



# UNIVERSIDAD DE GRANADA

Facultad de Ciencias

Escuela Técnica Superior de Ingeniería Informática y Telecomunicación

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS

TRABAJO DE FIN DE GRADO

Realidad Virtual y Cuaterniones:  
aplicaciones a la planificación de  
representaciones teatrales.

Presentado por:  
Lucía Salamanca López

Tutor:  
Carlos Ureña Almagro  
*Departamento de Lenguajes y Sistemas Informáticos*  
Pedro A. García Sánchez  
*Departamento de Álgebra*

Curso académico 2022-2023



#### **DECLARACIÓN DE ORIGINALIDAD**

Dña. Lucía Salamanca López

Declaro explícitamente que el trabajo presentado como Trabajo de Fin de Grado (TFG), correspondiente al curso académico 2022-23, es original, entendida esta, en el sentido de que no ha utilizado para la elaboración del trabajo fuentes sin citarlas debidamente.

En Granada a 20 de noviembre de 2022

Fdo: Lucía Salamanca López



# Índice general

<b>1. Resumen</b>	<b>1</b>
<b>2. Abstract</b>	<b>3</b>
<b>I. Informática</b>	<b>5</b>
<b>3. Introducción y motivación</b>	<b>7</b>
3.1. Contexto . . . . .	7
3.1.1. Definición del teatro . . . . .	7
3.2. Elementos esenciales del teatro como arte escénico . . . . .	8
3.3. Descripción del problema . . . . .	9
3.4. Alcance de la memoria . . . . .	10
<b>4. Objetivos</b>	<b>13</b>
<b>5. Resolución del trabajo</b>	<b>15</b>
5.1. Planificación y presupuesto . . . . .	15
5.1.1. Planificación temporal . . . . .	15
5.1.2. Presupuesto . . . . .	16
5.2. Análisis y diseño . . . . .	17
5.2.1. Especificación de requisitos . . . . .	17
5.2.2. Historias de usuarios y Product Backlog . . . . .	18
5.2.3. Elementos importados . . . . .	19
5.2.4. Diseño Gráfico . . . . .	20
5.2.5. Descripción detallada del proyecto . . . . .	21
5.2.6. Bocetos interfaz de usuario . . . . .	25
5.2.7. Diagrama de Clases . . . . .	29
5.2.8. Principales algoritmos . . . . .	30
5.3. Implementación . . . . .	32
5.3.1. Ciclo de vida de un script de Unity . . . . .	32
5.3.2. Iluminación . . . . .	34
5.3.3. Creación del sistema de partículas . . . . .	39
5.3.4. Creación de los actores . . . . .	41
5.3.5. Movimiento del actor . . . . .	42
5.4. Manual de instalación . . . . .	47
<b>6. Conclusiones y Vías Futuras</b>	<b>49</b>

<b>II. Cuaterniones</b>	<b>51</b>
<b>7. Introducción</b>	<b>53</b>
<b>8. Historia de los cuaterniones</b>	<b>55</b>
<b>9. Cuaterniones</b>	<b>61</b>
<b>10. Teorema de Frobenius</b>	<b>65</b>
<b>11. Cuaterniones y rotaciones en 3D</b>	<b>71</b>
<b>Bibliografía</b>	<b>78</b>

# Índice de figuras

5.1.	Diagrama de Gantt del proyecto. . . . .	15
5.2.	Modelos escogidos . . . . .	20
5.3.	Telón de fondo añadido al escenario. . . . .	21
5.4.	Método de McCandless . . . . .	22
5.5.	Ejemplo real de luz focal . . . . .	23
5.6.	Boceto 1 . . . . .	26
5.7.	Boceto 2 . . . . .	26
5.8.	Boceto 3 . . . . .	27
5.9.	Boceto 4 . . . . .	27
5.10.	Boceto 5 . . . . .	28
5.11.	Boceto 6 . . . . .	28
5.12.	Diagrama de Clases . . . . .	30
5.13.	Dimensiones y coordenadas del escenario . . . . .	31
5.14.	Esquema para el cálculo de la esquina superior izquierda . . . . .	31
5.15.	Ciclo de vida de un <i>script</i> de <i>Unity</i> . Extraído de <a href="https://docs.unity3d.com/es/530/Manual/ExecutionOrder.html">https://docs.unity3d.com/es/530/Manual/ExecutionOrder.html</a> . . . . .	33
5.16.	Luces del método <i>McCandless</i> en <i>Unity</i> . . . . .	35
5.17.	Post-procesado. . . . .	36
5.18.	Configuración de la luz normal por el usuario. . . . .	37
5.19.	Luz normal sobre el escenario. . . . .	37
5.20.	Focos sobre los personajes. . . . .	38
5.21.	Capas para la luz . . . . .	39
5.22.	<i>Particle System</i> . . . . .	40
5.23.	Humo. . . . .	40
5.24.	Selección de la trayectoria para un actor. . . . .	41
5.25.	Importación de la animación . . . . .	43
5.26.	Importación de la animación . . . . .	43
5.27.	Animator Controller del actor . . . . .	44
5.28.	<i>NavMesh</i> del programa . . . . .	45
5.29.	<i>NavMesh</i> y Capsule Collider . . . . .	45
5.30.	Trayectoria de la Figura 5.24 . . . . .	48
11.1.	En la imagen de la izquierda se puede observar la representación de los cuaterniones $q$ y $r$ como vectores del espacio y el ángulo $\theta$ entre éstos siendo $Q$ el plano normal a $q$ . En la imagen de la derecha se muestra la representación de $r'$ . Imagen extraída de [1]. . . . .	73

11.2. Representación de los planos P y Q, la intersección de éstos $n$ y el ángulo $\phi$ entre ellos. Imagen extraída de [1]. . . . .	74
11.3. Representación de la rotación de los planos P y Q en $\tilde{P}$ y $\tilde{Q}$ , siendo la intersección de éstos una línea perpendicular al plano de la imagen. Imagen extraída de [1]. . . . .	75

# Índice de tablas

5.1. Presupuesto . . . . .	16
5.2. Product Backlog . . . . .	18
9.1. Tabla de Cayley . . . . .	63



# 1. Resumen

El presente trabajo de fin de grado tiene como objetivos el desarrollo de un software gráfico para la ayuda en el montaje de escenas de teatro y el estudio de los cuaterniones.

El proyecto informático parte de las dificultades con que se encuentran los directores de escena tanto en la concepción de la puesta en escena, como en la comunicación o transmisión de sus ideas a los distintos equipos de quienes dependen.

Este proyecto se presenta en la primera parte del trabajo, donde se desarrolla el análisis, diseño e implementación del programa. Este software permitirá simular aspectos técnicos de una representación teatral, así como el movimiento de los personajes. Para conseguir esto, se estudiará el motor de videojuegos *Unity*, donde se implementará el programa.

La segunda parte del trabajo corresponde con la parte más relacionada con las matemáticas. En ésta, se abordará la definición del álgebra sobre los reales conocida como los cuaterniones. Para ello, se incluirá un contexto histórico de éstos y una descripción del álgebra junto con las distintas operaciones. También, se presentará una demostración del Teorema de Frobenius donde se caracterizan las álgebras asociaitivas de división de dimensión finita sobre los números reales. Por último, se mostrará la relación de los cuaterniones con la informática gráfica mediante las rotaciones en el espacio.

**Palabras clave:** Unity, Teatro, Software Gráfico, Cuaterniones, Frobenius, Rotaciones, Hamilton.



## 2. Abstract

The aim of this dissertation is the development of a graphic software to help in the staging of theatre scenes and the study of quaternions.

The Software Development project is based on the difficulties encountered by stage directors both in the conception of the staging and in the communication or transmission of their ideas to the different teams on which they depend.

This project is presented in the first part of this document, where the analysis, design and implementation of the program are developed. In this software, it will be possible to simulate technical aspects of a theatrical performance, as well as the movement of the characters. To achieve this, the video game engine *Unity* will be studied, where the program will be implemented.

The second part of this document corresponds to the mathematical part. In this part, the definition of the algebra over the reals known as quaternions will be addressed. This will include a historical context of the quaternions and a description of the algebra together with the different operations. Also, a proof of the Frobenius Theorem characterising finite-dimensional associative division algebras over the real numbers will be presented. Finally, the relation of quaternions with graphical computing by means of rotations in space will be shown.

**Keywords:** Unity, Theatre, Graphic Software, Quaternions, Frobenius, Rotations, Hamilton.



# **Parte I.**

## **Informática**

Esta parte corresponde con la vertiente informática del trabajo. En ésta se profundizará en el desarrollo de un programa para la ayuda en el montaje de escenas de obras de teatro.



### **3. Introducción y motivación**

Este proyecto parte de la idea previa surgida durante la experiencia personal como parte integrante del elenco en diversas obras teatrales y de la colaboración estrecha con los directores de escena, sobre todo, durante los ensayos, al observar las dificultades con que se encuentran éstos tanto en la concepción de la puesta en escena, como en la comunicación o transmisión de sus ideas a los distintos equipos de quienes dependen. Este hecho generó el interés por realizar una valoración de la posibilidad de crear un programa informático, como conjunto de herramientas válidas para la configuración de las tareas preparatorias de un montaje escénico. Teatro y nuevas tecnologías ya estaban unidos desde hacía tiempo pero vinculadas estas últimas, fundamentalmente, a los resultados, no así al momento de la gestación o planificación.

