignaremos al game object *player* y le referenciaremos todas las variables públicas, de la misma forma que se realizó con el controlador visto en la extensión CORE.

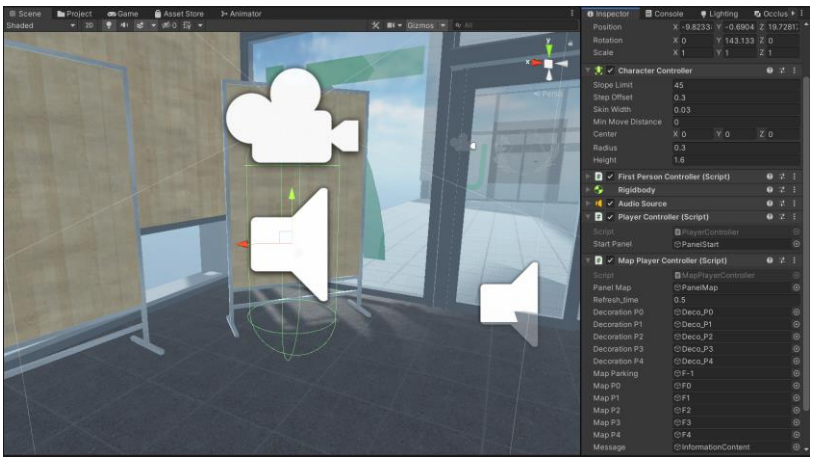


Fig. 102. Asociando el MapPlayerController al player.

4.4.3. CLASES GAME DE LA EXTENSIÓN MAP

En este apartado vamos a ver los scripts creados para elementos del modelo interactivo con referencia a la funcionalidad del mapa que existe en el proyecto. Dispondremos de dos scripts:

I. **FixMovimentInOneAxe:** Destinado a elementos del mini mapa, su función es restringir el movimiento en un eje.
II. **MapTeleport:** Destinado a los directorios del modelo, su función es permitir el acceso al panel del mapa con la funcionalidad de movimiento rápido.

FIX MOVIMENT IN ONE AXE

Esta clase se empleará en los elementos que se muestran en el mini mapa referentes al usuario, los cuales corresponden al sprite del usuario y al indicador de brújula de la misión en curso. La función de este script es restringir el movimiento del objeto en un eje, esto es necesario debido a que estos elementos del mini mapa se desplazan con el usuario, por tanto, al ascender y descender ellos también lo harían, lo que podría provocar que se desplazasen debajo de los sprites de los mapas del mini mapa.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class FixMovimentInOneAxe: MonoBehaviour {

    public AxeDirection axe = AxeDirection.Y;
    private float fixedValue;

    void Start() {
        if (axe == AxeDirection.X)
            fixedValue = transform.position.x;
        if (axe == AxeDirection.Y)
            fixedValue = transform.position.y;
        if (axe == AxeDirection.Z)
            fixedValue = transform.position.z;
    }

    void Update () {
        if (axe == AxeDirection.X)
            transform.position = new Vector3(fixedValue, transform.position.y,
transform.position.z);
        if (axe == AxeDirection.Y)
            transform.position = new Vector3(transform.position.x, fixedValue,
transform.position.z);
        if (axe == AxeDirection.Z)
            transform.position = new Vector3(transform.position.x,
transform.position.y, fixedValue);
    }
}
```

En el cuadrado rojo, disponemos de las variables de la clase. La pública permite definir el eje que queremos restringir. La variable privada permite almacenar el valor inicial de posición en ese eje.

El cuadrado verde, muestra el método start de la clase. Este método almacena el valor inicial de la posición del eje configurado en la variable axe.

En el cuadrado azul, se define el método Update de la clase. Este método reposiciona el game object manteniendo el valor inicial almacenado en fixedValue, para el eje configurado en axe.

MAP TELEPORT

Esta clase está destinada a modelar el comportamiento de los directorios, estos serán elementos que estarán repartidos por las distintas plantas del edificio. Estos objetos permitirán al usuario interactuar con ellos, mostrándoles el mapa del edificio, con la particularidad de que mostrará la ubicación de los demás directorios, para poder moverse de uno a otro de forma inmediata.

```
using System.Collections;
using UnityEngine;
using UnityEngine.UI;

public class MapTeleport : MonoBehaviour {

    public GameObject player;
    public GameObject panelMap;
    public GameObject message;

    private bool informationShow;

    IEnumerator showInformation() {
        while (informationShow) {
            if (Input.GetKeyDown("f") && GamePositionManager.isInGame()) {
                GameMapManager.setTeleport(true);
                GameMapManager.setNameTeleport(name);
                addMessageInfo("");
                panelMap.gameObject.SetActive(true);
            }
            yield return null;
        }
    }

    private void OnTriggerEnter(Collider other) {
        informationShow = true;
        StartCoroutine(showInformation());
        addMessageInfo("Presiona 'F' para usar el teletransporte");
    }

    private void OnTriggerExit(Collider other) {
        informationShow = false;
        StopCoroutine(showInformation());
        addMessageInfo("");
    }

    private void addMessageInfo(string messageValue) {
        message.GetComponent<Text>().text = messageValue;
    }
}
```

En el cuadrado rojo, aparecen las variables públicas de la clase, en ellas se determina el jugador, el panel que mostrará la información del mapa y el elemento UI que se dispone para interactuar con los directorios.

El cuadrado azul, define la variable privada de la clase, esta será la encargada de indicar si el usuario es capaz de interactuar con los directorios.

En el cuadrado verde, se observan los métodos que tendrán en cuenta la interacción con los triggers, estos iniciarán o pararán la corrutina y mostrarán u ocultarán el texto de interacción con los directorios.

El cuadrado naranja, dispone del método auxiliar que empleará la clase para modificar el texto de interacción con el directorio.

En el cuadrado morado, se muestra el método de la corrutina, el cual permitirá al usuario interactuar con el directorio mediante la tecla F. Al presionarla guardará la información de que se ha empleado un directorio y su id. Finalmente mostrará el panel con la información del mapa.

ADICCIÓN Y CONFIGURACIÓN DE LOS GAME OBJECTS

Al igual que en la extensión ROOM, hemos de añadir estos scripts en los game objects, para así disponer de las funcionalidades programadas. Para el caso de los directorios, modificaremos su PREFAB para disponer directamente de los cambios en todas sus instancias.

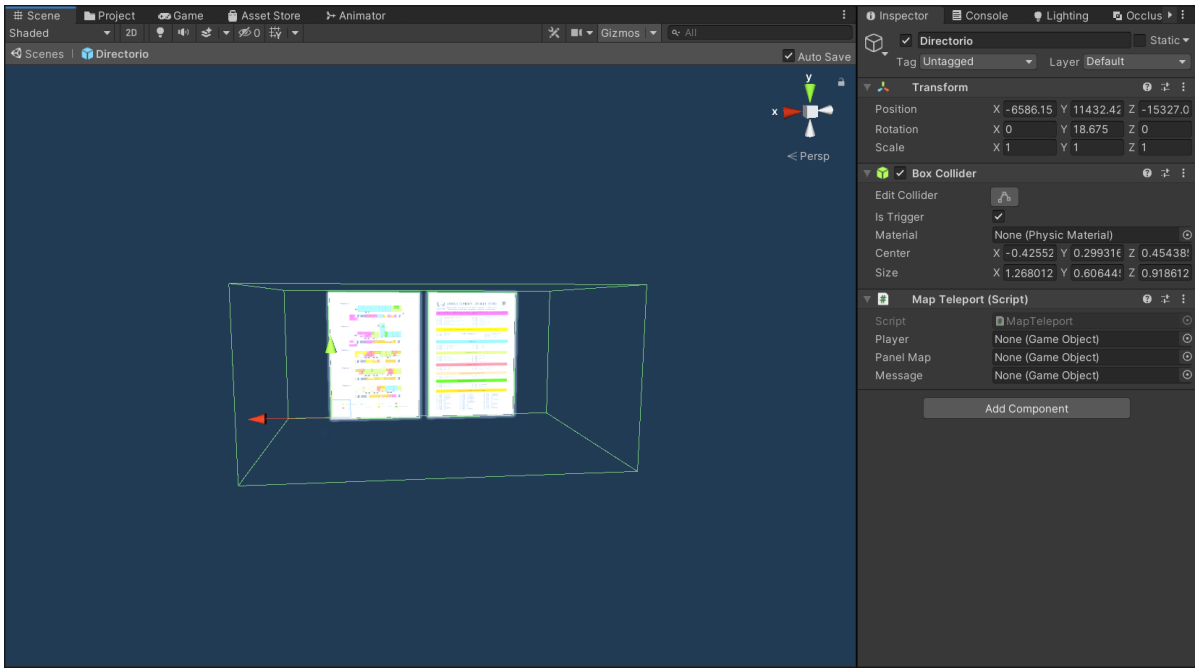


Fig. 103. Configurando los PREFABS de los directorios.

4.4.1. CLASES UI DE LA EXTENSIÓN MAP

En este apartado vamos a ver las clases creadas para dar la funcionalidad al *PanelMap*.

MAP SCENE INIT

Clase que se asignará directamente al *PanelMap*, está será la encargada mostrar el panel correspondiente al piso que se encuentre el usuario.

```
using System;
using System.Collections.Generic;
using UnityEngine;

public class MapSceneInit : MonoBehaviour {

    public List<GameObject> teleports;
    public List<GameObject> floorButtons;
    private GameObject player;

    void OnEnable() {
        player = GameObject.Find("Player");
        player.GetComponent<PlayerController>().showCursor();
        stopTime(true);
        changeTeleportPointsVisibility();
        selectFloor();
    }

    public void closeMap() {
        stopTime(false);
        player.GetComponent<PlayerController>().hideCursor();
        gameObject.SetActive(false);
    }

    private void stopTime(bool stop) {
        Time.timeScale = stop ? 0 : 1;}

    private void changeTeleportPointsVisibility() {
        bool teleportActive = GameMapManager.isTeleportActive();
        foreach(GameObject teleport in teleports) {
            teleport.SetActive(teleportActive);
        }
    }

    private void selectFloor() {
        string floorButtonToSelectName = String.Format("MapFloorButton{0}",
GameMapManager.getFloor());
        foreach (GameObject floorButton in floorButtons) {
            if (floorButton.name == floorButtonToSelectName) {
                floorButton.GetComponent<MapFloorButton>().selectFloor();
                break;
            }
        }
    }
}
```

En el cuadrado rojo, aparecen las variables públicas y privadas de la clase. En las variables púbicas, disponemos de los puntos de teletransporte y de los botones de cambio de planta. En la variable privada, tenemos al game object del *player*.

El cuadrado azul, define el método que se empleará para cerrar el *PanelMap.* En él se reanuda el tiempo del juego y se oculta el cursor y el panel.

En el cuadrado verde, tenemos el método que se activa cuando el panel se hace visible. Este obtiene al player, muestra el cursor, detiene el tiempo de juego, muestra si son necesarios los teletransportes y activa el mapa del piso actual.

El cuadrado naranja, dispone de los métodos auxiliares de la clase. El primero, permite modificar el tiempo de juego. El segundo permite controlar la visibilidad de los teletransportes en función al método *isTeleportActive* existente en *GameMapManager.* El segundo, permite activar el botón del piso en el que se encuentra el usuario, para ello se apoya en el método *getFloor*.

MAP FLOOR BUTTON

Esta clase permite dar funcionalidad a los botones que permiten seleccionar las distintas plantas del edificio. Su finalidad será controlar la visibilidad de los distintos paneles referentes a los pisos existentes en la escuela.

```
using System;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class MapFloorButton : MonoBehaviour {

    public List<GameObject> mapFloors;
    public GameObject mapFloorToSelect;
    public string mapTitleFloorText;
    public GameObject mapTitleFloorLabel;

    public void selectFloor() {
        showMap();
        changeTitleFloorLabel();
    }

    private void showMap() {
        foreach (GameObject mapFloor in mapFloors) {
            mapFloor.SetActive(mapFloor == mapFloorToSelect);
        }
    }

    private void changeTitleFloorLabel() {
        mapTitleFloorLabel.GetComponent<Text>().text = mapTitleFloorText;
    }
}
```

En el cuadrado rojo, se muestran las variables públicas de la clase.

- **MapFloors:** Lista de paneles que contienen el mapa de las distintas plantas
- **MapFloorToSelect:** Panel que contiene el mapa con la planta del botón.
- **MapTitleFloorText:** Texto que mostrará en el título del *PanelMap*.
- **MapTitleFloorLabel:** Elemento UI que donde se mostrará el título del *PanelMap.*

El cuadrado naranja, contiene métodos auxiliares que empleará la clase. El primero, controla la visibilidad del mapa a mostrar. El segundo, actualiza el título mostrado en el *PanelMap.*

En el cuadrado verde, se define el método que realizará el botón. Este se apoya en los métodos auxiliares, permitiendo mostrar el mapa y actualizar el título del *PanelRoom*.

MAP TELEPORT ACTIVE

Esta clase será añadida a los sprites de los teletransportes mostrados en el mapa. Permitirán al usuario desplazarse al directorio seleccionado.

```
using UnityEngine;
using UnityEngine.UI;

public class MapTeleportActive : MonoBehaviour {
    public GameObject mapPanel;
    public GameObject teleport;
    private GameObject player;

    private void OnEnable() {
        player = GameObject.Find("Player");
        GetComponent<Button>().interactable = !GameMapManager.isTeleportInUse(name);
    }

    public void useTeleport() {
        Vector3 teleportPosition = teleport.transform.position;
        GamePositionManager.setPlayerCoordinates(teleportPosition);
        player.GetComponent<PlayerController>().moveToPosition(teleportPosition);
        mapPanel.GetComponent<MapSceneInit>().closeMap();
    }
}
```

En el cuadrado rojo, se muestran las variables de la clase.

.

- **MapPanel:** Texto que mostrará en el título del PanelRoom.
- **Teleport:** Directorio donde queremos que se desplace el usuario.
- **Player:** Game object que hace referencia al usuario.

El cuadrado verde, contiene el método que se ejecuta al hacer visible el game object asociado, este determinará si el directorio es válido para teletransportarse, en caso de ser por el que se ha accedido, no será válido. Este método también será el encargado de rellenar la variable *player.*

En el cuadrado azul, se define el método que realizará el botón. Esta llama al método *useTeleport* de *MapSceneInit* con las coordenadas del directorio.

MOSTRAR CORRECTAMENTE AL USUARIO

Debido a que las coordenadas del player son diferentes en función del tiempo, estas serán diferentes dependiendo de cuando abrió el mapa. Es por ello, que necesitamos transformar su posición en el modelo a su posición en el UI. Para ello, emplearemos los directorios como puntos de control, a partir de ellos posicionaremos al usuario. Veamos los sistemas que disponemos.

- El sistema de coordenadas del Player, el cual viene definido por *PlayerContent*.
- Los sistemas de coordenadas de los directorios, los cuales vienen definidos en cada planta por el game object que contiene los directorios.
- El sistema de coordenadas del UI, el cuál es diferente para cada planta, debido a que el tamaño de los sprites es diferente en cada una, pero el espacio del UI es el mismo. Por lo tanto, un punto en el espacio 3D, tendrá un posicionamiento diferente en función de la planta donde se quiera mostrar.

Para poder posicionar correctamente al usuario, debemos poder relacionar varios sistemas de coordenadas, para ello se han realizado los siguientes procesos:

- Hemos hecho coincidir todos estos sistemas 3D. Definiendo el mismo origen y orientación para todos. Permitiendo comparar las coordenadas entre los distintos sistemas. El origen se ha posicionado en un lugar visible en todos los sprites 2D.
- Hemos posicionado el origen del sistema 3D en cada uno de los sprites de cada planta, permitiendo así conocer en cada uno de los sistemas UI la posición del sistema 3D.

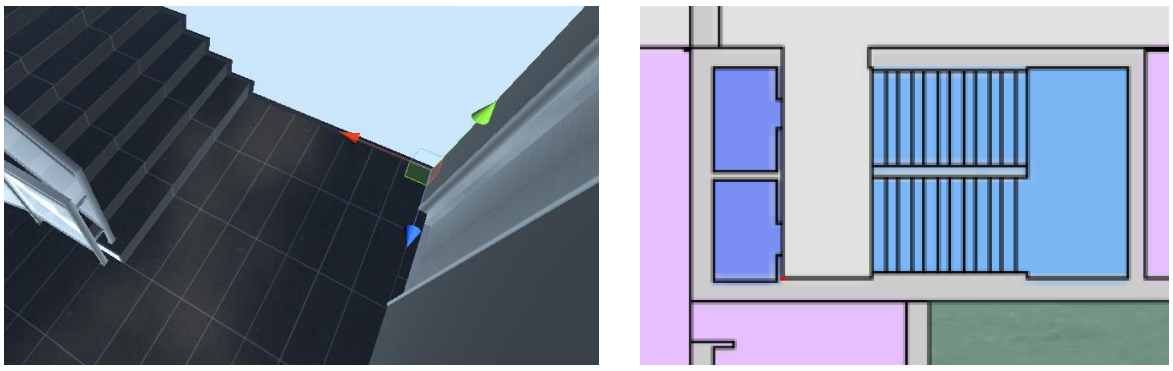


Fig. 104. Eje de coordenadas del player en el espacio 3D y 2D.

Una vez disponemos de los elementos configurados, podemos ya trabajar con ellos en un script que permita posicionar al usuario en el mapa.

Vamos a generar un script genérico que permita posicionar cualquier objeto en el mapa, siempre que este se encuentre en el sistema 3D definido anteriormente. Posteriormente realizaremos una implementación concreta del usuario, la cual se apoyará en la lógica genérica.

```
using System;
using System.Collections.Generic;
using UnityEngine;

public abstract class MapShowIconFloor : MonoBehaviour {

    public int mapFloor;
    public GameObject mapIcon;
    public List<GameObject> mapRefereceIcons;
    public GameObject map3dOriginCoordIcon;

    private Vector2 map3dOriginCoordPosition;

    void Start() {
        map3dOriginCoordPosition =
map3dOriginCoordIcon.GetComponent<RectTransform>().anchoredPosition;}

    void OnEnable() {
        Vector2 mapIconGamePosition = obtainMapIconGamePosition();
        if (isVisibeMapIconFloor() && mapIconGamePosition != null) {
            updatePositionMapIcon(mapIconGamePosition);
            mapIcon.SetActive(true);
        } else {
            mapIcon.SetActive(false);
    }}

    protected abstract bool isVisibeMapIconFloor();

    protected abstract Vector2 obtainMapIconGamePosition();

    protected virtual void updatePositionMapIcon(Vector2 mapIconGamePosition) {
        GameObject mapRefereceIcon = getNearMapReferenceIcon(mapIconGamePosition);
        if (mapRefereceIcon != null) {
            Vector2 mapReferecePosition =
getMapRefereceIconMapPosition(mapRefereceIcon);
            Vector2 mapRefereceGamePosition =
mapRefereceIcon.GetComponent<MapTeleportActive>().teleportLocalPosition;
            mapIcon.GetComponent<RectTransform>().anchoredPosition =
calculateMapIconPosition(mapReferecePosition, mapRefereceGamePosition,
mapIconGamePosition);
    }}

    private GameObject getNearMapReferenceIcon(Vector2 mapIconGamePosition) {
        GameObject mapReferenceIcon = null;
        double distance = 99999999999999999999999999.0d;
        foreach (GameObject icon in mapRefereceIcons) {
            double d = calculateDistance(mapIconGamePosition,
icon.GetComponent<MapTeleportActive>().teleportLocalPosition);
            if (d < distance || mapReferenceIcon == null) {
                distance = d;
                mapReferenceIcon = icon;
        } }
        return mapReferenceIcon;
    }
```

```
    private double calculateDistance(Vector2 mapIconGamePosition, Vector2
mapReferenceIconPosition) {
        float iX = mapReferenceIconPosition.x - mapIconGamePosition.x;
        float iY = mapReferenceIconPosition.y - mapIconGamePosition.y;
        return Math.Sqrt(Math.Pow(iX, 2) + Math.Pow(iY, 2));
    }

    private Vector2 getMapRefereceIconMapPosition(GameObject mapRefereceIcon) {
        Vector2 mapReferecePosition =
mapRefereceIcon.GetComponent<RectTransform>().anchoredPosition;
        return new Vector2(mapReferecePosition.x - map3dOriginCoordPosition.x,
mapReferecePosition.y - map3dOriginCoordPosition.y);
    }

    private Vector2 calculateMapIconPosition(Vector2 mapReferecePosition, Vector2
mapRefereceGamePosition, Vector2 mapIconGamePosition) {
        Vector2 mapIconPosition = new Vector2((mapIconGamePosition.x *
mapReferecePosition.x) / mapRefereceGamePosition.x, (mapIconGamePosition.y *
mapReferecePosition.y) / mapRefereceGamePosition.y);
        return new Vector2(mapIconPosition.x + map3dOriginCoordPosition.x,
mapIconPosition.y + map3dOriginCoordPosition.y);
    }
}
```

En el cuadrado rojo, se muestran las variables públicas de la clase.

- **MapFloor:** Número que indica la planta a la que hará referencia el script.
- **MapIcon:** Elemento UI que queremos posicionar en el mapa.
- **MapRefereceIcons:** Elementos UI que emplearemos como puntos de referencia.
- **Map3dOriginCoordIcon:** Elemento UI indica la representación del centro del eje de coordenadas 3D.

El cuadrado azul, dispone de la variable privada de la clase y la asignación en el método *Start.* Esta variable contendrá las coordenadas 2D, referentes a la posición de la representación del centro de coordenadas del sistema 3D, correspondiente al panel de la planta al que se le asignará el script.

En el cuadrado morado, aparecerán los métodos a definir en las clases hijas del script. El primero, devolverá si el icono se ha de mostrar en la planta en la que está asociada el script. El segundo, proporcionará las coordenadas del elemento que queremos mostrar en su sistema de coordenadas 3D, sin embargo, no se necesitará la información del eje de altura.

Los cuadrados naranjas, definen los métodos de utilidad de la clase:

- **CalculateDistance:** Método que calcula la distancia entre dos puntos.
- **GetNearMapReferenceIcon:** Método que obtiene el punto de teletransporte más cercano al usuario, sólo emplea los existentes en la planta actual.
- **GetMapRefereceIconMapPosition:** Devuelve las coordenadas UI de un punto de teletransporte con respecto a la representación del eje de coordenadas 3D en el UI.
- **CalculateMapIconPosition:** Encargado de calcular la posición del elemento en el UI. Primero adecuará la escala de los dos sistemas, para ello empleará las coordenadas de

un punto de referencia (punto de teletransporte más cercano), mediante su posición en el modelo y en el UI determinará la posición de usuario. Posteriormente realizará una translación empleando las coordenadas de la representación del centro de coordenadas 3D.

- **UpdatePositionMapIcon:** Este método será el encargado de obtener las coordenadas UI del elemento a partir de sus coordenadas 3D.
  - **I.** Obtendrá el punto de referencia más cercano.
  - **II.** Obtendrá para ese punto sus coordenadas UI en referencia a la representación 2D del origen del eje de coordenadas 3D.
  - **III.** Obtendrá las coordenadas 3D de ese punto de referencia, gracias a la variable existente en su *MapTeleportActive*.
  - **IV.** Calculará la posición del objeto en el mapa mediante el método CalculateMapIconPosition, a partir de sus coordenadas 3D, coordenadas del punto de referencia 3D y coordenadas 2D del punto de referencia en función a la representación 2D del centro del eje de coordenadas 3D.

En el cuadrado verde, se muestra el método que se aplicará al mostrar el mapa de la planta seleccionada. Este obtendrá las coordenadas 3D del elemento a representar en el mapa, comprobará si ha de ser visible o no. En caso de ser visible empleará el método *updatePositionMapIcon* para corregir su posición.

El siguiente paso es definir un script que sea el referente al sprite del usuario, para ello crearemos un script que herede de *MapShowIconFloor* e implemente los métodos abstractos.

```
using System;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class MapShowPlayerIconFloor : MapShowIconFloor {

    public GameObject player;

    protected override bool isVisibeMapIconFloor() {
        return GameMapManager.isPlayerInFloor(mapFloor);
    }

    protected override  Vector2 obtainMapIconGamePosition() {
        return new Vector2(player.transform.localPosition.x,
player.transform.localPosition.z);
    }
}
```

En el cuadrado azul, disponemos de la definición de la clase, donde se puede observar que hereda de la clase *MapShowIconFloor*. Esto permite que herede toda la lógica anteriormente explicada, además obliga a implementar los métodos abstractos que se definieron.

En el cuadrado rojo, aparece la variable pública de la clase. Esta hace referencia al usuario.

En el cuadrado morado, vemos los métodos abstractos a implementar por las clases que heredan de *MapShowIconFloor*. El primero se apoya en el método *isPlayerInFloor*, definido en *GameMapManager*, para indicar si el sprite del usuario ha de mostrarse en el mapa. El segundo, devuelve las coordenadas 2D del usuario, con respecto a eje de coordenadas 3D definido.

ASOCIANDO LOS SCRIPTS AL UI

Una vez definidos los scripts, los asociamos a los distintos paneles y botones, al igual que vimos en el RoomPanel.

4.5. EXTENSIÓN ELEVATOR

En esta sección, vamos a modelar el comportamiento de los ascensores de los edificios. Así como dar funcionalidad al *PanelElevator*.

4.5.1. ENUMERADOS DE LA EXTENSIÓN ELEVATOR

ENUM ELEVATOR TYPE

Esta clase define un enumerador, el cual indica el tipo de ascensor.

```
public enum EnumElevatorType {
    Right,
    Left
}
```

FLOOR

Esta clase define un enumerador, el cual indica el piso del edificio.

```
public enum Floor {
    Aparcamiento,
    PlantaBaja,
    PrimeraPlanta,
    SegundaPlanta,
    TerceraPlanta,
    CuartaPlanta
}
```

4.5.2. GAME ELEVATOR MANAGER

Encargada de disponer de variables y métodos necesarios para gestionar la información del ascensor empleado. Así como disponer de los distintos métodos que permitan rellenar la información de las habitaciones existentes por planta, métodos necesarios para rellenar la información en *ElevatorPanel.*

```
using System;
using System.Collections.Generic;

public static class GameElevatorManager {

    //CONSTANTS
    private static List<string> FIELDS_CLASSROOM_INFO = new List<string>() {
"ID_AULAS", "NOMBRE" };
    private static string TABLE_CLASSROOM = "Aulas";
    private static List<string> FIELDS_OFFICE_MAIN_INFO = new List<string>() {
"ID_DESPACHOS" };
    private static List<string> FIELDS_OFFICE_JOIN_INFO = new List<string>() {
"NOMBRE", "APELLIDOS" };
    private static string FIELD_MAIN_OFFICE = "ID_PERSONAL";
    private static string FIELD_JOIN_OFFICE = "ID";
    private static string TABLE_MAIN_OFFICE = "Despachos";
    private static string TABLE_JOIN_OFFICE = "Personal";
    private static List<string> FIELDS_SEMINARY_INFO = new List<string>() {
"ID_SEMINARIO", "NOMBRE" };
    private static string TABLE_SEMINARY = "Seminarios";
    private static List<string> FIELDS_SERVICE_INFO = new List<string>() {
"ID_SERVICIOS", "NOMBRE" };
    private static List<string> FIELDS_FUN_INFO = new List<string>() { "ID_FUN",
"NOMBRE" };
    private static List<string> FIELDS_DICGF_INFO = new List<string>() { "ID_DICGF",
"NOMBRE" };
    private static string TABLE_SERVICE = "Servicios";
    private static string TABLE_FUN = "Fundacion";
    private static string TABLE_DICGF = "DICGF";
    private static string WHERE_CONDITION = "PLANTA = '{0}'";

    //VARIABLES
    private static EnumElevatorType elevatorType = EnumElevatorType.Right;

    public static EnumElevatorType getElevatorType() {
        return elevatorType;
    }

    public static void setElevatorType(EnumElevatorType newElevatorType) {
        elevatorType = newElevatorType;
    }

    public static List<ClassroomData> getClassroomsFromFloor(string floorValue) {
        List<ClassroomData> result = new List<ClassroomData>();
        string query = GameDatabaseManager.obtainBasicQuery(FIELDS_CLASSROOM_INFO,
TABLE_CLASSROOM, String.Format(WHERE_CONDITION, floorValue));
        List<List<string>> dataList =
GameDatabaseManager.obtainMultipleElements(query);
        if (dataList != null && dataList.Count != 0) {
            foreach (List<string> dataListRow in dataList) {
                if (dataListRow != null && dataListRow.Count != 0) {
                    result.Add(new ClassroomData(dataListRow[0], dataListRow[1]));
                }
            }
        }
        return result;
    }
```