#### **3.1. Contexto**

##### **3.1.1. Definición del teatro**

Conviene que hagamos las siguientes precisiones en cuanto a que el término teatro hace referencia tanto a género literario, como a espacio donde tiene lugar la representación. Pero lo que a nosotros nos interesa, y es en lo que nos vamos a centrar, es en el hecho teatral como arte escénica que combina las áreas de actuación, escenografía, música, sonido y espectáculo; no obstante, etimológicamente, la palabra teatro proviene del griego *θέατρον* (théatron), que a su vez deriva *θεᾶθαι* (theasthai), que significa “mirar”, lo que subraya y destaca su condición de arte audiovisual. Teatro es el nombre que se da tanto al arte y la técnica de la composición de obras de teatro, como a su interpretación o representación sobre las tablas de un escenario.

El ser humano siempre ha tenido la necesidad de contar historias. De hecho, prácticamente todo el arte se rige por una finalidad de intentar contar una historia o expresar una emoción. Siempre ha existido una necesidad intrínseca de poder verbalizar, ya sea mediante la palabra, la música o gráficamente, algún concepto o valor más abstracto que necesitamos compartir.

Si nos remontamos a sus orígenes, debemos recurrir a Aristóteles, quien en su *Poética*[2] describe qué es la tragedia:

La tragedia es la imitación de una acción seria y completa, de una extensión considerable, de un lenguaje sazonado, empleando cada tipo, por separado, en sus diferentes partes, y en la que tiene lugar la acción y no el relato, y que por

medio de la compasión y del miedo logra la catarsis de tales padecimientos.

[...] Una parte de la tragedia será el aderezo del espectáculo, y después la composición musical y la elocución, porque con estos medios llevan a cabo la imitación (Aristóteles, Poética, VI. Traducción de Alicia Villar [2004: 47-48])

Así pues, ya desde Aristóteles se describe la tragedia como una imitación que debe constar de ciertos elementos de expresión artística, entre los que vislumbramos la poesía, la música y la danza, todos estos elementos integrados por igual en la representación. Esta concepción del drama como un conjunto de diversas expresiones artísticas, llega hasta el concepto de *Gesamtkunstwerk* o concepto de obra de arte total teorizado más tarde por Wagner.

Asimismo, desde sus orígenes, debemos destacar el carácter político y social del teatro, puesto que está destinado a la sociedad. De este modo, el teatro es una de las disciplinas artísticas que puede identificarse con el tiempo en el que vive. El hecho teatral no se concibe sin el público, los ciudadanos, que son quienes ven, contemplan e interpretan todos los signos que el director/a ha empleado y moldeado.

### **3.2. Elementos esenciales del teatro como arte escénico**

El teatro ha ido evolucionando con las diferentes épocas y culturas. Hoy en día podemos contar con una gran variedad de subgéneros y podemos decir que el teatro contemporáneo abarca múltiples técnicas para su puesta en escena.

La práctica teatral está formada por un todo que no puede dividirse. Es posible, sin embargo, distinguir tres elementos básicos, como el texto (aquellos que dicen los actores), la dirección (las órdenes que dicta el responsable de la puesta en escena) y la actuación (el proceso que lleva a un actor a asumir la representación de un personaje). A estos componentes se pueden sumar otros elementos muy importantes, como el vestuario, el decorado, el maquillaje, la coreografía -entendida como los movimientos de los personajes por la escena-, la música o los efectos sonoros y los efectos especiales.

De entre todos, destacamos una parte esencial para que la pieza teatral se haga realidad, tome forma: la dirección. El director o directora lleva a cabo la coordinación de todos los elementos que conforman la representación, desde las actuaciones hasta la escenografía, el vestuario, la decoración, el maquillaje, la música, el sonido, la iluminación, etc. Esto es, las obras de teatro son de carácter interdisciplinario. En ellas se combinan elementos literarios, dramáticos, musicales y plásticos. Tienen como punto de inicio un elemento lingüístico: el texto dramático, sí, pero le toca a la labor de dirección tomar unas decisiones que harán que la obra tenga un resultado comunicativo u otro. Muchísimos montajes en la actualidad son de inspiración e interpretación muy libre por parte del director/a, aunque otros se atienden a las acotaciones marcadas por el autor teatral en sus obras.

Llegados a este punto, vemos la necesidad de que las nuevas tecnologías lleguen también a la parte previa del montaje de una obra como herramienta útil para estudiar todas

las posibilidades de una escena y, posteriormente, materializarlas tras la más adecuada elección según el criterio del director/a y todo su equipo (recordemos que la puesta en pie de un texto dramático obedece a un trabajo conjunto y muy interrelacionado de técnicos, especialistas y profesionales de diversos campos: escenógrafos, eléctricos, costureras, maquilladores, peluqueros... que el director coordina y finalmente, impone sus directrices).

Muchas veces existen ciertos problemas derivados de la falta de medios. Es frecuente que un director/a no tenga presente al equipo técnico durante la fase de preparación del proyecto o incluso, no disponga del lugar de la representación hasta pocos días previos al estreno. Esto dificulta y retrasa muchas veces el proceso de la puesta en escena.

Durante estos últimos años, se ha vuelto muy común la elaboración de prototipos mediante herramientas gráficas en proyectos de distinta naturaleza, lo cual no ha de sorprendernos ya que es la evolución que se podía esperar del boceto conforme al desarrollo de la tecnología.

Una de las herramientas más importantes en el mundo empresarial actual es *Unity*. *Unity* es un motor de videojuegos multiplataforma creado en 2015. El 50 % de los juegos están realizados con este motor (de media entre juegos para consolas, ordenadores y móviles). El éxito radica en que se centran en conseguir cubrir las necesidades de los desarrolladores independientes (que generalmente no pueden permitirse el pago de muchas herramientas ni la creación de su propio motor), buscando que el desarrollo de este tipo de contenido sea accesible para todo el mundo. A pesar de poder pensar que está especializado sólo en videojuegos, no es así, *Unity* ofrece numerosas herramientas para poder crear contenido interactivo 2D y 3D, siendo frecuente su uso para Realidad Virtual, películas de animación o incluso, maquetas de distintos productos (arquitectónicas, automovilísticas, etc), que, teniendo en cuenta que muchos de los videojuegos incluyen características similares, no es de extrañar la versatilidad de este motor.

Por lo que, la principal idea de este trabajo, surge de poder crear un programa que ayude al director de escena a imaginar el resultado final de ésta, incluyendo la parte técnica de la que carece en ese momento. Esto puede ayudar a potenciar la creatividad en el montaje o simplemente ayudar a descartar ideas debido a que el ambiente creado en la escena no es el esperado. Para poder crear este programa se decide usar *Unity* debido a las numerosas cualidades que posee, que se explicarán más adelante.

### 3.3. Descripción del problema

El objetivo del trabajo será el desarrollo de un programa mediante el cual el usuario podrá previsualizar un prototipo de una escena de teatro. Es decir, el usuario podrá elegir entre una serie de variables para finalmente, reproducir una escena en la cual esas variables estén involucradas. Concretamente, el usuario podrá elegir entre un tipo de iluminación para el escenario y la presencia de humo en dicho espacio. También podrá elegir el número de actores que se encuentren en escena, así como el movimiento de

éstos. Cuando haya hecho todas esas elecciones, obtendrá una muestra de los elementos seleccionados en la escena, pudiendo reproducir el movimiento de los personajes tantas veces como quiera, con la posibilidad de cambiar los otros aspectos para obtener el resultado que mejor se adegue a su gusto.

Para ello será necesario la relación de los conceptos aprendidos en asignaturas enfocadas a la creación de software gráfico como Informática Gráfica y Sistemas Gráficos con la aplicación práctica en *Unity*.

Una de las principales ventajas de *Unity* es que puede usarse junto con otros programas tales como *Blender*, *3ds Max*, *Maya*, *Softimage*, *Modo*, *ZBrush*, *Cinema 4D*, *Cheetah3D*, *Adobe Photoshop*, *Adobe Fireworks* y *Allegorithmic Substance*. Esto es, pueden importarse objetos de otros programas e integrarlos en *Unity*, por lo que no es necesario el modelado de los elementos ya que podemos importarlos de otros ya existentes, siempre que lo permitan los derechos de uso.

Otra de las ventajas es la amplia cantidad de documentación [3] que ofrece *Unity*. Además, posee un foro [4] para poder comentar los distintos errores o dudas que puedan surgir. Más allá de esto, *Unity* ofrece un proceso de aprendizaje [5] del motor mediante tutoriales divididos en distintos caminos. Estos caminos son cuatro: uno, en el que se aprende lo básico del motor; otro, enfocado a la programación; otro, enfocado hacia una parte más creativa; y un último camino, enfocado a la Realidad Virtual.

Para la programación, *Unity* usa el lenguaje C#, lenguaje de programación multi-paradigma desarrollado y estandarizado por la empresa Microsoft como parte de su plataforma .NET. Su sintaxis básica deriva de C/C++ y utiliza el modelo de objetos de la plataforma .NET, similar al de Java, aunque incluye mejoras derivadas de otros lenguajes. Este lenguaje va a ser usado mayoritariamente en los *scripts* asociados a los distintos objetos creados en el programa para modificar algún comportamiento en específico.

El camino de los tutoriales de creatividad, comentado previamente, ha sido de relevante importancia ya que, gracias a él, se ha podido dar forma a la idea del programa, permitiendo un mayor conocimiento de las herramientas que ofrece *Unity* y consiguiendo así el enfoque necesario para abordar la creación de éste. Este camino se subdivide en distintos apartados. Los más útiles han sido el de materiales y *shaders*, iluminación, animación, efectos especiales, cámaras, audio, interfaz de usuario y post-producción.

### 3.4. Alcance de la memoria

La memoria sigue una estructura que abarca distintos aspectos del desarrollo del proyecto. Además, existe una parte II, que no tiene relación con este proyecto, en el que se presenta el álgebra de división sobre los reales denominada cuaterniones  $\mathbb{H}$ .

En el siguiente capítulo 4 se comentan los objetivos iniciales del proyecto y los finalmente alcanzados.

Posteriormente, en el capítulo 5 se detalla el desarrollo del programa. Para ello se

hablará del proceso previo a éste en la sección 5.1, donde se especifica la planificación inicial y el presupuesto, y en la sección 5.2. En esta sección se muestran aspectos del análisis y el diseño del programa, incluyendo la especificación de requisitos, bocetos de la interfaz de usuario, el diagrama de clases y la explicación de los algoritmos. Además, se incluye una descripción detallada del problema a abordar. Finalmente, en la sección 5.3 se abarcará la implementación del programa, incluyendo extractos de código.

En el capítulo 6 se destacan posibles vías futuras para el programa así como conclusiones sobre éste.

Para finalizar se encuentra la bibliografía en el capítulo 11, donde se encuentran las fuentes consultadas tanto para este proyecto como para la parte matemática.



## 4. Objetivos

El objetivo general del trabajo era el análisis, diseño e implementación de un software gráfico y multimedia 3D que ayudase a la planificación de escenas de teatro, que permitiese el diseño de las mismas y la visualización y animación en 3D de un escenario virtual, incluyendo los elementos fijos, móviles y las trayectorias, así como los diálogos de los personajes.

Sin embargo, muy al comienzo del proyecto uno de los elementos a tener en consideración cambió: se vio que el diálogo de los personajes no tenía una importancia relevante para la planificación de las escenas, puesto que su interpretación era algo que no se podía planear con antelación pues obedecía y estaba íntimamente ligado a la intervención y presencia real de los actores. Por lo que se suprimió este elemento y se dio más importancia a la parte técnica, esto es, a la parte más “objetiva”.

Así que el objetivo real del trabajo se fijó en el desarrollo de un software gráfico para la ayuda en el montaje de escenas de teatro, que permitiese simular los aspectos técnicos de una representación teatral, así como el movimiento de los personajes, intentando recrear la escena, buscando la mayor similitud con la realidad, sin que esto afectara mucho a la eficiencia del programa.

Para conseguir este objetivo principal, era necesario un conocimiento previo de la principal herramienta a usar, que en este caso es el motor de videojuegos *Unity*, y de las posibilidades que éste ofrecía.

También se debía crear un espacio que emulara un teatro y la adaptación de este escenario a distintos cambios. Para ello era necesario encontrar un modelo que simulase dicha estructura, con las modificaciones necesarias. Los prototipos que fueran seleccionados tendrían que ser lo más genéricos posibles para que pudiesen ser usados en cualquier situación, permitiendo una generalización en el programa que revirtiera en una mayor utilidad.

Como hemos comentado previamente, se buscaba intencionadamente una similitud con la realidad, por lo que, se debían explorar técnicas concretas y usuales en el montaje de escenas de obras de teatro para implementarlas en el programa.

Otro de los objetivos específicos lo constituye la alteración de la iluminación en la escena, buscando que el usuario pudiese personalizar el diseño de iluminación a su gusto.

En cuanto al movimiento de los personajes se hacía necesario elaborar éste siguiendo una trayectoria de puntos específica, buscando naturalidad en las transiciones mediante animaciones y evitando los posibles errores tales como las colisiones entre éstos.

Asimismo, se buscaba personalizar la escena aportando una ambientación específica para la misma. Para esto, se ha creado la opción del humo, dejando de lado otras posibilidades de creación de ambientes como vías futuras, puesto que debíamos acotar las posibilidades reales de nuestro trabajo.

Por último, hemos de destacar que una de las finalidades más importantes a conseguir era la visualización de la escena, combinando todos los elementos comentados previamente. Para conseguirlo era necesaria una interfaz de usuario fácil e intuitiva que permitiese que el usuario pudiese cumplir dichos objetivos.

En el siguiente capítulo se abordará el desarrollo de los objetivos.

# 5. Resolución del trabajo

En este capítulo se mostrarán los métodos y procesos empleados para el desarrollo del trabajo.

## 5.1. Planificación y presupuesto

Se estima que el proyecto de Informática va a requerir unas 300 horas de trabajo.

### 5.1.1. Planificación temporal

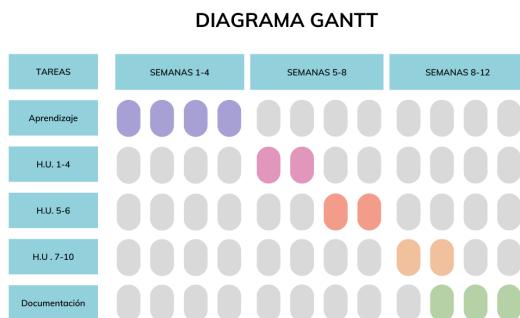


Figura 5.1.: Diagrama de Gantt del proyecto.

Para la planificación se va a tener en cuenta tres partes del proceso que se realizarán en cascada. Primero, se necesita una primera etapa donde se aprenda Unity y la herramienta. Como se dijo en la introducción, se seguirá un proceso de aprendizaje por tutoriales. Dentro de esta primera etapa, también se contemplará el proceso de análisis y diseño del programa a realizar. Como se comenta más adelante, se seguirá una metodología *Scrum*, por lo que el proceso de desarrollo del *Product Backlog* (Tabla 5.2) corresponde a esta etapa. Otras partes del análisis y diseño pueden ser realizadas en la implementación debido a la metodología escogida.

La implementación del programa se desarrolla en tres iteraciones de dos semanas aproximadamente, siendo la última iteración más corta que las dos primeras, como se puede

ver en la Sección 5.2.2. Dentro de cada historia de usuario se elaborarán los distintos bocetos de la interfaz necesarios así como las pruebas necesarias para la implementación.

Por último, se necesita una etapa de documentación que corresponde con la realización de este documento.

El total de semanas para este trabajo es de 12 semanas aproximadamente, comenzando la primera semana de septiembre. Por lo que el trabajo por semana será de unas 25 horas semanales. Seis semanas corresponderán con la implementación del programa, cuatro con el aprendizaje de la herramienta y dos semanas para la documentación, pudiéndose comenzarse ésta antes, ya que la última iteración de la implementación será más corta.

### 5.1.2. Presupuesto

Se estima que el precio del proyecto sea de aproximadamente 4779 €.

	Coste	Total
Ordenador Lenovo	900 €	900 €
Auriculares JBL	30 €	30 €
Espacio co-working	180 €/mes	540 €
Programadora Junior	11'03 €/hora	3309 €
<b>Total</b>		<b>4779 €</b>

Tabla 5.1.: Presupuesto

Para realizar el proyecto necesitamos un ordenador con la capacidad suficiente para poder ejecutar *Unity*, en este caso se ha usado un portátil Lenovo ideapad 700-15ISK, cuyo precio ronda aproximadamente los 900 €. Para poder comprobar el correcto funcionamiento del sonido en el programa y para poder alterar éste de manera correcta, se han necesitado unos cascos analógicos JBL de gama media cuyo precio es de 30€ aproximadamente. No ha sido necesario comprar una licencia de *Unity* ya que, en este caso, se ha usado la licencia de estudiante que es gratuita y ofrece ciertas ventajas con respecto a la licencia gratuita convencional. Toda la bibliografía usada para el proyecto ha sido gratuita y no ha hecho falta adquirir ninguna con coste.

En España, el sueldo base promedio de un programador junior es de 11'03 €/hora[6], como el proyecto ha constado de 300 horas, el precio del trabajo de la programadora sería de 3309 €.

Para el espacio de trabajo se opta por alquilar un espacio de *co-working*, en España el precio medio del alquiler ronda los 150-200 €/mes [7] por lo que se ha fijado como cantidad 180 €/mes aproximadamente.

## 5.2. Análisis y diseño

En la siguiente sección se detallarán los aspectos más importantes en el análisis y diseño del software a realizar.

### 5.2.1. Especificación de requisitos

#### Requisitos funcionales

**RF-1** Gestión de reproducción de la escena.

**RF-1.1** El sistema debe ser capaz de reproducir la escena con el movimiento de los personajes y las luces escogidas por el usuario.

**RF-1.2** El sistema debe ser capaz de volver a reproducir el movimiento de los personajes después de haber sido reproducida la escena previamente si el usuario lo desea.

**RF-2** Gestión de la iluminación.

**RF-2.1** El sistema debe ser capaz de cambiar el tipo de luz según la configuración escogida.

**RF-2.3** El sistema debe ser capaz de activar y desactivar las luces que siguen el método *McCandless* dependiendo de la elección del usuario.

**RF-2.4** El sistema debe ser capaz de crear focos de iluminación asociados a los actores.

**RF-2.4.1** El sistema debe ser capaz de orientar los focos al movimiento de los actores.

**RF-2.4.2** El usuario debe ser capaz de decidir si los focos siguen al personaje.

**RF-3** Gestión de los actores.

**RF-3.1** El sistema debe ser capaz de crear actores.

**RF-3.2** El sistema debe ser capaz de obtener la trayectoria del actor.

**RF-3.2.1** El usuario debe ser capaz de añadir una posición a la trayectoria del actor mediante el ratón.

**RF-3.2.2** El usuario debe ser capaz de borrar el camino de la trayectoria.

**RF-3.3** El sistema debe ser capaz de mover al actor de acuerdo con su trayectoria.

**RF-3.3.1** El sistema debe ser capaz de cambiar la animación del actor dependiendo del estado del movimiento en el que se encuentre.