```
    public static List<OfficeData> getOfficeFromFloor(string floorValue) {
        List<OfficeData> result = new List<OfficeData>();
        string query = GameDatabaseManager.obtainJoinQuery(FIELDS_OFFICE_MAIN_INFO,
TABLE_MAIN_OFFICE, FIELD_MAIN_OFFICE, FIELDS_OFFICE_JOIN_INFO, TABLE_JOIN_OFFICE,
FIELD_JOIN_OFFICE, "M." + String.Format(WHERE_CONDITION, floorValue));
        List<List<string>> dataList =
GameDatabaseManager.obtainMultipleElements(query);
        if (dataList != null && dataList.Count != 0) {
            foreach (List<string> dataListRow in dataList) {
                if (dataListRow != null && dataListRow.Count != 0) {
                    result.Add(new OfficeData(dataListRow[0], dataListRow[1],
dataListRow[2]));
        } } }
        return result;
    }

    public static List<SeminaryData> getSerminariesFromFloor(string floorValue) {
        List<SeminaryData> result = new List<SeminaryData>();
        string query = GameDatabaseManager.obtainBasicQuery(FIELDS_SEMINARY_INFO,
TABLE_SEMINARY, String.Format(WHERE_CONDITION, floorValue));
        List<List<string>> dataList =
GameDatabaseManager.obtainMultipleElements(query);
        if (dataList != null && dataList.Count != 0) {
            foreach (List<string> dataListRow in dataList) {
                if (dataListRow != null && dataListRow.Count != 0) {
                    result.Add(new SeminaryData(dataListRow[0], dataListRow[1]));
        } } }
        return result;
    }

    public static List<ServiceData> getServicesFromFloor(string floorValue) {
        List<ServiceData> result = new List<ServiceData>();
        List<ServiceData> services = getServicesFromFloorAndType(floorValue,
RoomTable.Servicios, TABLE_SERVICE, FIELDS_SERVICE_INFO);
        result.AddRange(services);
        List<ServiceData> dicgf = getServicesFromFloorAndType(floorValue,
RoomTable.DICGF, TABLE_DICGF, FIELDS_DICGF_INFO);
        result.AddRange(dicgf);
        List<ServiceData> fundation = getServicesFromFloorAndType(floorValue,
RoomTable.Fundacion, TABLE_FUN, FIELDS_FUN_INFO);
        result.AddRange(fundation);
        return result;}

    private static List<ServiceData> getServicesFromFloorAndType(string floorValue,
RoomTable serviceType, String tableName, List<string> fields) {
        List<ServiceData> result = new List<ServiceData>();
        string query = GameDatabaseManager.obtainBasicQuery(fields, tableName,
String.Format(WHERE_CONDITION, floorValue));
        List<List<string>> dataList =
GameDatabaseManager.obtainMultipleElements(query);
        if (dataList != null && dataList.Count != 0) {
            foreach (List<string> dataListRow in dataList) {
                if (dataListRow != null && dataListRow.Count != 0) {
                    result.Add(new ServiceData(serviceType, dataListRow[0],
dataListRow[1]));
        } } }
        return result;
    }
}
```

En el cuadrado rojo, aparecen las variables privadas necesarias para poder realizar las queries que se emplearán para la obtención de información de las distintas habitaciones existentes por planta del edificio.

El cuadrado azul, muestra la variable elevatorType que contendrá la información del ascensor seleccionado por el usuario. Además, dispone de los métodos necesarios para su obtención y modificación.

En los cuadrados verdes, se definen los métodos que permitirán obtener la información de las distintas habitaciones por planta, donde encontramos un método público por tipología. En el caso de servicios obtendrá la información de las tablas de SERVICIOS, FUNDACIÓN y DICGF.

4.5.3. CLASES GAME DE LA EXTENSIÓN ELEVATOR

En este apartado vamos a ver los scripts creados para elementos del modelo interactivo con referencia a la funcionalidad de los ascensores que existen en el proyecto. Dispondremos de dos scripts:

I. **Elevator:** Encargado de modelar el desplazamiento del ascensor.
II. **ElevatorTrigger:** Encargado de llamar al ascensor asociado y que este comience el desplazamiento mediante el script *Elevator*.

ELEVATOR

Esta clase permite modelar el desplazamiento del ascensor. Permitiendo el desplazamiento vertical del mismo.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class Elevator : MonoBehaviour {
    public float transitionPerSecond = 1.2f;
    public EnumElevatorType elevatorType;
    public GameObject panelAscensor;
    public GameObject fakeDoor;
    public GameObject player;
    public GameObject message;

    private bool isActive;
    private bool finished;
    private Vector3 finalPos;
    private Vector3 initialPos;
    private bool informationShow;
    private List<ElevatorTrigger> triggers;
    private bool isPlayerInside = false;
    private GameObject playerContent;
    private GameObject playerElevatorContent;
```

```
void Start () {
    isActive = false;
    GameObject triggerParent = transform.parent.Find("Triggers").gameObject;
    triggers = new List<ElevatorTrigger>();
    for (int i = 0; i < triggerParent.transform.childCount; i++) {
        triggers.Add(triggerParent.transform.GetChild(i)
                                  .gameObject.GetComponent<ElevatorTrigger>());
    }
    playerContent = GameObject.Find("PlayerContent");
    playerElevatorContent = transform.Find("PlayerElevatorContent").gameObject;
}
```

```
public bool isFinished() { return finished; }

public void callElevator(float ZCoord, bool playerInside) {
    closeDoors();
    fakeDoor.SetActive(true);
    StopCoroutine(transitionElevator());
    initialPos = transform.localPosition;
    finalPos = new Vector3(initialPos.x, initialPos.y, ZCoord);
    isPlayerInside = playerInside;
    if (isPlayerInside)
        player.transform.SetParent(playerElevatorContent.transform);
    StartCoroutine(transitionElevator());
}
```

```
private void closeDoors() {
    foreach (ElevatorTrigger trigger in triggers) {
        trigger.closeDoors();
} }

private IEnumerator transitionElevator() {
    isActive = true;
    finished = false;
    while (isActive) {
        moveElevator(finalPos);
        yield return null;
} }

private void moveElevator(Vector3 finalPos) {
    transform.localPosition = Vector3.Lerp(transform.localPosition,
                              finalPos, Time.deltaTime * transitionPerSecond);
    if (Mathf.Abs(transform.localPosition.z - finalPos.z) < 0.01f) {
        fakeDoor.SetActive(false);
        finished = true;
        if (isPlayerInside)
            player.transform.SetParent(playerContent.transform);
        isPlayerInside = false;
    }
    if (transform.localPosition == finalPos) {
        isActive = false;
} }
```

```
protected void OnTriggerEnter(Collider other) {
    informationShow = true;
    StartCoroutine(showInformation());
    changeMessageText("Presiona 'F' elegir el piso deseado");
}
```

```
    protected void OnTriggerExit(Collider other) {
        informationShow = false;
        StopCoroutine(showInformation());
        changeMessageText("");
    }

    IEnumerator showInformation() {
        while (informationShow) {
            if (Input.GetKeyDown("f") && GamePositionManager.isInGame()) {
                GameElevatorManager.setElevatorType(elevatorType);
                informationShow = false;
                changeMessageText("");
                panelAscensor.gameObject.SetActive(true);
            }
            yield return null;
        }
    }

    private void changeMessageText(string newMessage) {
        message.GetComponent<Text>().text = newMessage;
    }
}
```

En el cuadrado rojo, se muestran las variables públicas de la clase.

- **TransitionPerSecond:** Movimiento que realiza el ascensor por segundo.
- **PanelAscensor:** Panel UI con la información de los ascensores.
- **FakeDoor:** Puerta falsa, empleada para ocultar el hueco del ascensor desde dentro él.
- **Player:** Game object con el usuario.
- **Message:** Elemento del PanelGame donde se mostrará el texto de interacción.

El cuadrado azul, dispone las variables privadas de la clase.

- **IsActive:** Determina si el ascensor se encuentra en movimiento.
- **Finished:** Indica si el movimiento ha terminado.
- **FinalPos:** Indica las coordenadas destino del movimiento del ascensor.
- **InitialPos:** Indica las coordenadas iniciales del ascensor.
- **InformationShow:** Determina si se ha de mostrar el mensaje para interactuar con el ascensor.
- **Triggers:** Almacena los triggers de las distintas plantas, estos son empleados para llamar al ascensor.
- **IsPlayerInside:** Indica si el usuario se encuentra dentro del ascensor cuando este realiza su movimiento.
- **PlayerContent:** Game object padre del elemento *player.*
- **PlayerElevatorContent:** Elemento hijo del ascensor donde se añadirá al player para evitar problemas de colisiones.

En el cuadrado verde, se define el método *Start* de la clase. Este se encarga de iniciar las variables privadas de *isActive, triggers, playerContent* y *playerElevatorContent.*

El cuadrado amarillo, contiene las funciones públicas que permiten interactuar con el ascensor. El primer método, nos indica si el movimiento del ascensor ha terminado. El segundo método inicia el

movimiento del ascensor, definiendo la posición inicial y final e iniciando el movimiento mediante la corrutina. Se deben destacar dos aspectos.

- Para simplificar lógicas y procesos, los ascensores no contienen puertas, estas se encuentran en cada planta junto a los elementos del trigger. Es por ello que cuando se llama a un ascensor, se recorren todos los triggers asociado a ese ascensor y se cierra las puertas. Para que no se note dicho problema cuando nos encontremos dentro del ascensor, se añadió una puerta falsa que tapa desde el interior el hueco del ascensor.

- Debido a colisiones entre el game object del usuario y el ascensor cuando este se mueve, se adoptó la idea de obtener el game object del player y moverlo como hijo del ascensor cuando este inicia su desplazamiento, con ello conseguimos que ambos se desplacen juntos y sin colisiones.

En el cuadrado morado, se muestran los métodos auxiliares empleado por *callElevator.*

- **CloseDoors:** Recorre todos los triggers asociados y cierra las puertas.
- **TransitionElevator:** Corrutina que se encarga de mover el ascensor mientras este activo.
- **MoveElevator:** Método que realiza el movimiento del ascensor, si la diferencia entre la posición actual y la final es muy pequeña, oculta la puerta falsa, marca como que ha terminado y vuelve a restaurar su game object padre.

El cuadrado rosa, dispone los métodos que se ejecutan cuando se interactúa con el trigger. El objetivo de estas será mostrar el texto que permite interactuar con ascensor e iniciar la corrutina que permitirá acceder al *PanelElevator*. En caso de salir de la zona de acción del trigger, ocultará la información y detendrá la corrutina

En el cuadrado negro, tenemos el método de la corrutina. Este permite interactuar pulsando la tecla F del teclado, al realizarlo almacenamos el tipo de ascensor con el que accedimos. Posteriormente se muestra el *PanelElevator.*

El cuadrado naranja, define un método auxiliar que modifica el mensaje del *PanelGame* para mostrarnos como podemos interactuar con el ascensor.

ELEVATOR TRIGGER

Este script permitirá llamar a un ascensor en un piso concreto. Pero antes de revisar el script, veamos de qué elementos dispondremos.

- Elemento principal que contendrá el trigger y el script *ElevatorTrigger.*

- Puertas asociadas al trigger de esa planta, las puertas contendrán el script DoorTransitionOpen visto en la extensión DOORS.

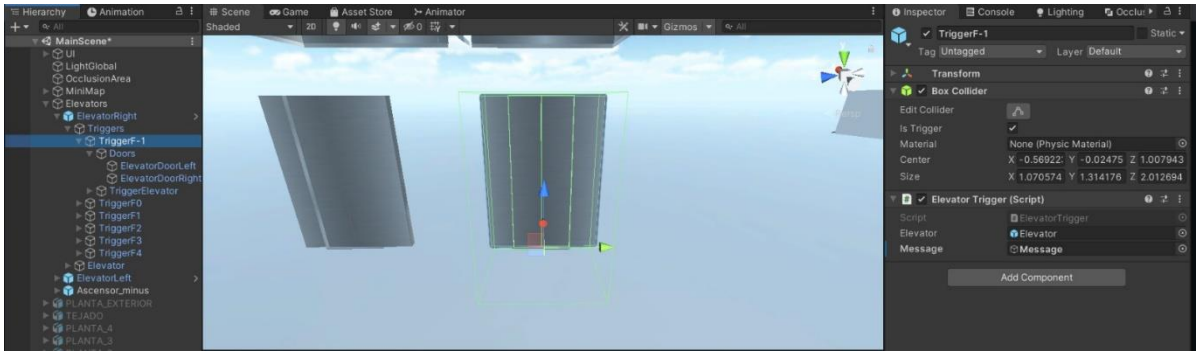


Fig. 105. Elementos existentes en el trigger de los ascensores

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;


public class ElevatorTrigger : MonoBehaviour {

    public GameObject elevator;
    public GameObject message;

    private float zCoord;
    private bool informationShow;
    private bool waitingElevator;
    private Elevator elevatorScript;
    private List<DoorTrasitionOpen> doors;

    void Start () {
        elevatorScript = elevator.GetComponent<Elevator>();
        zCoord = transform.localPosition.z;
        GameObject child = transform.Find("Doors").gameObject;
        doors = new List<DoorTrasitionOpen>();
        for (int i = 0; i < child.transform.childCount; i++) {
            doors.Add(child.transform.GetChild(i)
                           .gameObject.GetComponent<DoorTrasitionOpen>());
        }
    }

    public void closeDoors() {
        foreach (DoorTrasitionOpen door in doors) {
            door.close();
        }
    }

    public void callElevator(bool isPlayerInside) {
        elevatorScript.callElevator(zCoord, isPlayerInside);
        StartCoroutine(waitElevator());
    }
```

```
    protected void OnTriggerEnter(Collider other) {
        informationShow = true;
        StartCoroutine(showInformation());
        changeMessageText("Presiona 'F' para llamar al ascensor");
    }

    protected void OnTriggerExit(Collider other) {
        informationShow = false;
        StopCoroutine(showInformation());
        changeMessageText("");
    }
```

```
    IEnumerator showInformation() {
        while (informationShow) {
            if (Input.GetKeyDown("f") && GamePositionManager.isInGame()) {
                informationShow = false;
                changeMessageText("");
                callElevator(false);
            }
            yield return null;
        }
    }
```

```
    private void changeMessageText(string newMessage) {
        message.GetComponent<Text>().text = newMessage;
    }
```

```
    IEnumerator waitElevator() {
        waitingElevator = true;
        yield return null;
        while (waitingElevator) {
            if (elevatorScript.isFinished()) {
                waitingElevator = false;
                openDoors();
            }
            yield return null;
        }
    }
```