**RF-3.3.2** El sistema debe ser capaz de evitar las colisiones entre los distintos actores.

**RF-3.3.3** El sistema debe ser capaz de reproducir un sonido cada vez que el

actor realiza un paso.

**RF-3.3.4** El sistema debe ser capaz de reiniciar la trayectoria llevando al actor a su posición inicial.

**RF-3.4** El sistema debe ser capaz de borrar a los actores de la escena.

**RF-4** Gestión del humo.

**RF-4.1** El sistema debe ser capaz de activar el sistema de partículas del humo dependiendo de la elección del usuario.

### Requisitos no funcionales

**RNF-1** La interfaz debe ser sencilla e intuitiva.

**RNF-2** El programa será una aplicación ejecutable para sistemas operativos Linux.

### 5.2.2. Historias de usuarios y Product Backlog

Historias de usuario	Puntos de historia
1. Como usuario quiero visualizar la escena.	3
2. Como usuario quiero poder elegir entre dos tipos de luces.	3
3. Como usuario quiero poder seleccionar las luces que quiero encendidas en el modo normal.	1
4. Como usuario quiero poder elegir si la luz focal sigue al personaje.	1
5. Como usuario quiero poder añadir actores a la escena.	4
6. Como usuario quiero poder elegir la trayectoria de los actores.	3
7. Como usuario quiero que los actores se muevan conforme a la trayectoria seleccionada.	3
8. Como usuario quiero poder añadir un efecto de humo a la escena.	2
9. Como usuario quiero poder borrar a los actores de la escena.	0
10. Como usuario quiero poder volver a reproducir la escena.	1
<b>Total</b>	<b>21</b>

Tabla 5.2.: Product Backlog

Como el programa a usar no era conocido previamente, se ha optado por seguir una Metodología de Desarrollo Ágil. La adaptación al cambio era imprescindible en vista de que era muy probable que una idea inicial fuese alterada por el desconocimiento de la herramienta. La metodología a seguir ha sido *Scrum*.

Puesto que el equipo de trabajo estaba conformado por una persona, para priorizar y determinar los puntos de historia no se ha tenido que seguir ningún método de con-

senso. En la tabla 5.2 se pueden ver las distintas historias de usuarios priorizadas y sus correspondientes puntos de historia, obteniendo así el *Product Backlog* del proyecto.

El proyecto se desarrollará en iteraciones de dos semanas. Una iteración corresponde con 12 días de trabajo. Suponiendo que se aprovechará el 70 % del tiempo y teniendo en cuenta que un punto de historia corresponde con un día real tenemos que la velocidad sería de  $12 \text{ días/iteración} * 70\% = 8.4 \text{ PH/iteración} \approx 8 \text{ PH/iteración}$ . El proyecto se desarrollará entonces en  $21/8 = 2.625$  dos iteraciones de dos semanas y una iteración de una semana aproximadamente.

La primera iteración incluirá las historias de usuario 1-4, la segunda iteración 5-6 y en la tercera iteración 7-8-9-10.

### 5.2.3. Elementos importados

*Unity* ofrece la posibilidad de importar *assets*, que no son más que elementos que puedes usar en un proyecto. Para ello *Unity* ofrece un espacio donde encontrar e importarlos a tus proyectos asociándolos a la cuenta de *Unity* con la cual te registres. Este espacio se denomina *Unity Asset Store* [8] y se pueden encontrar todo tipo de elementos desde animaciones hasta audios y efectos especiales. Algunos de estos *assets* son gratuitos y otros de pago.

Para este proyecto eran necesarios simplemente tres elementos: el modelo del teatro, el modelo del humanoide que haría de actor/actriz y las animaciones de estar parado y andando de dicho humanoide.

Para el modelo del teatro se ha accedido a *CGTrader* [9], que es un mercado donde se pueden comprar distintos modelos y también se pueden encontrar modelos gratuitos.

El modelo seleccionado se denomina *Opera Hall*, creado por el usuario *vr5531*, este modelo tiene una licencia *Royalty Free License*, que permite su uso en todos los proyectos después de un único pago, sin embargo, en este caso, el precio era gratuito.

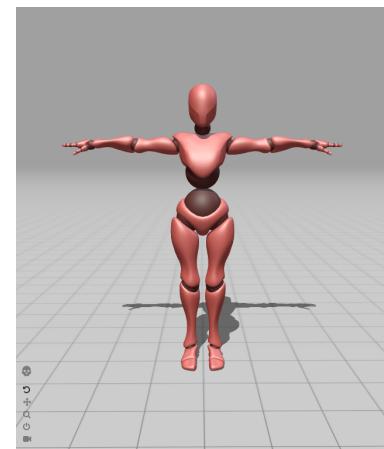
El modelo de persona y las animaciones se han encontrado en el mismo lugar, *Mixamo* [10]. Mixamo es una página en la cual se pueden encontrar animaciones y modelos de personajes articulados. Debido a que la personalización de un personaje en una obra de teatro puede ser muy variada, ya que el vestuario puede ser muy diverso, se ha optado por elegir un modelo lo más escueto, en cuanto a características humanas y detalles físicos, posible. El modelo tiene como nombre *X Bot* y representa una figura femenina humana, recordando a un maniquí de madera usado para ver la anatomía del cuerpo en movimiento en dibujo.

La parte más útil de este modelo es que es *rigged*, es decir, tiene un esqueleto que va a permitir añadirle animaciones humanas. Las animaciones escogidas se han seleccionado desde la misma página y tienen el nombre de *Standard Walk* e *Idle*, que, como su nombre indica, son las animaciones en las que el modelo andará y se quedará parado. Para poder después personalizar el movimiento del personaje, se descargan las animaciones en el sitio, en otras palabras, que no realicen ningún movimiento extra de desplazamiento

del personaje, es decir, sólo incluyen el movimiento de las piernas al andar, pero no el desplazamiento.



(a) Modelo *Opera Hall*



(b) Modelo *X Bot*

Figura 5.2.: Modelos escogidos

Existen otros elementos importados de menor importancia que los comentados previamente, ya que lo que proporcionan es un mayor nivel de detalle o ayudan a crear un estilo específico a la interfaz de usuario. Uno de estos elementos importados, aunque previamente modificado, es el sonido de los pasos en el escenario. Para la obtención del sonido se ha usado *Freesound* [11]. *Freesound* es una página donde podemos encontrar sonidos con licencia *Creative Commons 0*, que, por lo tanto, pueden ser usados en el proyecto. Debido a que la mayoría de escenarios son de madera, se ha escogido un archivo con pisadas en un suelo de madera. Este archivo se llama *footsteps on wood* del usuario *Mydo1*. Para poder aplicarlo en el proyecto, los distintos sonidos de paso han de separarse en archivos distintos, por lo que se hace uso del programa *Audacity* [12]. *Audacity* es un software libre de editor de audio.

Para algunos elementos de la interfaz se ha hecho uso de iconos de Google, así como para la fuente de la interfaz, se ha usado la letra *Courier Prime*, ambos descargados de Google Fonts [13]. Dicha letra se elige debido a que es la letra usada en los guiones de teatro, simulando la letra de una máquina de escribir.

#### 5.2.4. Diseño Gráfico

Como se ha comentado, no se ha modelado desde cero ningún elemento. Sin embargo, se han realizado algunos cambios a los modelos existentes. Al modelo humanoide se le ha cambiado el material, asignándole un color que contrastase más con el fondo del escenario (ver Figura 5.29b), y a la estructura del teatro se le han añadido ciertos elementos, mayoritariamente cajas, que simulaban paredes que no se encontraban al importar el modelo. Por ejemplo, una de las paredes incorporadas que más destaca en la ejecución

del programa es la del fondo del escenario que simula un telón negro.

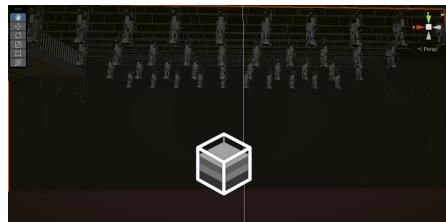


Figura 5.3.: Telón de fondo añadido al escenario.

### 5.2.5. Descripción detallada del proyecto

Podría decirse que para el desarrollo del proyecto podemos hacer una distinción entre distintas partes, que comprenderán diferentes enfoques y herramientas.

#### Aspectos generales

El proyecto ha sido realizado en la versión del editor de Unity 2021.3.10f1, versión *Long Term Support* y se ha utilizado el Universal Render Pipeline.

Principalmente, en Unity existen tres *render pipelines* distintos: Built-in Render Pipeline, Universal Render Pipeline (URP) y High Definition Render Pipeline (HDRP). Aunque también se puede crear uno propio. Los tres mencionados al principio tienen características distintas que hacen que, dependiendo del proyecto, sea conveniente usar uno en lugar de otro.

El *render pipeline* se encarga de realizar las operaciones necesarias para transformar la información de la escena en la imagen 2D que se verá como resultado.

La principal ventaja del Built-in Render Pipeline es que funciona para todas las plataformas y es bastante fiable. Sin embargo, no permite personalizarlo ni tampoco es demasiado eficiente.

Tanto el URP como el HDRP son Scriptable Render Pipelines(SRPs), es decir, pueden ser modificados mediante código. Los SRPs son muy personalizables e incluso pueden ser escritos desde cero. Ya que crear un *render pipeline* es una tarea demasiado costosa, Unity provee estos dos SRPs como plantilla con los que se pueden crear la mayoría de los proyectos.

El URP es ideal para proyectos móviles, web y de realidad virtual, ya que está altamente optimizado para el rendimiento. Produce gráficos bastante decentes y optimizados para una amplia gama de plataformas.

EL HDRP está diseñado para producir gráficos de alta calidad para plataformas de alta gama como consolas u ordenadores para videojuegos, donde la capacidad de procesamiento es alta. Es bastante más complejo de configurar que el URP.

Por la dimensión de este proyecto, la personalización del render pipeline era conveniente, sin embargo, la complejidad que suponía usar el HDRP para la obtención de mejores gráficos tampoco era necesaria por lo que, como se dijo al principio, se usa el Universal Render Pipeline.

## Iluminación

Uno de los aspectos más importantes en el proyecto es la iluminación del escenario. Para ello se ha elegido una propuesta basada en el diseño de iluminación en montaje de obras teatrales [14].

Se podrá elegir entre dos tipos de iluminación.

El primer tipo de iluminación podríamos considerarla la iluminación usual del escenario donde la importancia reside en la visibilidad de la interpretación de los actores en el escenario más que en la creación de un estado de ánimo o un ambiente específico. Para este tipo de iluminación se ha elegido un método común llamado el método de *McCandless* [14].

Este método propone que el escenario ha de dividirse en distintas áreas de actuación, cada una de ellas iluminadas por dos fuentes de luz. Éstas deben colocarse como luces frontales a aproximadamente 90 grados con respecto a la zona a iluminar y situarse entre ellas aproximadamente 45 grados en horizontal. McCandless propuso que cada lámpara tuviese un filtro diferente: uno cálido y otro frío.

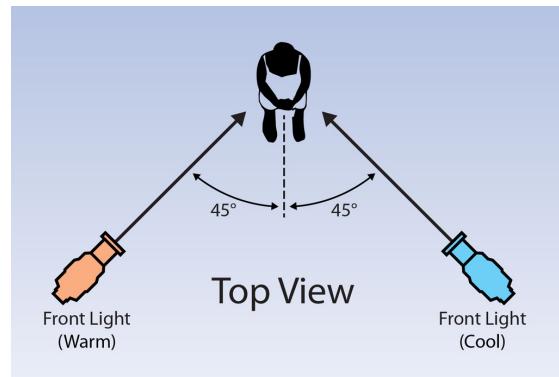


Figura 5.4.: Método de McCandless

Las dos luminarias proporcionan visibilidad al actor, permitiendo que pueda “jugar” tanto a la derecha como a la izquierda y seguir estando en una luz clave. El ángulo entre los focos proporciona una excelente plasticidad y forma al rostro humano. Los colores cálidos y fríos opuestos ayudan a proporcionar interés, contraste y una iluminación natural.

Este método, como se decía previamente, prioriza visibilizar a los actores primero y

luego iluminar el entorno por separado para crear ambiente. A veces no es necesario una iluminación adicional, ya que la propia iluminación de la zona de actuación ilumina el decorado.

Para simular este método, se ha dividido el escenario en seis zonas, tres zonas en la parte frontal del escenario y tres zonas en la parte de atrás. Cada una de las zonas está iluminada por tres luces *spotlights* que simulan los focos de un teatro: dos luces frontales, una de ellas con un filtro cálido y la otra con un filtro frío (simulando el método *McCandless*), y una luz trasera para permitir aún mayor visibilidad a la acción de la escena.

El usuario podrá modificar la iluminación de las zonas, seleccionando aquellas que deseé iluminadas, haciendo así que la iluminación de cada zona sea independiente al resto.

Por otra parte, existirá otro método de iluminación, contrapuesto a éste, en un sentido de uso, es decir, sólo podrá usarse uno de los dos métodos de iluminación, no pudiendo optar por ambos métodos al mismo tiempo.

Este segundo método es una técnica bastante recurrente para el montaje de escenas cortas en las que la dramatización o importancia del actor de manera individualizada es relevante. Se trata de un foco de luz totalmente vertical desde la parte superior del escenario enfocando hacia el actor.



Figura 5.5.: Ejemplo real de luz focal

Para que este tipo de luz exista deben encontrarse actores en la escena, puesto que cada foco estará asociado a un actor. Para ello se usará un *prefab*.

Un *prefab* en *Unity* no es más que un tipo de *asset* que permite almacenar un objeto *GameObject* con sus componentes y propiedades. Los *GameObject* son los objetos fundamentales en *Unity* que representan personajes, props y el escenario. Este tipo de objeto alberga las propiedades más básicas de los elementos, como por ejemplo, si está activo en la escena, la etiqueta a la que pertenece o la posición. Pero realmente su principal función es ser el contenedor de componentes, que son los que implementan la funcionalidad del objeto.

El *prefab* actúa como una plantilla a partir de la cual se pueden crear nuevas instancias del objeto en la escena. Además, cualquier edición hecha a un *prefab* estará reflejada en todas las instancias producidas por él, pero también se puede modificar cada instancia individualmente.

El *prefab* usado en este caso será un foco, creado como los anteriores, a partir de un *spotlight* situado verticalmente en el escenario.

Cada foco será creado cuando se vaya a reproducir la escena, para evitar la creación y destrucción de éstos mientras se están cambiando otras características de la escena, por ejemplo, si se añaden más actores, si se eliminan o si se cambia el modo de luz.

Además se podrán configurar los focos para que sigan la trayectoria del actor, en caso de que éste tenga movimiento, o para que se queden estáticos en la primera posición del actor en la escena. Esto dependerá del efecto que quiera crear el usuario.

### Efectos especiales: Máquina de humo

En el teatro es frecuente el uso de máquinas de humo para crear un ambiente específico en una escena. Estas máquinas se suelen encontrar a ras de suelo en los laterales del escenario.

Para simular este humo se ha hecho uso del *Particle System* de Unity. Este sistema consiste en la emisión de partículas que flotan siguiendo una forma específica por el aire. Estas partículas son imágenes pequeñas o mallas. La modificación de ciertos parámetros de las imágenes seleccionadas como partículas es lo que permitirá crear el efecto de humo.

Para dar una mayor sensación de realidad, este sistema de partículas se ha duplicado, posicionando uno en una parte inferior y el otro en una parte superior. Al sistema de la parte inferior se le ha aplicado un escalado a la mitad para que parezca que el humo está más condensado en esta parte, ya que, como hemos dicho, las máquinas de humo se suelen situar a ras de suelo lo que provoca que la concentración de humo sea mayor en ese lugar. De este modo, la dispersión de la parte superior creará un efecto de mayor naturalidad con respecto al movimiento usual de los gases.

### Movimiento de los actores

Para realizar el movimiento del actor se ha coordinando la animación de andar y estar parado junto con una trayectoria que realizará. Para la trayectoria se seleccionarán puntos del escenario por los que deberá pasar el actor.

Primero, se ha alterado la animación de andar, para que puedan sonar los pasos. Para ello, se han añadido eventos en la animación, denominados *Animation Events*, que permiten seleccionar un frame de la animación y asignarle una función. Dicha función seleccionará uno de los sonidos de pasos de forma aleatoria. Se eligen distintos sonidos para que simule el sonido real de las pisadas, ya que, si siempre fuese el mismo sonido, crearía un efecto en el usuario de irreabilidad.

Para que se puedan reproducir los sonidos, el actor tiene un componente de *Audio Source* que lo único que hace es permitir que el actor sea un “altavoz” que reproduce sonidos. Además se ha convertido en 3D poniendo el *Spatial Blend* a 3D. Que el sonido sea 3D va a permitir que el volumen del audio varíe dependiendo donde esté el *Audio Listener*, que serían los oídos de nuestra escena, encontrándose éstos en el lugar de la cámara, simulando así el lugar del espectador en el teatro. Dentro de un rango, el sonido se escuchará al mismo volumen que el audio original, sin embargo, pasado este rango, el volumen descenderá de manera logarítmica, simulando el sonido en el mundo real.

Para crear una mayor sensación de realidad, se ha añadido al escenario una *Reverb Zone* que no es más que una zona en la que los distintos audios que se reproduzcan dentro de ella tendrán una pequeña distorsión. Unity ofrece algunos efectos por defecto, incluyendo uno para auditorios, por lo que ese ha sido el seleccionado.

Para poder controlar las animaciones de andar y estar parado se ha creado un *Animation Controller*, este controlador permite alterar entre distintos *Animation Clip*, que en este caso serían las animaciones importadas, especificando las transiciones entre éstas. Dos parámetros *trigger* han sido creados, para controlar mediante código cuándo debe pasar de una animación a otra el modelo del actor.

En segundo lugar, está la selección de los puntos del escenario. Se ha creado un elemento de la interfaz con forma de rectángulo que simula en dos dimensiones una vista del escenario desde arriba. Se ha controlado el lugar donde el usuario pincha con el ratón, calculando las correspondientes coordenadas en el escenario.

Cuando existen varios actores en escena es importante que no se choquen entre ellos cuando se muevan. Por lo que, para el movimiento de los actores, se ha decidido que éstos sean *NavMesh Agent*. Esto permite que los actores puedan calcular mediante Inteligencia Artificial un camino a un punto, evitando los obstáculos y teniendo que recalcular el camino cuando sea necesario. Para ello, se ha elegido la zona donde puede navegar el agente y se ha añadido un *Collider*, que simplemente define la forma del actor para las colisiones físicas, para que no choquen entre ellos.

Este actor con los componentes mencionados previamente se ha almacenado en un *prefab*.