```
    private void openDoors() {
        foreach (DoorTrasitionOpen door in doors) {
            door.open();
        }
    }
}
```

En el cuadrado rojo, se muestran las variables públicas de la clase. La primera, hace referencia al ascensor asociado al trigger. La segunda, indica el elemento del *PanelGame* donde se mostrará el texto de interacción.

El cuadrado azul, dispone las variables privadas de la clase.

- **ZCoord:** Indica la posición hasta donde se desplazará el ascensor para llegar a la planta correspondiente al trigger.
- **InformationShow:** Determina si se ha de mostrar el mensaje para interactuar con la llamada al ascensor.
- **WaitingElevator:** Indica si se encuentra esperando a que llegue el ascensor.

- **ElevatorScript:** Contiene el script del elevador asociado al trigger.
- **Doors:** Dispondrá de las puertas asociadas al trigger.

En el cuadrado verde, se define el método *Start* de la clase. Este se encarga de iniciar las variables privadas de e*levatorScript, zCoord y doors.*

Los cuadrados amarillos, contiene las funciones que permiten abrir o cerrar las puertas asociadas al trigger. El método de cerrado será público, debido a que este se emplea desde el script *Elevator.*

En los cuadrados morados, disponemos de los métodos que se ejecutan cuando se llama al ascensor. El primero, llama al método *callElevator* visto en *Elevator* e inicia la corrutina *waitElevator*. El segundo, define la corrutina, cuyo objetivo es esperar hasta que el ascensor termine su movimiento, cuando eso sucede procede a abrir las puertas asociadas al trigger.

El cuadrado rosa, dispone los métodos que se ejecutan cuando se interactúa con el trigger. El objetivo de estas será mostrar el texto que permite interactuar con ascensor e iniciar la corrutina que permitirá llamar al ascensor. En caso de salir de la zona de acción del trigger, ocultará la información y detendrá la corrutina

En el cuadrado negro, tenemos el método de la corrutina. Este permite interactuar pulsando la tecla F del teclado, al realizarlo ejecutaremos el método *callElevator.*

El cuadrado naranja, define un método auxiliar que modifica el mensaje del PanelGame para mostrarnos como podemos interactuar con el ascensor.

ADICCIÓN Y CONFIGURACIÓN DE LOS GAME OBJECTS

Al igual que en la extensión ROOM, hemos de añadir estos scripts en los game objects, para así disponer de las funcionalidades programadas.

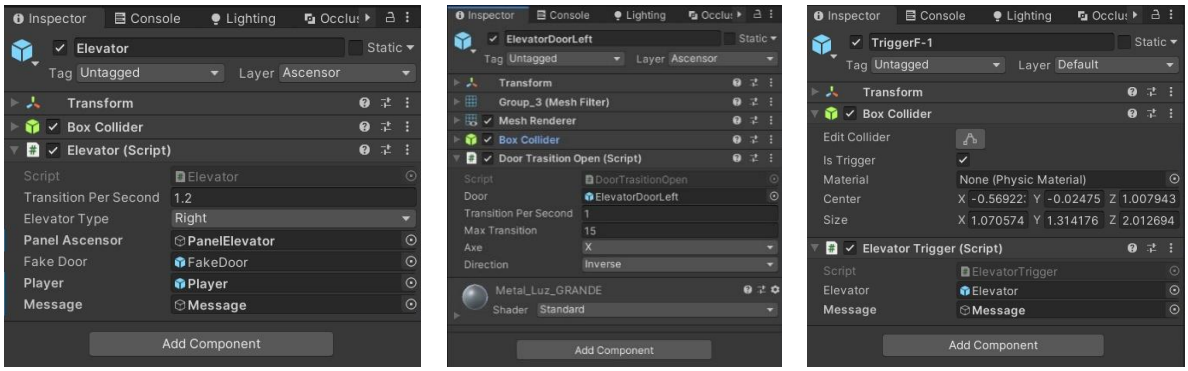


Fig. 106. Añadiendo scripts a los elementos game de los ascensores

4.5.4. CLASES UI DE LA EXTENSIÓN ELEVATOR

En este apartado vamos a ver las clases creadas para dar la funcionalidad al *PanelMap*.

ELEVATOR PANEL

Script asociado a *PanelElevator,* es el encargado de disponer de los métodos que permitirán salir del panel, mostrar la información del piso actual y seleccionar el piso actualmente mostrado para iniciar el movimiento del ascensor.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ElevatorPanel : MonoBehaviour {

    public GameObject buttonParking;
    public GameObject buttonZeroFloor;
    public GameObject buttonFirstFloor;
    public GameObject buttonSecondFloor;
    public GameObject buttonThirdFloor;
    public GameObject buttonFourthFloor;
    public GameObject triggerRight;
    public GameObject triggerLeft;

    private GameObject player;
    private Dictionary<EnumElevatorType, GameObject> triggers =
                                    new Dictionary<EnumElevatorType, GameObject>();
    private Dictionary<int, GameObject> buttons = new Dictionary<int, GameObject>();

    void Start() {
        triggers[EnumElevatorType.Right] = triggerRight;
        triggers[EnumElevatorType.Left] = triggerLeft;
        buttons[-1] = buttonParking;
        buttons[0] = buttonZeroFloor;
        buttons[1] = buttonFirstFloor;
        buttons[2] = buttonSecondFloor;
        buttons[3] = buttonThirdFloor;
        buttons[4] = buttonFourthFloor;
    }

    void OnEnable() {
        player = GameObject.Find("Player");
        showInformation();
    }

    public void selectElevatorFloor() {
        triggers[GameElevatorManager.getElevatorType()]
            .GetComponent<ElevatorTrigger>().callElevator(true);
        closeElevator();
    }

    public void closeElevator() {
        stopTime(false);
        player.GetComponent<PlayerController>().hideCursor();
        gameObject.SetActive(false);
    }

    private void stopTime(bool stop) {
        Time.timeScale = stop ? 0 : 1;
    }
```

```
    private void showInformation() {
        player.GetComponent<PlayerController>().showCursor();
        stopTime(true);
        int floor = GameMapManager.getFloor();
        if (buttons.ContainsKey(floor)) {
            buttons[floor].GetComponent<ElevatorButtonPanel>()
                          .showElevatorFloorInfotmation();
        } else {
            buttonZeroFloor.GetComponent<ElevatorButtonPanel>()
                           .showElevatorFloorInfotmation();
        }
    }
```

```
    public void setTriggerRight(GameObject newTrigger) {
        this.triggerRight = newTrigger;
        triggers[EnumElevatorType.Right] = newTrigger;
    }

    public void setTriggerLeft(GameObject newTrigger) {
        this.triggerLeft = newTrigger;
        triggers[EnumElevatorType.Left] = newTrigger;
    }
}
```

En el cuadrado rojo, se muestran las variables públicas de la clase. Las primeras, hacen referencia a los distintos botones, referentes a los pisos, que existen en el *PanelElevator*. Las últimas, hacen referencia al trigger que se llamará cuando se acepte el piso seleccionado. Este se iniciará con un valor inicial en la configuración del script, sin embargo, se modificará posteriormente a partir de los métodos que aparecen en el cuadrado negro.

El cuadrado azul, dispone las variables privadas de la clase. La primera, almacena al usuario de la aplicación. La segunda, contiene un mapa donde se referencia el *EnumElevatorType* al trigger asociado. La tercera, tendrá un mapa que permitirá referenciar el piso con el botón asociado.

En el cuadrado verde, se define el método *Start* de la clase. Este se encarga de inciar los mapas privados de la clase.

El cuadrado naranja, define los métodos auxiliares de la clase. El primero, permite modificar el tiempo del juego. El segundo, es el encargado de mostrar la información inicial del *PanelElevator,* en él se adecua la aplicación para interactuar con elementos UI y se muestra la información de la planta actual mediante el método *showElevatorFloorInfotmation* del botón correspondiente. En caso de no disponer del botón mostrará la información de la planta baja.

En el cuadrado rosa, se muestra la función que permite salir del *PanelElevator,* en ella se reanuda el tiempo del juego y se oculta el cursor y el panel.

El cuadrado amarillo, contiene el método que se empleará para aceptar el piso seleccionado. Este método ejecuta el método *callElevator* del trigger vinculado en el mapa *triggers*, seleccionándolo en función de la variable *elevatorType* existente en *GameElevatorManager*. Finalmente llama a *closeElevator* para continuar explorando el modelo interactivo.

ELEVATOR BUTTON PANEL

Script asociado a cada uno de los botones de selección de planta existentes en *PanelElevator,* es el encargado de mostrar la información del piso seleccionado en los paneles.

```
using System;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class ElevatorButtonPanel : MonoBehaviour {

    public Floor floor;
    public List<GameObject> buttons;
    public GameObject buttonAccepted;
    public GameObject triggerRight;
    public GameObject triggerLeft;
    public GameObject classroomText;
    public GameObject officeText;
    public GameObject serviceText;
    public GameObject seminaryText;
    public GameObject elevatorPanel;

    private string floorValue;
    private ElevatorPanel elevatorPanelScript;

    void Awake() {
        elevatorPanelScript = elevatorPanel.GetComponent<ElevatorPanel>();
        inicializeFloorValue();}

    private void inicializeFloorValue() {
        switch ((int) floor) {
            case 0:
                floorValue = "-1";
                break;
            case 1:
                floorValue = "0";
                break;
            case 2:
                floorValue = "1";
                break;
            case 3:
                floorValue = "2";
                break;
            case 4:
                floorValue = "3";
                break;
            case 5:
                floorValue = "4";
                break;
        }
    }

    public void showElevatorFloorInfotmation() {
        obtainValues();
        changeStyles();
        updatePanelElevatorValues();
    }
```

```
private void obtainValues() {
    classroomText.GetComponent<Text>().text = populateClassroom(
                GameElevatorManager.getClassroomsFromFloor(floorValue));
    officeText.GetComponent<Text>().text = populateDespachos(
                GameElevatorManager.getOfficeFromFloor(floorValue));
    serviceText.GetComponent<Text>().text = populateService(
                GameElevatorManager.getServicesFromFloor(floorValue));
    seminaryText.GetComponent<Text>().text = populateSeminary(
                GameElevatorManager.getSerminariesFromFloor(floorValue));
}


private string populateClassroom(List<ClassroomData> classroomList) {
    List<string> result = new List<string>();
    foreach (ClassroomData classroom in classroomList) {
        result.Add(populateLine(classroom.getCode(), classroom.getName()));
    }
    return joinListToString(result); }

private string populateDespachos(List<OfficeData> officeList) {
    List<string> result = new List<string>();
    foreach (OfficeData office in officeList) {
        result.Add(populateOfficeLine(office.getCode(), office.getLastname(),
                                      office.getFirstname()));
    }
    return joinListToString(result); }

private string populateService(List<ServiceData> serviceList) {
    List<string> result = new List<string>();
    foreach (ServiceData service in serviceList) {
        result.Add(populateLine(service.getCode(), service.getName()));
    }
    return joinListToString(result); }

private string populateSeminary(List<SeminaryData> seminaryList) {
    List<string> result = new List<string>();
    foreach (SeminaryData seminary in seminaryList) {
        result.Add(populateLine(seminary.getCode(), seminary.getName()));
    }
    return String.Join(Environment.NewLine, result.ToArray());
}

private string populateLine(string code, string name) {
    return String.Format("{0} - {1}", code, name); }

private string populateOfficeLine(string code, string lastname, string name) {
    return String.Format("{0} - {1}, {2}", code, lastname, name); }

private string joinListToString(List<string> stringList) {
    return String.Join(Environment.NewLine, stringList.ToArray()); }
```

```
private void changeStyles() {
    foreach (GameObject btn in buttons) {
        btn.GetComponent<Image>().color = Color.white;
    }
    this.GetComponent<Image>().color = new Color(1f, 0.47f, 0.47f);
    buttonAccepted.GetComponent<Button>().interactable = true;
}
```

```
    private void updatePanelElevatorValues() {
        elevatorPanelScript.setTriggerRight(triggerRight);
        elevatorPanelScript.setTriggerLeft(triggerLeft);
    }
}
```

En el cuadrado rojo, se muestran las variables públicas de la clase.

- **Floor:** Indica la planta a la que hace referencia el botón.
- **Buttons:** Lista de botones de selección de planta del *ElevatorPanel*.
- **ButtonAccepted:** Botón de aceptar la planta seleccionada.
- **TriggerRight:** Trigger del ascensor derecho correspondiente a la planta.
- **TriggerLeft:** Trigger del ascensor izquierdo correspondiente a la planta.
- Objetos que contendrán la información de las habitaciones de la planta.
- **ElevatorPanel:** Panel con la información del panel del ascensor.

El cuadrado azul, dispone las variables privadas de la clase. La primera, almacena la transformación del tipo *Floor* a integer para poder usar los métodos existentes en *GameElevatorManager*. La segunda, contiene el script asociado al *PanelElevator*.

En el cuadrado verde, se define el método *Awake* de la clase. Este se encarga de inciar las variables *elevatorPanelScript* y *floorValue*. Se emplea el método de utilidad *inicializeFloorValue*.

El cuadrado naranja, amarillo y rosa, define los métodos auxiliares de la clase. El cuadrado naranja, contiene los métodos que permiten obtener la información y visualización de esta en el panel UI. En el cuadrado amarillo, aparece el método que permite modificar los estilos de los botones, mostrando como seleccionado el botón y activando el botón de aceptar. El cuadrado rosa, muestra la función que permite actualizar los valores del trigger del script *ElevatorPanel.*

En el cuadrado morado, dispone del método que se registrará en el evento *onClick* del botón. Este se encarga de mostrar la información, cambiar los estilos de los botones y actualizar los valores de los triggers asociados en el *ElevatorPanel.*

ASOCIANDO LOS SCRIPTS AL UI

Una vez definidos los scripts, los asociamos a los distintos paneles y botones, al igual que vimos en el *PanelRoom*.

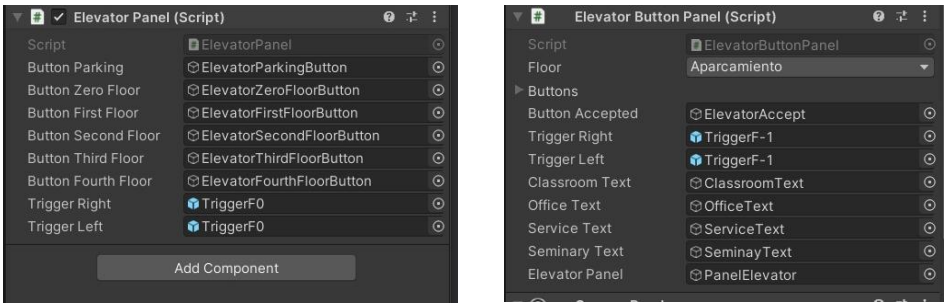


Fig. 107. Configurando las clases ui de la extensión ELEVATOR

4.6. EXTENSIÓN QUEST

En esta sección, vamos a modelar el comportamiento de las misiones de la aplicación. Así como dar funcionalidad al *PanelQuest*

.

4.6.1. CLASE DTO DE LA EXTENSIÓN QUEST

En esta extensión se ha creado una clase DTO que definirá las misiones exasistentes en la base de datos. Dispondrá de todos los datos necesarios para su empleo en la aplicación.

```
using UnityEngine;

public class QuestData : RoomData {

    private string code;
    private string name;
    private string description;
    private string goal;
    private string floor;
    private string room;
    private Vector3 coordinates;
    private Vector3 orientation;
    private Vector2 mapCoordinates;

    public QuestData(string codeValue, string nameValue, string descriptionValue,
                     string floorValue, string roomValue, string x, string y,
                     string z, string alpha, string beta, string gamma, string mapX,
                     string mapY) : base(codeValue, nameValue, RoomTable.Misiones) {
        code = codeValue;
        name = nameValue;
        description = descriptionValue;
        floor = floorValue;
        room = roomValue;
        coordinates = new Vector3(float.Parse(x), float.Parse(y), float.Parse(z));
        orientation = new Vector3(float.Parse(alpha), float.Parse(beta),
                                  float.Parse(gamma));
        mapCoordinates = new Vector3(float.Parse(mapX), float.Parse(mapY));
    }

    public QuestData(string codeValue, string nameValue, string descriptionValue,
                     string goalValue)
          : base(codeValue, nameValue, RoomTable.Misiones) {
        code = codeValue;
        name = nameValue;
        description = descriptionValue;
        goal = goalValue;
    }

    public QuestData(string codeValue, string nameValue)
        : base(codeValue, nameValue, RoomTable.Misiones) {}
```

```
    public string getDescription() { return description; }

    public string getGoal() { return goal; }

    public string getFloor() { return floor; }

    public string getRoom() { return room; }

    public Vector3 getCoordinates() { return coordinates; }

    public Vector3 getOrientation() { return orientation; }

    public Vector2 getMapCoordinates() { return mapCoordinates; }
}
```

En el cuadrado negro, tenemos la definición de la clase, se puede observar que hereda de *RoomData.* Disponiendo así de todos los métodos y variables definidas en el padre.

En el cuadrado rojo, encontramos las variables que dispone la clase, estas serán accesibles mediante los métodos existentes en el cuadrado azul.

El cuadrado verde, contiene los constructores de la clase. Disponemos de varios constructores, en función a la cantidad de información deseada.