### 5.2.6. Bocetos interfaz de usuario

Cada boceto de la interfaz está numerado, siendo el número el que está recuadrado. Si la función de algún botón lleva a otro boceto se muestra el número hacia el que va con el número redondeado.

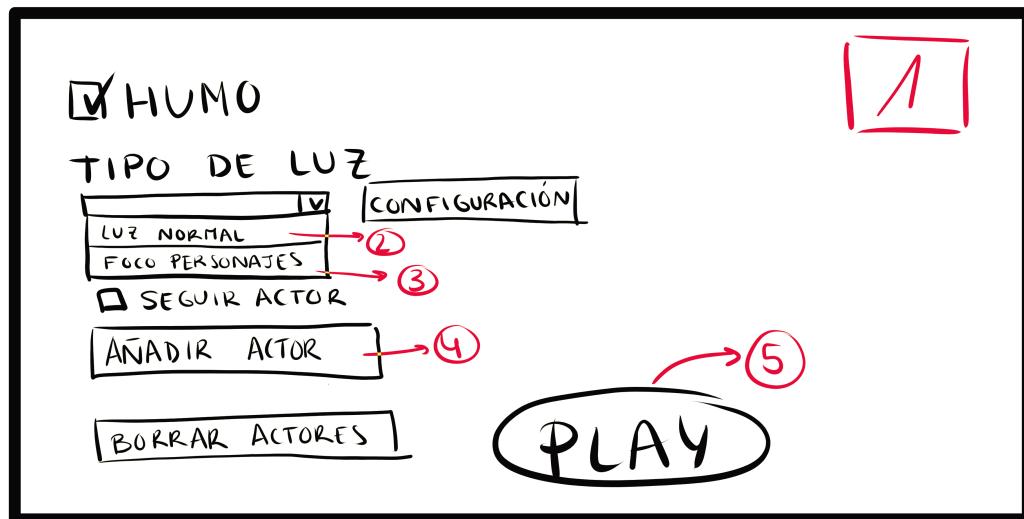


Figura 5.6.: Boceto 1

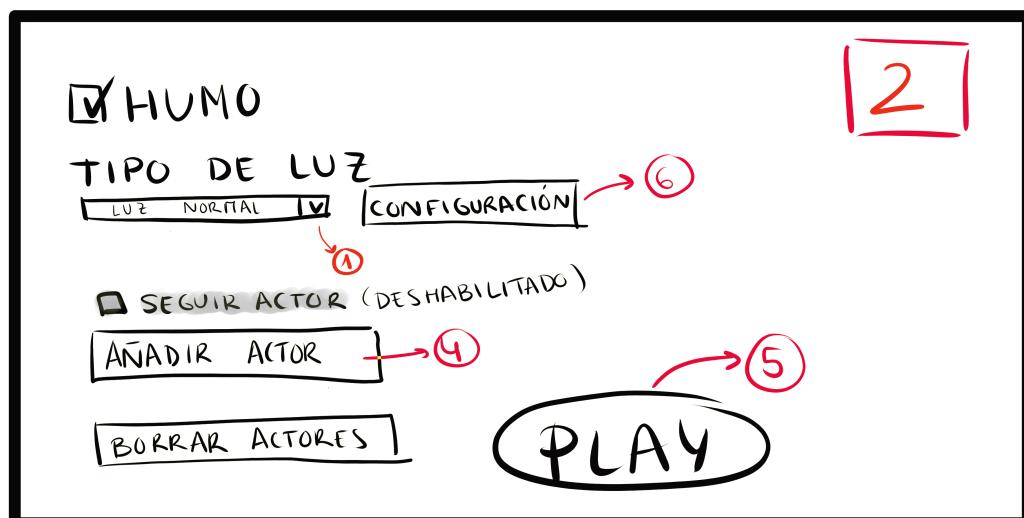


Figura 5.7.: Boceto 2

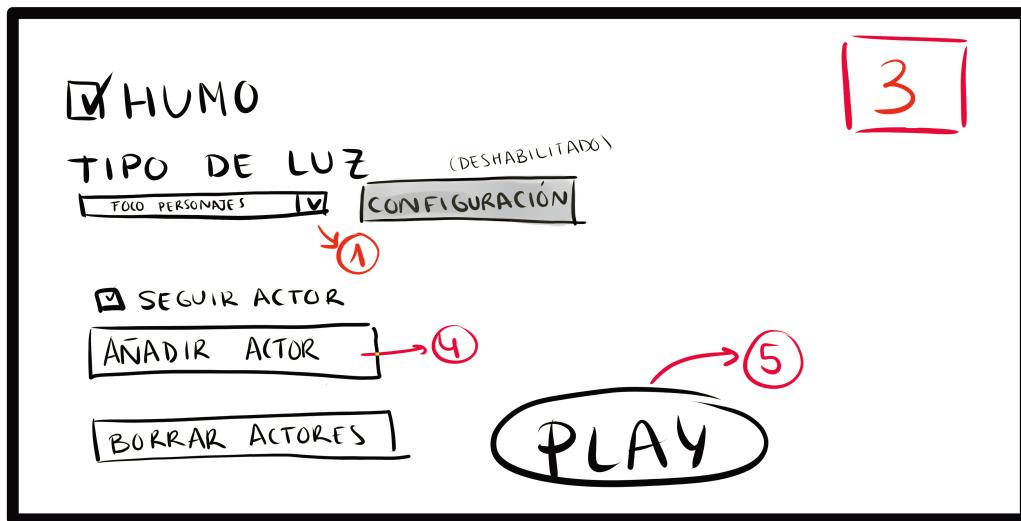


Figura 5.8.: Boceto 3

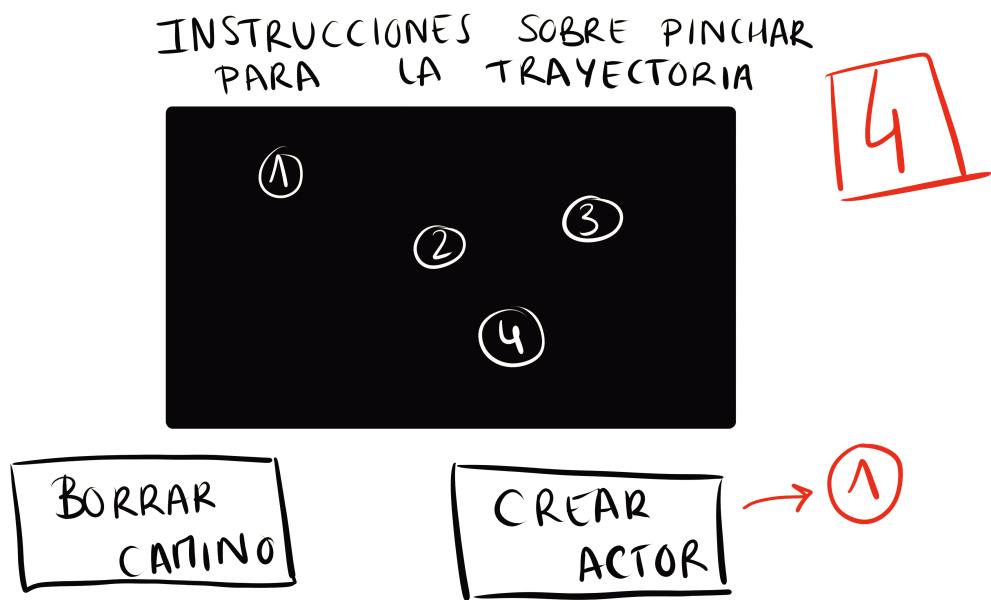


Figura 5.9.: Boceto 4

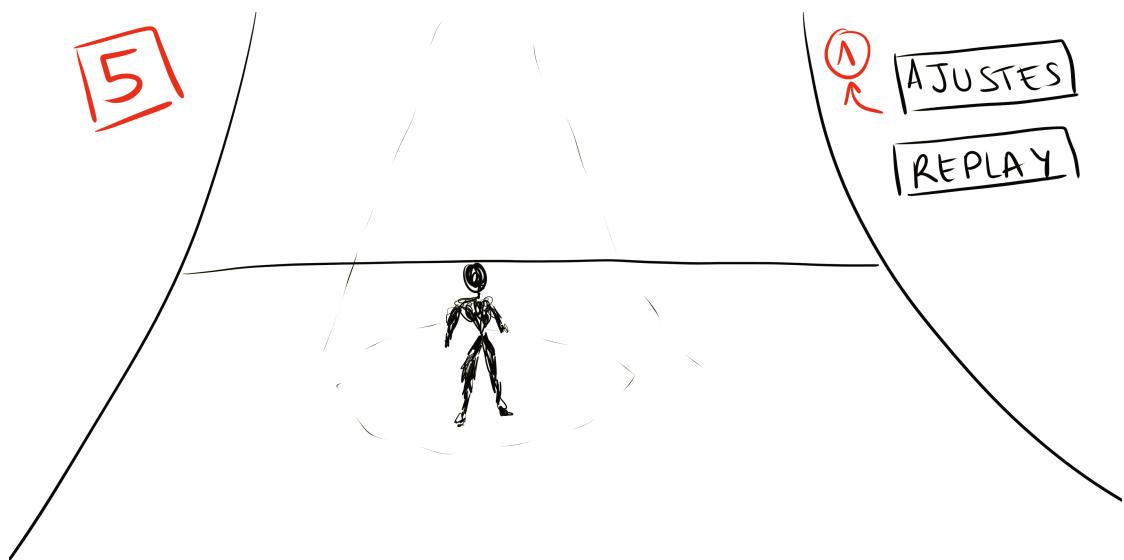


Figura 5.10.: Boceto 5

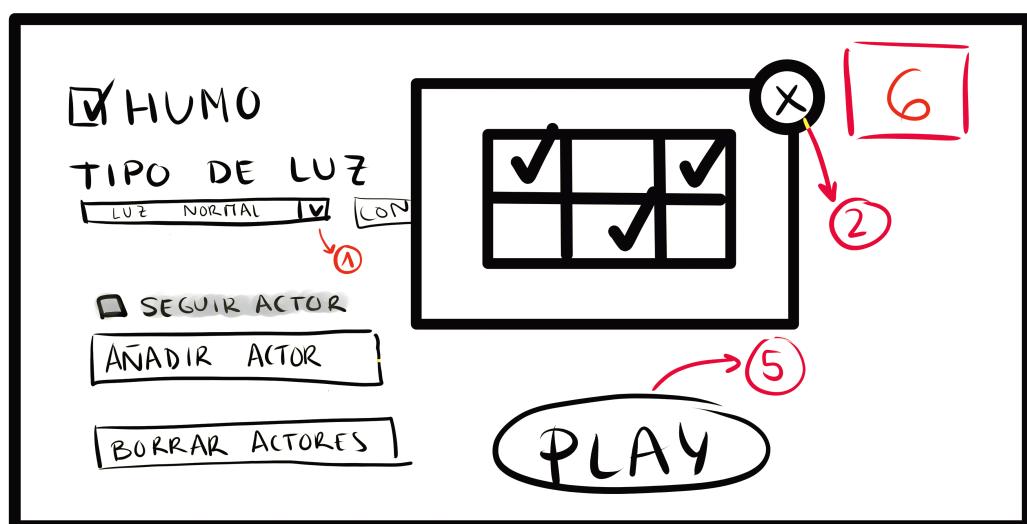


Figura 5.11.: Boceto 6

### 5.2.7. Diagrama de Clases

Para el proyecto se han creado cinco clases nuevas que no pertenecían a las ya existentes en *Unity*. Todas ellas heredan de `MonoBehaviour` que es la clase base de todos los *scripts* en *Unity*. La clase `MonoBehaviour` implementa la funcionalidad básica para cualquier objeto, permitiendo que cada *script* nuevo sobreescriba dicha funcionalidad. Dentro de esta clase se encuentran las funciones `Start` y `Update`. La función `Start` se llama antes del primer *frame* y la función `Update` se llama una vez por cada frame. En la sección 5.3.1 se habla más del ciclo de vida de un *script* en *Unity*.

Cada una de las clases nuevas se ha creado para albergar una funcionalidad específica de un objeto o funciones con temática similar que se necesitaban en el programa. A continuación se explican cada una de estas clases.

La clase `Manager` gestiona los aspectos relacionados con la escena a reproducir, como por ejemplo la creación de los focos centrados en los personajes, la creación de los actores, la reproducción del movimiento de los actores, etc. Además, contiene una lista con los actores y los focos que se encuentran en escena, para poder llevar un control de éstos. Esta clase está asociada a un objeto vacío de la escena, por lo que la funcionalidad de ésta no está asociada a ningún objeto como tal. También contiene los *prefab* del actor y el foco para poder crearlos. Tiene una variable para controlar el modo de iluminación en el que se encuentra el programa.

La clase `UIManager` solamente controla el habilitado de los botones de configuración y el *toggle* del foco dependiendo del modo de luz seleccionado en el *dropdown*. Al igual que en el caso anterior esta clase está asociada a un objeto vacío. Como variables contiene el *toggle*, el botón que tiene que ir habilitando y el elemento de la interfaz de la configuración de la luz para desactivarla dependiendo del modo de luz.

La clase `SpotMove` se encarga del movimiento del foco con respecto al actor al que esté asociado, para ello, esta clase es un componente de los focos verticales. Como variables contiene al actor al que está asociado y una variable para determinar si debe seguir a éste. Dicho movimiento lo realizará en el método `Update`.

La clase `ActorManager` controla el movimiento del actor, cambiando la animación asociada al *Animator Controller* de éste e indicando también la trayectoria que debe seguir según las posiciones almacenadas del recorrido. Esta clase es un componente de los actores. Contiene una lista con los audios de las distintas pisadas sobre el escenario, una referencia al altavoz del actor, una lista con las posiciones por las que debe pasar, un cuaternión que indica la rotación que tiene en la primera posición, el *Animator Controller* del actor y una variable que controla el índice de la posición del camino a la cual debe ir. También posee métodos para comenzar a realizar el camino, para volver a la posición inicial y para reproducir el sonido de las pisadas.

Por último, la clase `StageManager` gestiona las posiciones donde el usuario pincha en el elemento de la interfaz de usuario que simula el escenario, calculando las correspondientes coordenadas en el escenario. Para poder manejar el evento del ratón esta clase

debe implementar la interfaz `IPointerClickHandler`. Como variables contiene el *prefab* del elemento de la interfaz de usuario que va a ser usado como señal del camino que va a recorrer el actor, así como un vector con el conjunto de todos los que forman el camino. También contiene los valores del escenario necesarios para la obtención de coordenadas. Además, guarda en una variable una referencia al elemento de la clase `Manager` para poder realizar la creación del actor a partir de las posiciones almacenadas por los *clicks* del ratón. En cuanto a los métodos, contiene un método para realizar la transformación de coordenadas en la posición del escenario, también sobreescribe el método `OnPointerClick` para poder gestionar la información del ratón e incluye métodos para borrar el camino almacenado y para poder crear un actor con las posiciones del camino correspondiente.

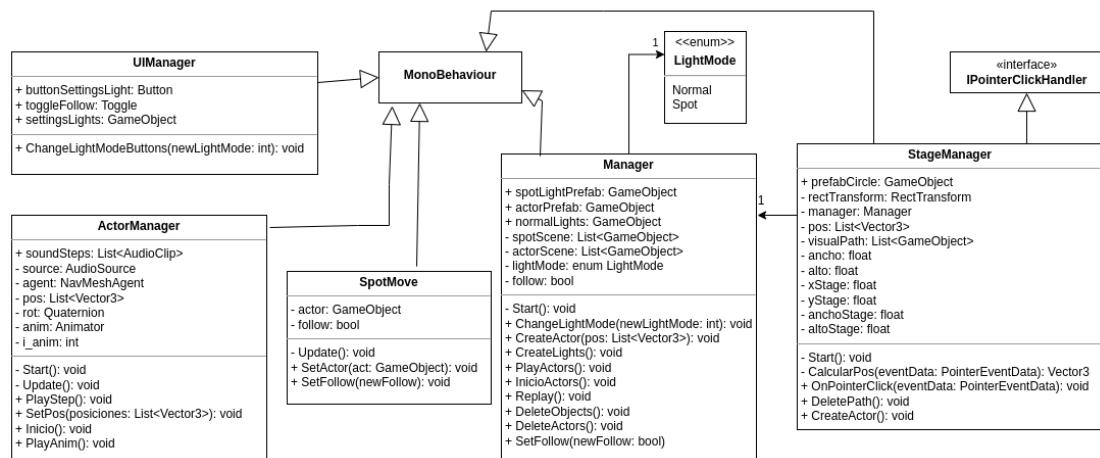


Figura 5.12.: Diagrama de Clases

### 5.2.8. Principales algoritmos

Uno de los algoritmos a comentar es la obtención de las coordenadas del escenario mediante la gestión de las acciones con el ratón del usuario. Este algoritmo se encuentra en la clase `StageManager`.

La información del ratón en la pantalla va a depender del tamaño de ventana en la que se esté visualizando el programa. Sin embargo, la interfaz de usuario se escala siempre a una misma proporción que es de 1600:900. Esto es gracias al componente *Canvas Scaler*, por lo que podemos tener siempre una referencia del lugar de la pantalla donde se encuentra el elemento de la interfaz a usar. El elemento de la interfaz, en este caso, es una imagen rectangular que simula la parte del escenario donde podemos añadir a los actores. Este rectángulo tiene las mismas proporciones que dicha parte.

Las coordenadas del escenario donde se podrá añadir al actor son se pueden ver en la Figura 5.13.

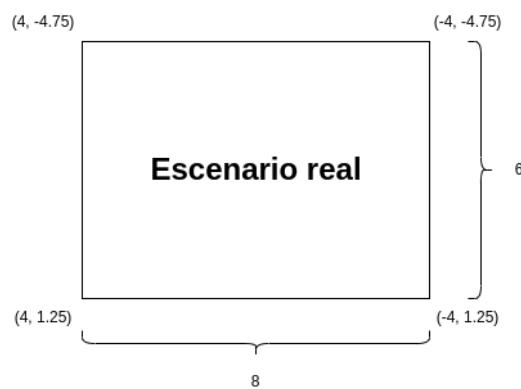


Figura 5.13.: Dimensiones y coordenadas del escenario

Estos datos están guardados en unas variables en la clase `StageManager`.

La imagen de la interfaz de usuario que simula al escenario se encuentra en una posición de la pantalla dada por el componente `RectTransform`, que es como el `Transform` de los elementos 2D, es decir, el que contiene la posición, el alto, el ancho, la rotación, etc.