4.6.2. GAME QUEST MANAGER

Encargada de disponer de variables y métodos necesarios para gestionar la información de la misión activa. Así como disponer de los distintos métodos que permitan obtener de la base de datos información de las misiones.

```
using System;
using System.Collections.Generic;
using UnityEngine;

public static class GameQuestManager {
    //CONSTANTS
    public static string QUEST_SCENE = "Quest";
    private static List<string> FIELDS_QUEST_FULL_INFO = new List<string>() {
    "NOMBRE", "DESCRIPCION_CORTA", "PLANTA", "ID_ROOM", "X", "Y", "Z", "ROTATE_X",
    "ROTATE_Y", "ROTATE_Z", "MAP_X", "MAP_Y" };
    private static List<string> FIELDS_QUEST_INFO = new List<string>() { "NOMBRE",
    "DESCRIPCION", "OBJETIVO" };
    private static List<string> FIELDS_QUEST_BASIC_INFO = new List<string>() {
    "ID_QUEST", "NOMBRE" };
    private static string TABLE_QUEST = "Misiones";
    private static string WHERE_QUEST_CONDITION = "ID_QUEST = '{0}'";
    //VARIABLES
    private static string questId = null;
    private static bool questIsActive = false;
    private static int questFloor = 999;
    private static string questRoom = null;
    private static Vector3 questEndCoordinates;
    private static bool questMessagePanelDisplayed = false;
    private static Vector2 questEndMapCoordinates;
    private static bool questNPCCreated = false;
```

```
public static bool isQuestInFloor(int floor) {return floor == getQuestFloor();}

public static bool hasInFloorOfQuest() {
    return isQuestInFloor(GameMapManager.getFloor());
}

public static bool isQuestSelected(string id) { return id == getQuestId(); }
```

```
public static QuestData getCurrentQuest(bool full) {
    return getQuestByID(getQuestId(), full);
}

public static QuestData getQuestByID(string questCode, bool full) {
    return full ? getQuestByIDFull(questCode) : getQuestByID(questCode);
}

private static QuestData getQuestByID(string questCode) {
    string query = GameDatabaseManager.obtainBasicQuery(FIELDS_QUEST_INFO,
            TABLE_QUEST, String.Format(WHERE_QUEST_CONDITION, questCode));
    List<string> dataList = GameDatabaseManager.obtainSingleElement(query);
    return dataList == null || dataList.Count == 0 ? null
         : new QuestData(questCode, dataList[0], dataList[1], dataList[2]);
}

private static QuestData getQuestByIDFull(string questCode) {
    string query = GameDatabaseManager.obtainBasicQuery(FIELDS_QUEST_FULL_INFO,
                TABLE_QUEST, String.Format(WHERE_QUEST_CONDITION, questCode));
    List<string> dataList = GameDatabaseManager.obtainSingleElement(query);
    return dataList == null || dataList.Count == 0 ? null
        : new QuestData(questCode, dataList[0], dataList[1], dataList[2],
          dataList[3], dataList[4], dataList[5], dataList[6], dataList[7],
          dataList[8], dataList[9], dataList[10], dataList[11]);
}

public static List<QuestData> getAllQuests() {
    List<QuestData> result = new List<QuestData>();
    string query = GameDatabaseManager.obtainBasicQuery(FIELDS_QUEST_BASIC_INFO,
                                                        TABLE_QUEST, null);
    List<List<string>> dataList =
                        GameDatabaseManager.obtainMultipleElements(query);
    if (dataList != null && dataList.Count != 0) {
        foreach (List<string> dataListRow in dataList) {
            if (dataListRow != null && dataListRow.Count != 0) {
                result.Add(new QuestData(dataListRow[0], dataListRow[1]));
            }
        }
    }
    return result;
}
```

```
public static string getQuestId() { return questId; }

public static void setQuestId(string id) { questId = id; }

public static bool isQuestActive() { return questIsActive; }

public static void setQuestActive(bool active) { questIsActive = active; }
```

```
    public static int getQuestFloor() { return questFloor; }

    public static void setQuestFloor(int floor) { questFloor = floor; }

    public static void setQuestFloor(string floor) { questFloor = int.Parse(floor); }

    public static string getQuestRoom() { return questRoom; }

    public static void setQuestRoom(string room) { questRoom = room; }

    public static Vector3 getQuestEndCoordinates() {
        return questEndCoordinates;
    }

    public static void setQuestEndCoordinates(Vector3 coordinates) {
        questEndCoordinates = coordinates;
    }

    public static bool isQuestMessagePanelDisplayed() {
        return questMessagePanelDisplayed;
    }

    public static void setQuestMessagePanelDisplayed(bool displayed) {
        questMessagePanelDisplayed = displayed;
    }

    public static Vector2 getQuestEndMapCoordinates() {
        return questEndMapCoordinates;
    }

    public static void setQuestEndMapCoordinates(Vector2 coordinates) {
        questEndMapCoordinates = coordinates;
    }

    public static bool isQuestNPCCreated() { return questNPCCreated;}

    public static void setQuestNPCCreated(bool created) {
        questNPCCreated = created;
    }
}
```

En el cuadrado rojo, aparecen las variables privadas necesarias para poder realizar las queries que se emplearán para la obtención de información de las misiones de la aplicación.

El cuadrado azul, muestra las variables que se emplearán para almacenar la información de la misión en curso. Todas estas variables serán accesibles y modificables mediante los métodos existentes en el cuadrado naranja.

En el cuadrado morado, encontramos métodos de utilidad de la clase, los cuales permiten saber si una misión se encuentra seleccionada o si se encuentra en un piso concreto.

En los cuadrados verdes, se definen los métodos que permitirán obtener la información de las misiones. Uno permite obtener la información de una misión a partir de su ID, indicando el nivel de información de la misión a obtener. El otro nos proporciona el listado de misiones existentes en la base de datos.

4.6.3. QUEST PLAYER CONTROLLER

En esta clase vamos a controlar la visualización de los elementos del PanelGame correspondientes a las misiones, el acceso al PanelQuest y la visualización del PanelMessageQuest.

```
using System.Collections;
using UnityEngine;
using UnityEngine.UI;


public class QuestPlayerController : MonoBehaviour {

    public float refresh_time = 0.5f;
    public GameObject questUI;
    public GameObject questEndPoint;
    public GameObject questIndicador;
    public GameObject questPanel;
    public GameObject questMessagePanel;

    void OnEnable() {
        checkQuestVisibility();
        StartCoroutine(questController());
    }

    void Update () {
        if (Input.GetKey("q") && GamePositionManager.isInGame()) {
            if (!GameQuestManager.isQuestActive()) {
                questPanel.gameObject.SetActive(true);
            }
    } }

    public void checkQuestVisibility() {
        checkQuestInfoVisibility();
        checkQuestMessagePanelVisibility();
    }

    private void checkQuestInfoVisibility() {
        if (GameQuestManager.isQuestActive()) {
            questUI.SetActive(true);
        }
    }

    private void checkQuestMessagePanelVisibility() {
        if (GameQuestManager.isQuestActive()) {
            if (GameQuestManager.hasInFloorOfQuest()) {
                if (!GameQuestManager.isQuestMessagePanelDisplayed()) {
                    initQuestMessagePanel("BUSCA LA HABITACIÓN " +
                            GameQuestManager.getQuestRoom(), true);
                }
            } else {
                initQuestMessagePanel("", false);
                GameQuestManager.setQuestMessagePanelDisplayed(false);
            }
        }
    }
```

```
    private void initQuestMessagePanel(string questMessage, bool active) {
        GameObject.Find("QuestMessage").GetComponent<Text>().text = questMessage;
        questIndicador.SetActive(active);
        questEndPoint.SetActive(active);
        questMessagePanel.SetActive(active);
    }

    IEnumerator questController() {
        while (true) {
            checkQuestMessagePanelVisibility();
            yield return new WaitForSeconds(refresh_time);
        }
    }
}
```

En el cuadrado rojo, se muestran las variables públicas de la clase.

- **Refresh_time:** Indica el tiempo que esperara entre las iteraciones de la corrutina.
- **QuestUI:** Panel UI con información de la misión actual en *PanelGame*.
- **QuestEndPoint:** Elemento UI del mini mapa con el final de la misión.
- **QuestIndicador:** Sprite del mini mapa que indica la dirección al punto final de la misión.
- **QuestPanel:** Panel que mostrará las misiones disponibles en la aplicación.
- **QuestMessagePanel:** Panel con la información de ayuda para terminar la misión.

El cuadrado azul, contiene la función *OnEnable* de la clase. Esta es la que permitirá al usuario acceder al panel con las misiones disponibles. Mediante la tecla Q, se mostrará el *PanelQuest*.

En el cuadrado verde, de define la función *Start* de la clase. Mediante esta función se comprueba si se ha de mostrar la información UI de misión y se inicia la corrutina. Para ello se apoya en los métodos de utilidad que aparecen en el cuadro naranja.

El cuadrado morado, dispone de la función de corrutina, esta permite comprobar si el usuario ha accedido al piso donde se encuentra el punto final de la misión y en ese caso mostrar los elementos UI de ayuda, que corresponden con el PanelMessageQuest, el punto final de la misión y la brújula que indica la dirección hacia el punto final de la misión actual.

4.6.4. CLASES GAME DE LA EXTENSIÓN QUEST

En la extensión QUEST sólo disponemos de una clase para elementos del modelo interactivo. Esta modela el comportamiento del elemento que permite finalizar la misión en curso. Antes de desarrollar el script debemos crear el PREFAB que contendrá la lógica para la finalización de la misión.

PREFAB DE FINALIZACIÓN DE LA MISIÓN

Para la finalización de la misión vamos a insertar un personaje, la idea es que para poder terminar la misión en proceso debamos acercarnos e interactuar con él.

Para el modelado 3D de este personaje, vamos emplear un modelo libre de derechos de autor que nos proporciona Adobe Mixamo. Desde su web ofrece personajes 3D y animaciones. De todo su catálogo utilizaremos a Pete, este nos recuerda a un posible topógrafo trabajando en campo.

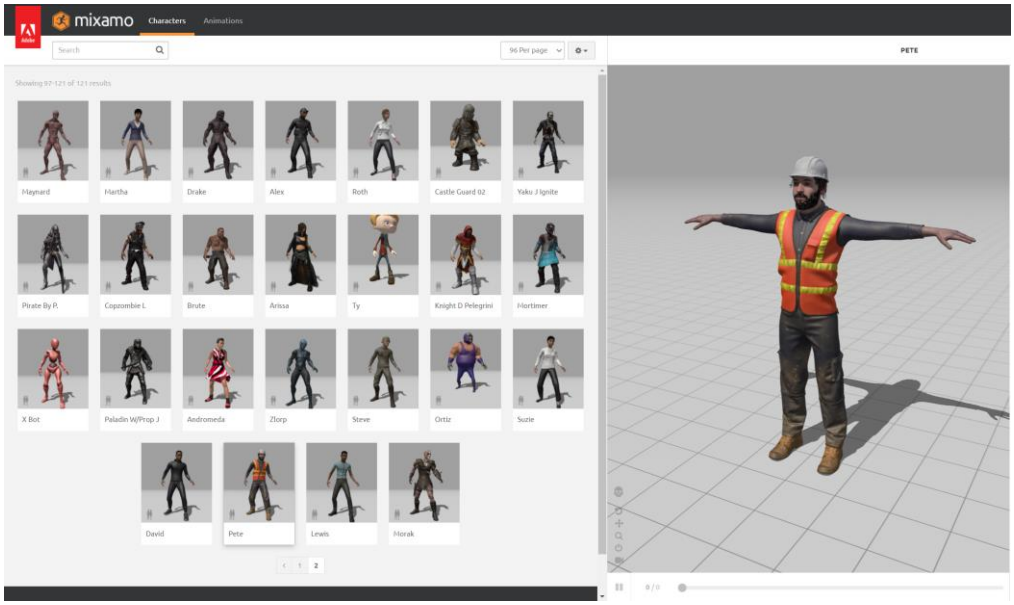


Fig. 108. Modelo 3D empleado de Adobe Mixamo

Mixamo, nos permite descargar el modelo en varios formatos, entre ellos uno ya existente para UNITY, sin embargo, sólo lo descargaremos en Collada, para disponer de las texturas de forma separada. El modelo en sí, se descargará cuando descarguemos las animaciones, ya que estas se optimizan en función del modelo 3D seleccionado.

Para dotarle de realismo al modelo 3D, vamos a descargar desde la misma página las animaciones del personaje. Descargaremos las animaciones de esperar, saludar y aplaudir. El comportamiento del personaje será saludarnos cuando nos acerquemos a la zona donde podamos interactuar con él y aplaudirnos cuando completemos la misión. Las animaciones permiten modificarse mediante algunos parámetros para adaptarlas a nuestras necesidades. Las descargaremos en formato UNITY, sólo para la animación de espera lo realizaremos con el modelo 3D.

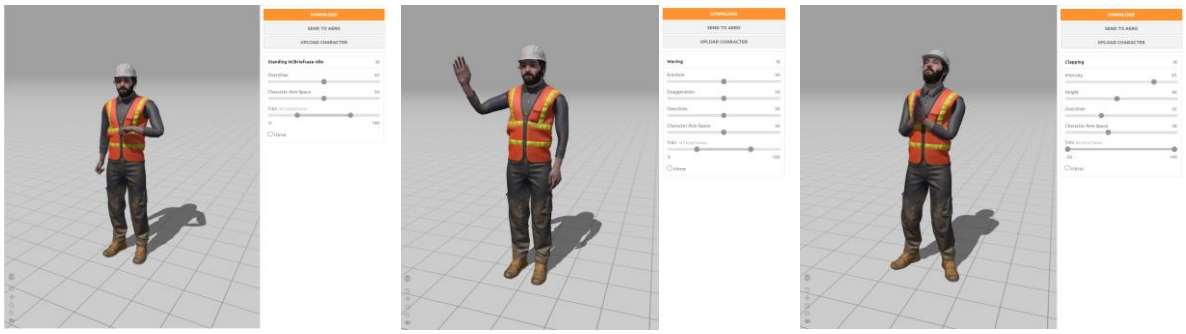


Fig. 109. Animaciones empleadas de Adobe Mixamo

Ahora es el momento de exportarlos a UNITY, pero primero realizaremos los materiales del personaje a partir de las texturas que nos proporcionan.

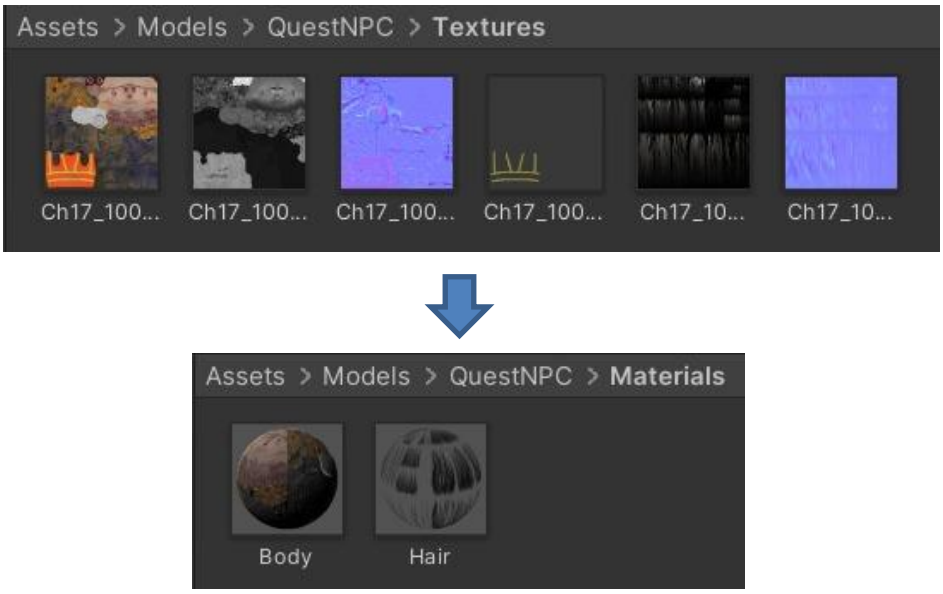


Fig. 110. Creando los materiales del personaje 3D

Una vez creadas las texturas vamos a exportar el modelo, para ello primero configuraremos la animación de espera que será animación base del personaje, para ello desde el la ventana de *Project* e *Inspector,* ajustaremos sus parámetros de importación. En la pestaña de *Rig*, indicaremos que se trata de un humanoide de tipo standard (4 huesos). En la pestaña de *Materials,* le indicaremos en *Locations -> Use Embedded Materials.* Esto nos permitirá asociarle unos materiales que habremos creado a partir de las texturas obtenidas del elemento Collada.

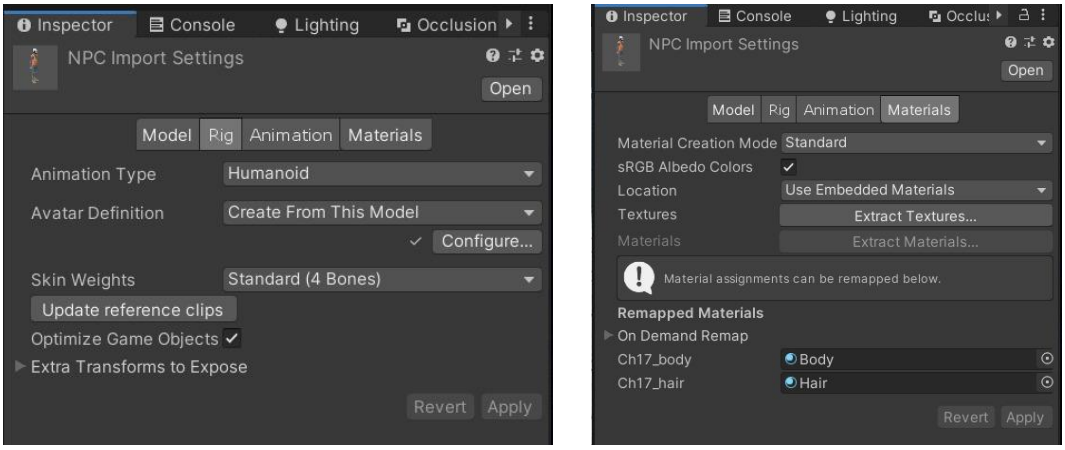


Fig. 111. Adaptando el personaje 3D

Para cada uno de los ficheros de animación descargados, extraeremos la animación y la guardaremos en una carpeta distinta. Modificaremos su información para indicar que se trata de una animación en bucle, para ello marcaremos las casillas de *Loop Time* y *Loop Pose.* Además, en esa carpeta crearemos un *Animator Controller.*

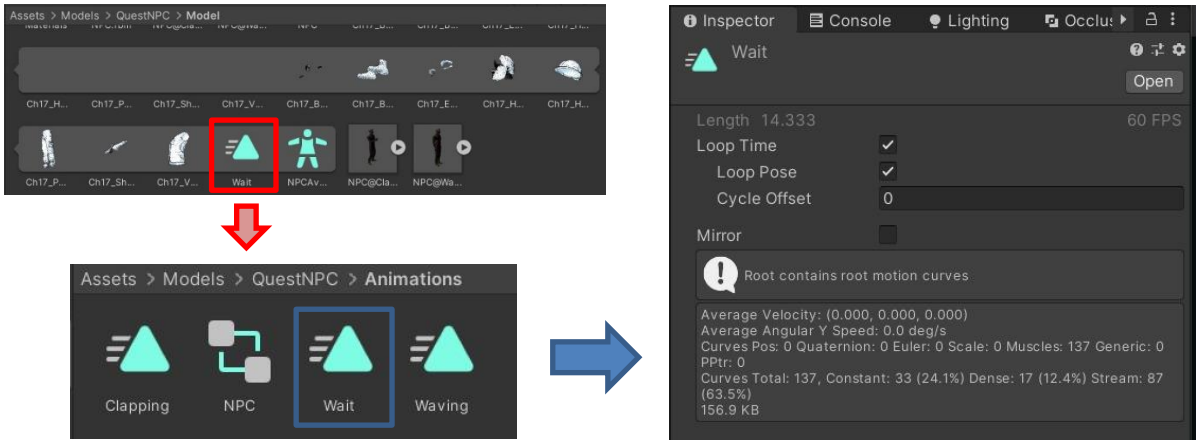


Fig. 112. Adaptando las animaciones 3D

Con todo configurado, ya podemos importar el modelo 3D, lo arrastramos a la escena. Veremos que se nos crea un game object con cada uno de los componentes del personaje. En el game object principal, se dispone de un componente de tipo *Animator,* a este le vamos a asignar el controlador creado. A este game object le vamos a asignar un componente de tipo *Capsule Collider,* el cuál adaptaremos a la geometría del modelo. Terminada su edición lo guardaremos como PREFAB.

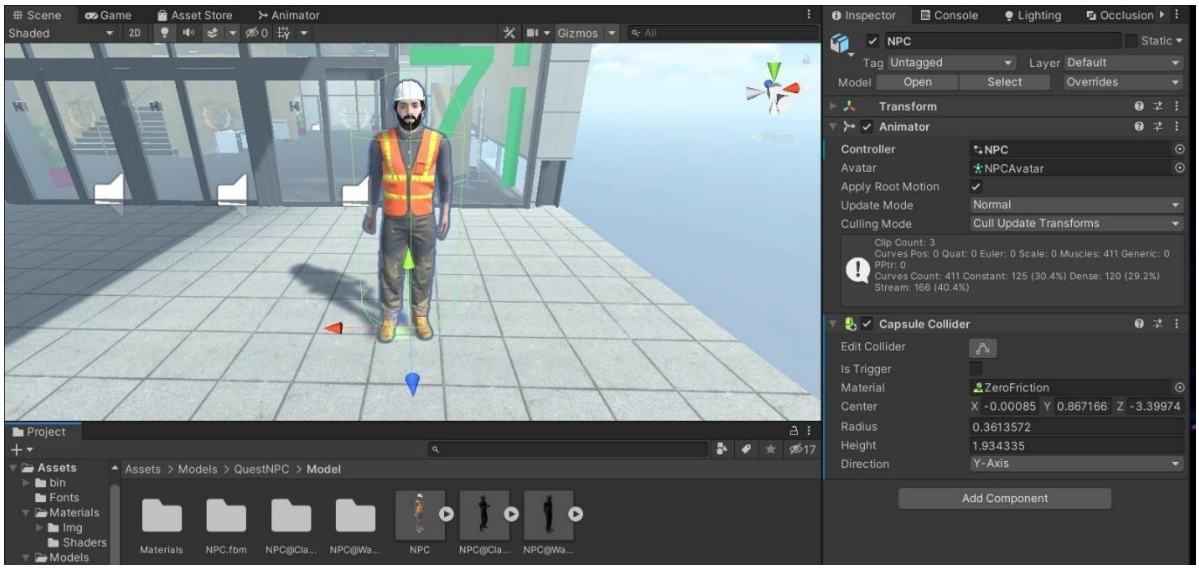


Fig. 113. Configurando el modelo en la escena

Vamos a configurar el controlador de animaciones, para ello lo seleccionamos desde la ventana *Proyect* y accedemos a la ventana *Animator.* A esta ventana arrastramos la animación de Wait, esta se relacionará directamente con la Entry. Seguidamente arrastramos las otras dos animaciones, estas aparecerán en gris y sin relacionar.

El siguiente paso será crear las transiciones desde Wait a las otras animaciones. Para realizarlo, haremos clic derecho sobre la animación y crearemos una transición uniéndolo con la otra, este procedimiento lo realizaremos en ambos sentidos.

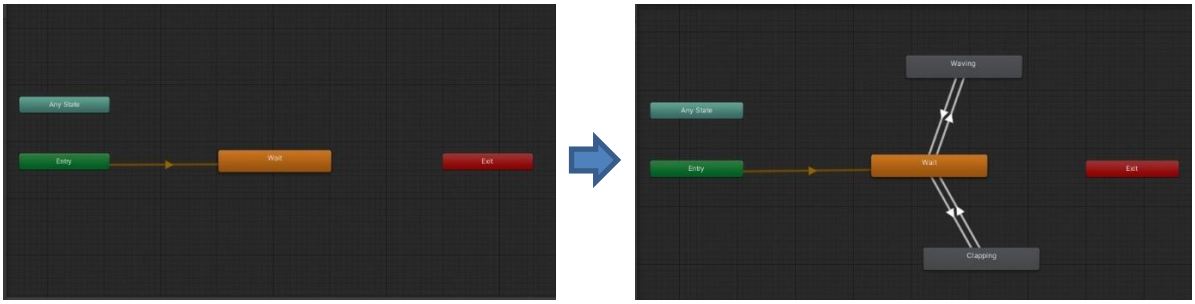


Fig. 114. Añadiendo animaciones y transiciones al Animator Controller

Para finalizar, deberemos indicar las condiciones que se han de dar para ir desde una transición a otra. Para ello, crearemos dos *parameters* de tipo *Bool*, uno lo llamaremos Hi y al otro Clap. El primero, permitirá cambiar entre las animaciones de saludo y espera. El segundo, controlará el cambio de la animación de aplauso a espera y viceversa. Para terminar, añadiremos las condiciones en las transiciones creadas anteriormente.

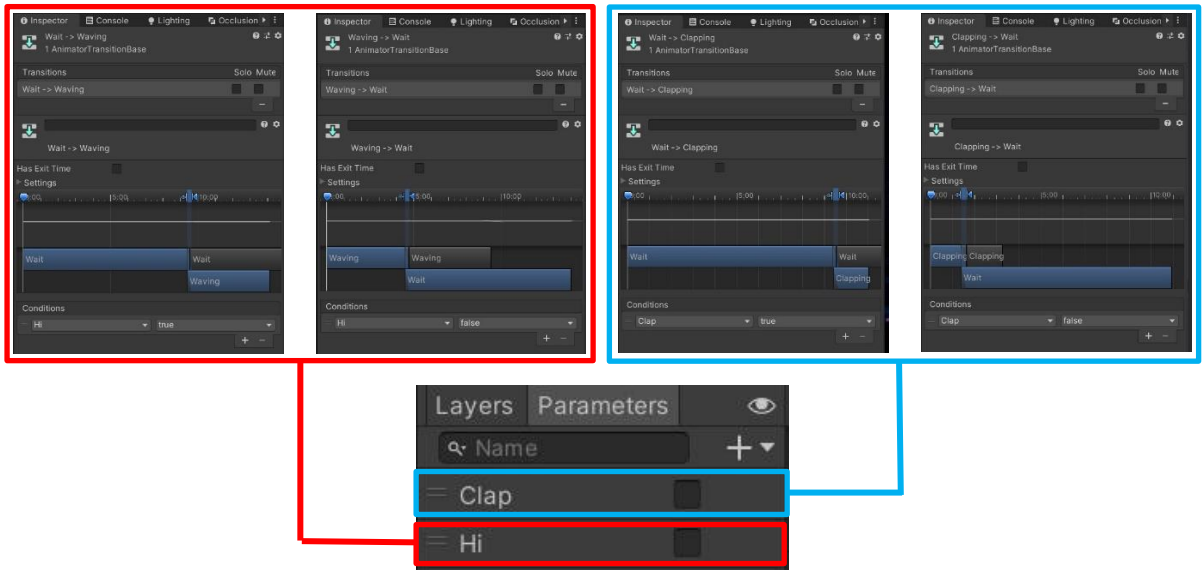


Fig. 115. Añadiendo animaciones y transiciones al Animator Controller

QUEST COMPLETED

Una vez dispongamos del PREFAB del personaje que permite terminar la misión, pasaremos a crear el script que le dotará de funcionalidad y modificará sus animaciones.

```
using System.Collections;
using UnityEngine;
using UnityEngine.UI;

public class QuestCompleted : MonoBehaviour {
```

```
private bool informationShow;
private GameObject message;
private bool isQuestFinished = false;
private AudioSource audioSource;
private Animator animator;
```

```
private void Start() {
    message = GameObject.Find("Message");
    audioSource = GetComponent<AudioSource>();
    animator = GetComponent<Animator>();
}
```

```
IEnumerator showInformation() {
    while (informationShow) {
        if (Input.GetKeyDown("f") && GamePositionManager.isInGame()) {
            animator.SetBool("Hi", false);
            endQuest();
        }
        yield return null;
    }
}
```

```
IEnumerator destroyObject() {
    while (true) {
        yield return new WaitForSeconds(4f);
        removeNPC();
        yield return null;
    }
}
```

```
private void endQuest() {
    animator.SetBool("Clap", true);
    audioSource.Play();
    showEndQuestMessage();
    resetQuestGlobalVariables();
    isQuestFinished = true;
    informationShow = false;
}
```

```
private void showEndQuestMessage() {
    message.GetComponent<Text>().text = "";
    GameObject.Find("QuestMessage").GetComponent<Text>().text =
                                                    "HAS COMPLETADO LA MISIÓN";
}

private void resetQuestGlobalVariables() {
    GameQuestManager.setQuestId(null);
    GameQuestManager.setQuestActive(false);
    GameQuestManager.setQuestFloor(999);
    GameQuestManager.setQuestMessagePanelDisplayed(false);
    GameQuestManager.setQuestNPCCreated(false);
}
```

```
    private void removeNPC() {
        GameObject.Find("QuestMessage").GetComponent<Text>().text = "";
        audioSource.Pause();
        animator.SetBool("Clap", false);
        Destroy(gameObject);
    }

    private void OnTriggerEnter(Collider other) {
        if (!isQuestFinished) {
            animator.SetBool("Hi", true);
            changeQuestMessageInfo(true, "Presiona 'F' para finalizar la misión");
            StartCoroutine(showInformation());
        }
    }

    private void OnTriggerExit(Collider other) {
        if (!isQuestFinished) {
            animator.SetBool("Hi", false);
            changeQuestMessageInfo(false, "");
        } else {
            StartCoroutine(destroyObject());
        }
    }

    private void changeQuestMessageInfo(bool visible, string messageValue) {
        informationShow = visible;
        message.GetComponent<Text>().text = messageValue;
    }
}
```

En el cuadrado rojo, se muestran las variables privadas de la clase.

- **InformationShow:** Indica si ha de mostrarse el mensaje de interactuar.
- **Message:** Elemento UI donde se mostrará el texto que permite interactuar.
- **IsQuestFinished:** Indicará si se la misión se ha finalizado.
- **AudioSource:** Componente de sonido que tendrá la pista de audio de los aplausos.
- **Animator:** Contendrá el componente *Animator* del personaje de finalización de quest.

El cuadrado azul, dispone del método *Start* de la clase. Este inicia las variables *message, audioSource* y *animator.*

Los cuadrados naranjas, definen los métodos de utilidad de la clase:

- **ShowEndQuestMessage:** Método muestra un mensaje en la parte superior del *PanelGame* indicando que se ha finalizado la misión.
- **ResetQuestGlobalVariables:** Encargado de resetear los valores de la misión en curso.

- **ChangeQuestMessageInfo:** Método que permite modificar el elemento UI del *PanelGame* que indica que puedes interactuar con un objeto.

En el cuadrado rosa, se muestra los métodos que permiten interactuar con los triggers asociados al game object. Estos inician o detienen la corrutina que permite interactuar con el personaje. El método que detecta la salida del trigger puede iniciar la corrutina que destruye el objeto si la misión ha sido finalizada. Se debe destacar que en los métodos de interacción con los triggers se transita entre las animaciones de esperar y saludar del personaje, esto se realiza modificando el parámetro *Hi.*

Los cuadrados verdes, dispone de la corrutina que permite interactuar con el personaje para terminar la misión y el método que finaliza la misión. La corrutina permite terminar la misión pulsando la letra F. Al hacerlo finaliza la animación de saludar, se inicia la de aplaudir, suena la pista de audio, muestra el mensaje de misión finalizada, resetea las variables de la misión en curso, actualiza *IsQuestFinished* a falso e indica que no debe mostrar el mensaje que permite interactuar con el objeto.

En los cuadrados morados, aparece el método de la corrutina y *removeNPC,* estos se encargan de borrar el personaje al terminar la misión. La corrutina espera cuatro segundos y llama al método *removeNPC*. Este método oculta el texto que indica que se ha terminado la misión, pausa el sonido iniciado en *endQuest*, detiene la animación de aplaudir del personaje y borra la instancia del prefab de la escena.

ADICCIÓN Y CONFIGURACIÓN DEL SCRIPT EN EL PREFAB

Tras realizar el script, debemos asignárselo al PREFAB del personaje que hemos realizado anteriormente. Para ello modificamos el PREFAB y se lo añadimos. Además, hemos de añadirle los componentes de *AudioSource* y *BoxCollider* para que el script funcione correctamente. El *collider* ha de estar marcado como trigger y ha de disponer de un buen tamaño para que podamos interactuar con él sin dificultad.

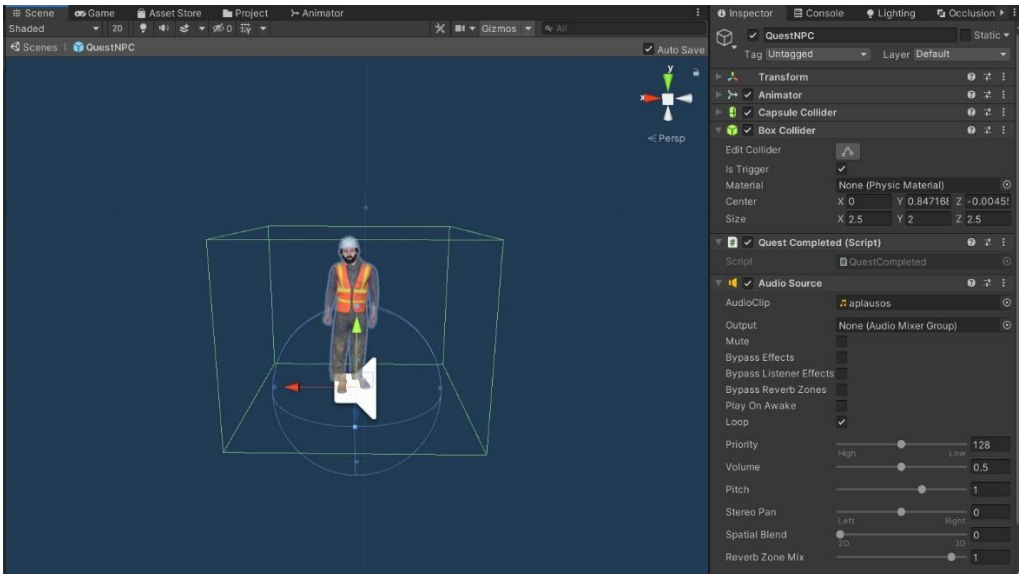


Fig. 116. Añadiendo script QuestCompleted al PREFAB

4.6.5. CLASES UI DE LA EXTENSIÓN QUEST

La extensión QUEST, añade varios scripts para elementos UI, vamos a distinguirlos según el panel UI al que vayan destinado. Disponemos de x paneles:

- **PanelQuest:** Dispondremos de tres scripts, uno para la inicialización del panel, otro para la carga de todas las misiones de la base de datos y uno para mostrar la información de la misión seleccionada.
- **PanelMessageQuest:** Script que permitirá leer la información del panel y reanudar la aplicación una vez leído por el usuario.
- **PanelGame:** Tendremos dos scripts, uno permitirá rellenar la información de la misión en curso y el otro modelar el comportamiento del indicador de ayuda de la quest.
- **PanelMap:** Script posicionará correctamente el marcador del punto final de la quest en el mapa.

SCRIPTS DEL PANEL QUEST

El primer script que veremos será el que se añadirá al *PanelQuest*, este se empleará para disponer de los métodos de los botones de aceptar y volver del panel. Además de la adecuación del sistema para interactuar con el UI o adecuarlo para continuar con la exploración del modelo interactivo.

```
using UnityEngine;

public class QuestSceneInit : MonoBehaviour {
    private GameObject player;

    void OnEnable() {
        player = GameObject.Find("Player");
        player.GetComponent<PlayerController>().showCursor();
        stopTime(true);
    }

    public void closeQuest() {
        GameQuestManager.setQuestActive(false);
        GameQuestManager.setQuestId(null);
        GameQuestManager.setQuestFloor(999);
        GameQuestManager.setQuestRoom(null);
        stopTime(false);
        player.GetComponent<PlayerController>().hideCursor();
        gameObject.SetActive(false);
    }

    public void acceptQuest() {
        GameQuestManager.setQuestActive(true);
        stopTime(false);
        player.GetComponent<PlayerController>().hideCursor();
        player.GetComponent<QuestPlayerController>().checkQuestVisibility();
        gameObject.SetActive(false);
    }

    private void stopTime(bool stop) { Time.timeScale = stop ? 0 : 1; }
}
```

En el cuadrado rojo, se muestra la variable privada de la clase. Esta variable hace referencia al game object *Player.*

El cuadrado verde, contiene la clase que se activa cuando se hace visible el *PanelQuest*, esta se encarga de instanciar la variable *player*, mostrar el cursor y detener el tiempo de la aplicación.

En el cuadrado azul, se define el método que permite cerrar el panel de las misiones sin aceptar ninguna. El método guarda el valor por defecto a todas las variables de misión actual, reanuda el tiempo de la aplicación y oculta el cursor y el *PanelQuest.*

El cuadrado morado, indica el método que empleará el botón de aceptar misión, el cual indica que se encuentra con la misión activa, reanuda el tiempo de la aplicación, llama al método checkQuestVisibility visto anteriormente y oculta el cursor y el *PanelQuest.*

En el cuadrado naranja, se dispone de un método de utilidad que permite modificar el tiempo de la aplicación.

El segundo script que veremos será el que mostrará la información de la quest seleccionada en los paneles correspondientes del *PanelQuest*.

```
using UnityEngine;
using UnityEngine.UI;

public class QuestButton : MonoBehaviour {

    private QuestData quest;

    public void setName(string newName) {
        name = newname;
        quest = GameQuestManager.getCurrentQuest(false);
    }

    public void selectQuest() {
        if (!GameQuestManager.isQuestSelected(name)) {
            GameQuestManager.setQuestId(name);
            populateQuestValues(quest);
        }
    }

    private void populateQuestValues() {
        GameObject.Find("QuestTitleLabel").GetComponent<Text>().text =
                            quest.getName();
        GameObject.Find("QuestDesciption").GetComponent<Text>().text =
                            quest.getDescription();
        GameObject.Find("QuestGoal").GetComponent<Text>().text = quest.getGoal();
    }
}
```

En el cuadrado rojo, disponemos de la variable privada de la clase, la cual contendrá la información de la misión asociada al botón.

El cuadrado azul, define el método que permitirá introducir en la variable *quest*, la información de la misión. Se apoyará en la clase *GameQuestManager* para obtenerla. Además, modifica el nombre del botón para que coincida con el id de la quest.

En el cuadrado verde, tenemos el método que empleará el botón de la misión. Este guardará la misión seleccionada y mostrará su información en los paneles, mediante el método de utilidad que aparece en el cuadrado naranja.

El último script que veremos será el que permitirá instanciar los botones de las quest existentes en la base de datos. Para ello, antes de realizar ningún script debemos generar el PREFAB que contendrá el botón de la quest.

Este PREFAB, contendrá un botón, con el script anteriormente visto y vinculando el método *selectQuest* en el evento *onClick* del botón. Además, contendrá un componente de tipo *Layout Element* para que se visualice correctamente en la zona del *scrollBar*.

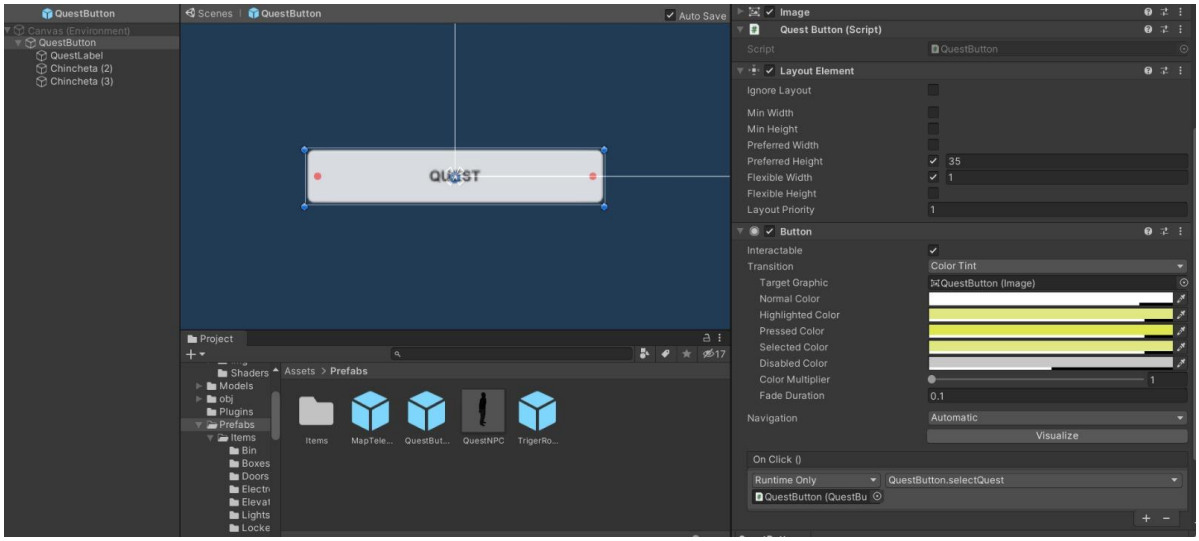


Fig. 117. Creando PREFAB de botón de selección de misión.

Una vez creado el PREFAB, pasamos a realizar el script que permitirá obtener todas las misiones y con ellas instanciar un PREFAB por cada una. Este se añadirá al panel que contendrá las misiones.

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class ObtainQuests : MonoBehaviour {

    public GameObject questButton;

    void Start () {
        List<QuestData> quests = GameQuestManager.getAllQuests();
        populateQuestValues(quests);
        GameObject.Find("Q001").GetComponent<QuestButton>().selectQuest();
    }
```

```
    private void populateQuestValues(List<QuestData> quests) {
        foreach (QuestData quest in quests) {
            GameObject bntQuest = Instantiate(questButton, transform);
            bntQuest.GetComponent<QuestButton>().setName(quest.getCode());
            bntQuest.transform.Find("QuestLabel").gameObject.GetComponent<Text>().text
                = quest.getName();
        }
    }
}
```

En el cuadrado rojo, disponemos de la variable pública de la clase, la cual hace referencia al PREFAB del botón de selección de misión.

El cuadrado naranja, define un método de utilidad de la clase. Este método es el encargado de que a partir de un listado de misiones, instanciar un PREFAB por cada una de ellas. Además, adecua los datos de cada una, indicándole la referencia de la misión y modificándole el texto a mostrar en el botón.

En el cuadrado verde, se muestra el método *Start* de la clase. En él se obtienen todas las misiones, se instancian los PREFABS y se selecciona la primera misión.

SCRIPTS DEL PANEL MESSAGE QUEST

Este script irá asociado al *PanelMessageQuest,* se encargará de adecuar la aplicación para la interacción con un panel UI, así como de restaurarlo para continuar explorando el modelo.

```
using UnityEngine;

public class QuestPanel : MonoBehaviour {

    private GameObject player;

    void OnEnable() {
        player = GameObject.Find("Player");
        player.GetComponent<PlayerController>().showCursor();
        stopTime(true);
    }

    public void BntAceptar() {
        GameQuestManager.setQuestMessagePanelDisplayed(true);
        stopTime(false);
        player.GetComponent<PlayerController>().hideCursor();
        gameObject.SetActive(false);
    }

    private void stopTime(bool stop) { Time.timeScale = stop ? 0 : 1; }
}
```

En el cuadrado rojo, se muestra la variable privada de la clase. Esta variable hace referencia al game object *Player.*

El cuadrado verde, contiene la clase que se activa cuando se hace visible el panel, esta se encarga de instanciar la variable *player*, mostrar el cursor y detener el tiempo de la aplicación.

En el cuadrado azul, se indica el método que empleará el botón de aceptar, el cual indica que ya se ha mostrado el mensaje de ayuda, reanuda el tiempo de la aplicación y oculta el cursor y el *PanelMessageQuest.*

El cuadrado naranja, dispone de un método de utilidad que permite modificar el tiempo de la aplicación.

SCRIPTS DEL PANEL GAME

El primer script que veremos será el que nos mostrará la información de la misión en curso en el *PanelGame*. Además, será el encargado de mostrar los elementos de la misión en el mini mapa e instanciar el personaje que nos permitirá terminar la misión.

```
using System;
using UnityEngine;
using UnityEngine.UI;

public class QuestInformationDisplay : MonoBehaviour {

    public GameObject questNPC;
    public GameObject questEndPoint;
    public GameObject questIndicator;

    private GameObject questName;
    private GameObject questDescription;
    private GameObject questSteps;

    private void OnEnable() {
        findQuestGameObjects();
        QuestData quest = GameQuestManager.getCurrentQuest(true);
        populateQuestValues(quest);
        updateGlobalQuestVariables(quest);
        showEndQuestObjects(quest);}

    private void Update() {
        if (!GameQuestManager.isQuestActive()){
            hideQuestObjects();
    } }

    private void findQuestGameObjects() {
        questName = GameObject.Find("QuestName");
        questDescription = GameObject.Find("QuestDescription");
        questSteps = GameObject.Find("QuestSteps");}

    private void populateQuestValues(QuestData quest) {
        questName.GetComponent<Text>().text = quest.getName();
        questDescription.GetComponent<Text>().text = quest.getDescription();
        string questStepsMessage = String.Format("- Dirigete a la Planta {0}",
                quest.getFloor());
        string questRoom = quest.getRoom();
        if (questRoom != "" && questRoom != null)
            questStepsMessage = String.Format("{0}{1}- Encuentra la habitación {2} ",
                                questStepsMessage, System.Environment.NewLine, questRoom);
        questSteps.GetComponent<Text>().text = questStepsMessage;
    }
```

```
    private void updateGlobalQuestVariables(QuestData quest) {
        GameQuestManager.setQuestFloor(quest.getFloor());
        GameQuestManager.setQuestRoom(quest.getRoom());
        GameQuestManager.setQuestEndCoordinates(quest.getCoordinates());
        GameQuestManager.setQuestEndMapCoordinates(quest.getMapCoordinates());}

    private void showEndQuestObjects(QuestData quest) {
      if (!GameQuestManager.isQuestNPCCreated()) {
            Vector3 questCoordinates = quest.getCoordinates();
            GameObject questPlayer = Instantiate(questNPC, questCoordinates,
                                    Quaternion.identity);
            QuestPlayer.transform.eulerAngles = quest.getOrientation();
            questEndPoint.transform.position = new Vector3(questCoordinates.x,
                    questEndPoint.transform.position.y, questCoordinates.z);
            GameQuestManager.setQuestNPCCreated(true);
       }
    }

    private void hideQuestObjects() {
        questIndicator.SetActive(false);
        questEndPoint.SetActive(false);
        gameObject.SetActive(false);}
}
```

En el cuadrado rojo, se observan las variables públicas de la clase:

- **QuestNPC:** PREFAB con el personaje que permite terminar la misión.
- **QuestEndPoint:** Elemento del mini mapa que indica el final de la misión.
- **QuestIndicator:** Elemento del mini mapa que marca la dirección hacia el punto final de la misión.

El cuadrado azul, contiene las variables privadas de la clase. Estas hacen referencia a elementos UI existentes en el PanelGame, en ellos se mostrará la información de la misión en curso.

En los cuadrados naranjas, se definen las funciones de utilidad de la clase.

- **FindQuestGameObjects:** Rellena las variables privadas con los elementos del UI.
- **PopulateQuestValues:** Muestra la información de la misión en el PanelGame.
- **UpdateGlobalQuestVariables:** Guarda la información de la misión actual.
- **ShowEndQuestObjects**: Instancia el personaje que permitirá terminar la misión actual y posiciona sobre el mini mapa el indicador del punto final de la misión.
- **HideQuestObjects**: Oculta todos los elementos UI de la misión.

El cuadrado verde, tiene el método *OnEnable.* Este método obtiene la misión en curso obtenida por GameQuestManager, asigna la información al PanelGame, guarda la información de la misión actual, posiciona el punto final de la misión en el mini mapa e instancia el personaje de final de misión.

En el cuadrado morado, se dispone del método *Update* de la clase, este se encarga de ocultar los elementos UI de misiones del *PanelGame* y ocultar los elementos de misión del mini mapa, en función al método isQuestActive que indica si el usuario dispone de una misión activa.

El último script que veremos, será el que modela el comportamiento del elemento del mini mapa que nos indica la dirección del final de la misión en curso.

```
using UnityEngine;

public class QuestPointer : MonoBehaviour {

    public GameObject endQuest;

    void Update() {
        transform.rotation =
            Quaternion.LookRotation(endQuest.transform.transform.position
                                                    - transform.position);
    }
}
```

En el cuadrado rojo, se muestra la variable pública de la clase, esta hace referencia al *Sprite* del mini mapa que indica la posición final de la misión.

El cuadrado verde, define el método *Update* de la clase, en él se calcula el ángulo entre la posición del final de la misión y el indicador y se corrige la rotación del indicador.

SCRIPTS DEL PANEL MAP

Este script irá asociado al indicador de final de misión que aparece en el *PanelMap,* se encargará de posicionarlo correctamente en el mapa. Como se trata de un script de adecuación de un elemento en el mapa, extenderá de la clase *MapShowIconFloor* vista en la extensión MAP.

```
using UnityEngine;

public class MapShowQuestIconFloor : MapShowIconFloor {

    protected override bool isVisibeMapIconFloor() {
        return GameQuestManager.isQuestInFloor(mapFloor);}

    protected override Vector2 obtainMapIconGamePosition() {
        return GameQuestManager.getQuestEndMapCoordinates();}

    protected override void updatePositionMapIcon(Vector2 mapIconGamePosition) {
        mapIcon.GetComponent<RectTransform>().anchoredPosition = mapIconGamePosition;
    }
}
```

En el cuadrado rojo, se define el método sobrescrito que hace referencia a si se ha de mostrar o no el indicador en la planta seleccionada en el *PanelMap.* Aquí nos apoyaremos en el método *isVisibeMapIconFloor* de la clase *GameQuestManager.*

El cuadrado azul, muestra el método que devuelve las coordenadas del elemento en el sistema 3D, debido a que el punto final para una misión siempre será el mismo. Se ha optado por simplificar la lógica y almacenar las coordenadas del mapa en la base de datos. Por tanto, la lógica se modifica para que *obtainMapIconGamePosition* devuelva las coordenadas ya transformadas en coordenadas UI y *updatePositionMapIcon* directamente las asigne en el sprite.

5. *EXPORTACIÓN DEL PROYECTO*

En este apartado vamos a crear la aplicación del proyecto UNITTY, para ello accedemos a Build Settings¸ allí indicamos que será una aplicación para PC, con arquitectura de x86_64 (64-bit CPU) y le añadimos la escena MainScene. Antes de continuar con configuración, diseñaremos un logo para la aplicación.

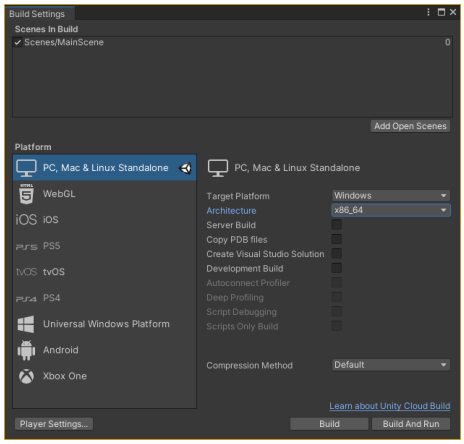
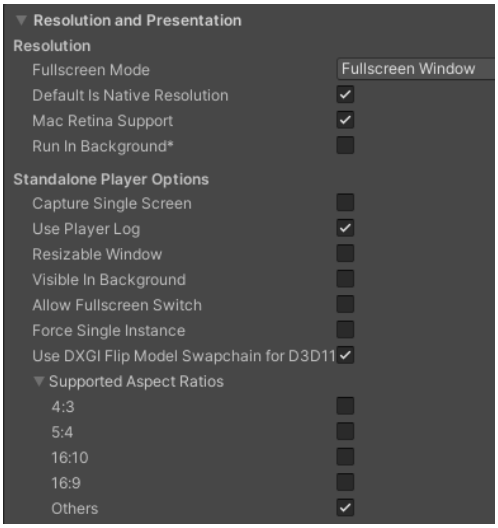


Fig. 118. Build Settings y diseño del logo de la aplicación

Desde la ventana Build Settings, accederemos a la opción de PlayerSettings, esta nos permitirá modificar varios parámetros de la aplicación generada. En esta ventana indicaremos el nombre del proyecto, el icono, la resolución y podremos configurar la pantalla de carga de la aplicación. En ella se pueden modificar el fondo y añadir logos, nosotros hemos optado por emplear un aula y añadir los iconos de la UPV y de la ETSIGCT



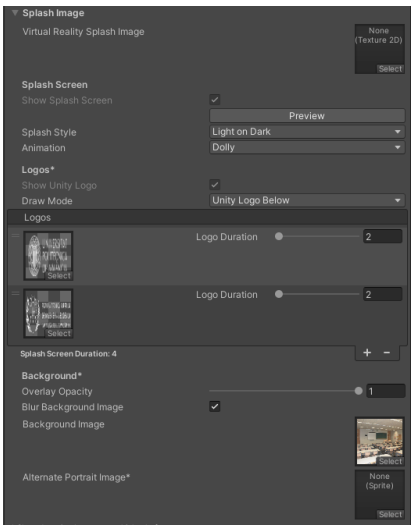


Fig. 119. Modificando parámetros de la aplicación

Para finalizar, vamos a definir la calidad visual de la aplicación, donde seleccionaremos la mejor calidad, posteriormente realizaremos otra exportación en fastest para ordenadores con menor potencia gráfica.

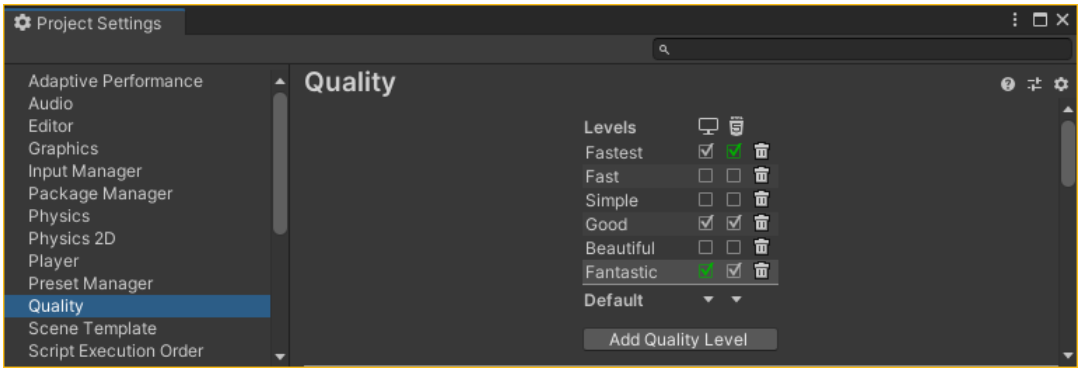


Fig. 120. Modificando calidad gráfica de la aplicación

Desde la ventana de Build Settings utilizaremos el botón Build y guardaremos la aplicación en una carpeta. Tras terminar el proceso, veremos en dicha carpeta nuestra aplicación, si la ejecutamos observamos que la información de la base de datos no se obtiene.

Fig. 121. Aplicación con error en la base de datos

Este problema es debido que UNITY no exporta la base de datos y por ello el script no la encuentra. Para arreglarlo, simplemente añadimos la carpeta Database a la carpeta Resources de la aplicación.

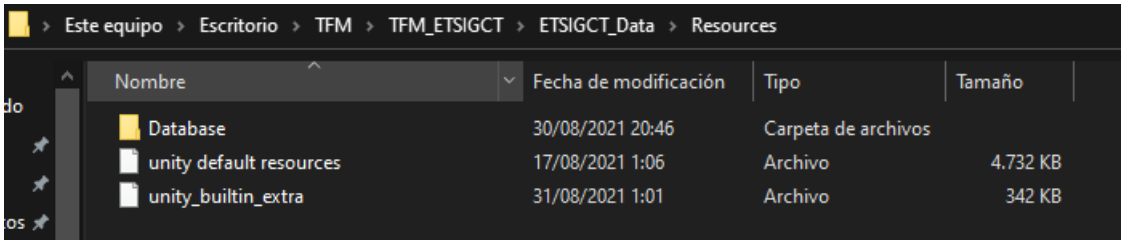


Fig. 122. Añadiendo la base de datos a la aplicación

Ahora si volvemos a ejecutar la aplicación, podremos observar que ya se dispone de acceso a la base de datos.

Fig. 123. Aplicación con la base de datos integrada

Con todo esto, ya dispondremos de nuestra aplicación, ejecutable desde cualquier ordenador a partir de los ficheros generados por el build, con un peso total de 642 MB, lo que la hace bastante portable y accesible desde un enlace en una web, permitiéndonos desde nuestra casa poder descargarla y emplearla cómodamente.

CONCLUSIONES Y OPINIÓN PERSONAL.

1. *CONCLUSIONES.*

Este proyecto es la prueba de que, con los conocimientos adquiridos en la titulación del máster, disponemos de gran cantidad de herramientas que nos permiten generar una mejora substancial del planteamiento de un proyecto desarrollado en la titulación de grado, donde hemos conseguido un modelo 3D bastante parecido al original y con funcionalidades interesantes para el público. El programa UNITY, nos ofrece infinitas posibilidades de desarrollo, donde nuestra imaginación y capacidades son el único hándicap existente.

Si bien es cierto que el proceso ha sido largo, no se ha tratado de un proceso complicado, por lo que podría ser una buena opción para la transformación digital de un edificio de carácter social, como podrían ser museos, tiendas… Si este tipo de modelos los combinamos con las nuevas tecnologías de realidad virtual, podríamos disponer de una experiencia aún más inversiva, esto es algo con lo que UNITY es compatible y sería una ampliación muy interesante.

Tras desarrollar el proyecto, podemos decir que el programa SketchUp permite realizar modelos 3D muy fácilmente, en cambio su integración con UNITY no es demasiado buena, haciéndonos invertir mucho tiempo de reprocesado y adecuación de los modelos. Es por ello, que no recomiendo para proyectos similares el empleo de este programa, en el mercado existen otras alternativas ampliamente difundidas, como BLENDER, que podrían darnos un mejor resultado.

2. *OPINIÓN PERSONAL.*

A título personal, este proyecto me ha parecido muy interesante, ya que en él se observa una cara menos conocida de aplicaciones que podemos realizar a partir de mapas, donde podemos dar rienda suelta a la gran cantidad de conocimientos de programación que nos ha proporcionado la titulación.

Este proyecto lo planteé para que se pudiera realizar con conocimientos aprendidos en la titulación, es por ello que se ha empleado el software de SketchUp. Sin embargo, este ha condicionado un poco el rendimiento de la aplicación. Me hubiese gustado emplear el lenguaje de Python, el cual hemos visto ampliamente en el master, pero Unity no permite su uso. No obstante, se ha podido aplicar lógica de programación, que es extrapolable a cualquier lenguaje.

Por otro lado, me gustaría dar mi opinión acerca de la información que contiene la aplicación, es cierto que no se encuentra actualizada, sin embargo, es sencillo el procedimiento de actualización, donde sólo debemos modificar las instancias de la base de datos y modificar los sprites de resources, con ello podríamos modificar los datos y disponer de la aplicación actualizada.

Es cierto que he tenido dificultades, debido a tratarse de un proyecto de planeación propia, donde uno parte de una idea y a medida que se va realizando el proyecto, este difiere de la idea original, ya sea por falta de conocimientos, por mejoras o por un concepto inicial no muy bien definido.

BIBLIOGRAFÍA.

1. *PUBLICACIONES Y CURSOS ONLINE.*

RUANO FOLCH, Daniel. *Diseño de un sistema de información geográfica de la ETSIGCT, visualización 2D/3D y análisis de caminos óptimos*.
Dirigido por Peregrina Eloina Coll Aliaga: Universitat Politècnica de València, 2015.
Trabajo fin de grado.

MOOC Introducción Desarrollo Videojuegos UNITY | Universitat Politècnica de València [en línea].
Disponible en: *https://www.youtube.com/playlist?list=PL6kQim6ljTJtOzDA4F_RDiJK_ljgk3_WQ*

2. *AYUDA DE PROGRAMAS.*

Documento de Ayuda SketchUp [en línea].
Disponible en: *https://help.sketchup.com/en/sketchup/sketchup*

Documento de Ayuda Unity [en línea].
Disponible en: *https://docs.unity3d.com/es/530/Manual/UnityManual.html*

3. *PÁGINAS WEB.*

Información de la ETSIGCT [en línea].
Disponible en: *http://www.upv.es/entidades/ETSIGCT/*

Módelos 3D de código libre [en línea].
Disponible en: *https://www.mixamo.com/#/*

Vídeos oficiales de ayuda de SketchUp [en línea].
Disponible en: *https://www.youtube.com/user/SketchUpVideo*