Para calcular las coordenadas de la esquina superior izquierda de la imagen podemos calcularla a partir de los valores almacenados en el `RectTransform`. La posición de *y* y de *x* tienen un valor con respecto al centro de la pantalla (que sería el centro del *Canvas*, que es el lugar que almacena los elementos de la interfaz de usuario), en otras palabras, representa el desplazamiento del centro de la imagen con respecto al centro de la pantalla. Para calcular la coordenada *x* de la esquina superior izquierda habría que sumarle al ancho de la pantalla la mitad del ancho de la imagen y además sumarle el desplazamiento de la *x* (que puede ser tanto positivo como negativo), de la misma manera se calcularía la coordenada *y* cambiando el ancho por alto.

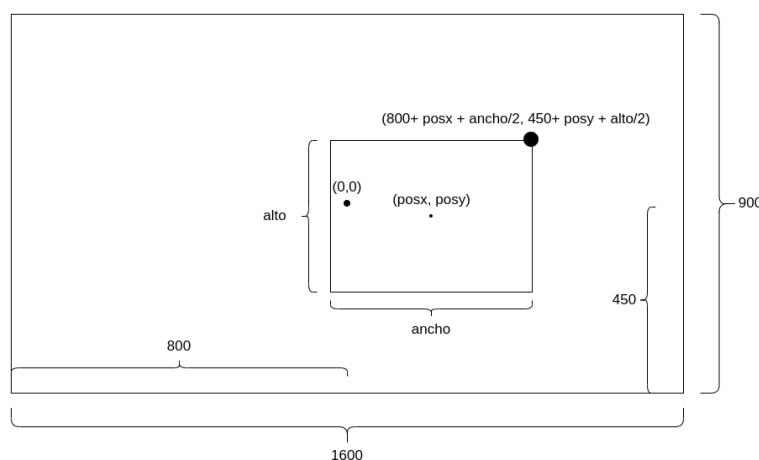


Figura 5.14.: Esquema para el cálculo de la esquina superior izquierda

Es decir, tenemos que las coordenadas  $x$  e  $y$  del escenario de la imagen corresponden con:

$$x_{\text{imagen}} = 800 + \text{posx} + \text{ancho}/2,$$

$$y_{\text{imagen}} = 450 + \text{posy} + \text{alto}/2.$$

Para poder obtener las coordenadas del ratón debemos añadir un componente *Event trigger* de *Pointer click* a la imagen, esto permitirá que cada vez que el usuario pinche en la imagen podamos gestionar la información de este evento en la función `OnPointerClick`. Cuando tenemos las coordenadas del ratón debemos convertirlas en coordenadas dentro de nuestra pantalla de tamaño  $1600 \times 900$ , ya que, como hemos comentado previamente el tamaño de la ventana donde se ejecuta el programa puede variar, pero siempre mantendrá esa escala. Para obtener estas coordenadas simplemente debemos dividir por el valor que tenga la pantalla de ancho/alto y multiplicarlo por 1600 y 900 respectivamente. Cuando ya tenemos estos valores podemos calcular la proporción de la pantalla en la que están con la esquina superior izquierda.

$$x_{\text{proporción}} = (x_{\text{imagen}} - x) / \text{ancho}_{\text{imagen}},$$

$$y_{\text{proporción}} = (y_{\text{imagen}} - y) / \text{alto}_{\text{imagen}}.$$

Tras obtener esta proporción ya podemos calcular las coordenadas en el escenario como:

$$x = -4 + 8 * x_{\text{proporción}},$$

$$y = -4.75 + 6 * y_{\text{proporción}}.$$

En realidad en el programa la coordenada  $y$  equivaldría a la  $z$ . Este cálculo se encuentra en la función `CalcularPos`.

## 5.3. Implementación

En este apartado se comentará la implementación de la distinta funcionalidad del programa incluyendo extractos de código de éste.

### 5.3.1. Ciclo de vida de un script de Unity

Para entender cómo funciona *Unity* se va a explicar el ciclo de vida de ejecución de un script en *Unity*.

Las funciones `Start` y `Update` son clave en el ciclo de vida de un *script*. Antes de la actualización del primer *frame* se llama a la función `Start` (sólo si el componente del script está activado) y la función `Update` es llamada una vez por *frame*.

El ciclo de vida de un *script* se divide en doce etapas: la etapa del editor, la iniciali-

zación, la etapa de actualización de las características físicas, los eventos de entrada, la lógica del juego, el renderizado de la escena, el renderizado del *gizmo*, la renderización de la interfaz de usuario, la finalización del *frame*, la pausa, la activación/desactivación y la etapa de cierre.

Las primeras funciones que se llaman son en el editor de *Unity*. Usualmente se llama a la función **Reset**. Esta función es llamada para inicializar las propiedades del *script* cuando es por primera vez adjuntado al objeto.

Tras esto se pasa a las funciones de inicialización que sólo se invocan una vez por *script*. En la etapa del cálculo de las físicas las funciones pueden ser llamadas más de una vez por *frame* si el intervalo de tiempo fijado es menor que el tiempo real en el que se actualiza el *frame*.

Posteriormente se siguen las distintas etapas comentadas previamente. El siguiente gráfico muestra el ciclo de vida del script

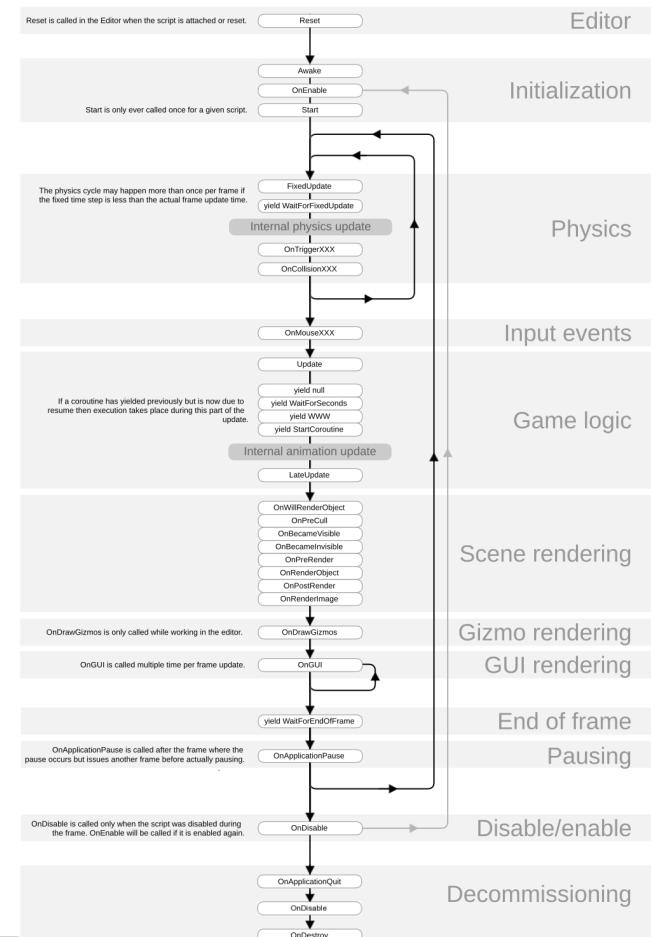


Figura 5.15.: Ciclo de vida de un *script* de *Unity*. Extraído de <https://docs.unity3d.com/es/530/Manual/ExecutionOrder.html>

### 5.3.2. Iluminación

*Unity* tiene dos modos de calcular la iluminación en la escena, una de ellas se denomina *Lightmapping*.

Se denomina *Lightmapping* al proceso de precalcular el brillo y las sombras de las superficies en una escena y almacenar el resultado en una textura llamada *lightmap*, a las luces que tienen este tipo de cálculo se dice que son *baked*. Este método para calcular la iluminación mejora el rendimiento del programa ya que toda la información se guarda antes de ejecutarse, lo que permite reducir el consumo de recursos hardware en la ejecución. Sin embargo, este tipo de iluminación no permite cambios en tiempo de ejecución.

El segundo método para calcular la iluminación se denomina *RealTime*, en este método los cálculos de iluminación se realizan en tiempo de ejecución, una vez por *frame*. Las propiedades de las *RealTime Lights* se pueden cambiar en tiempo de ejecución.

Existe un tercer modo de cálculo que combina los dos anteriores: *Mixed Lights*. Este tipo de luz se puede usar para combinar sombras dinámicas con iluminación *baked* de la misma fuente de luz, o cuando se quiera que una luz aporte iluminación directa en tiempo real e iluminación indirecta *baked*. Dado que las *Mixed Lights* siempre combinan al menos algo de iluminación *real-time* y algo de iluminación *baked*, siempre implican más cálculos en tiempo de ejecución que la iluminación totalmente *baked*, y un mayor uso de memoria que la iluminación totalmente *real-time*.

En este caso era importante el poder cambiar las propiedades de las luces, ya que, por ejemplo, algunas tendrían que cambiar su posición respecto al movimiento del actor y además se iba a poder cambiar entre dos tipos de iluminación, por lo que las sombras iban a estar cambiando a lo largo del programa, más aún con la trayectoria y la inserción de nuevos personajes. Así que la iluminación del programa será *real-time*.

Como se ha comentado previamente existen dos modos de luces. Para diferenciar estos dos modos se va a crear un *enum*, cuyo valor coincida con el índice de la iluminación que se encuentra en el *dropdown* de la interfaz de usuario.

```

1 enum LightMode
2 {
3     Normal = 0,
4     Spot = 1
5 }
```

Cuando se cambia el tipo de luz se deben deshabilitar los elementos de la interfaz que permiten la configuración del modo de luz contraria y activar la propia. Además, si se cambia al modo de luz de foco a personaje, hay que desactivar la ventana donde se seleccionan las zonas de actuación iluminadas por si acaso el usuario no la ha cerrado previamente. Esto se realiza en el método *ChangeLightModeButtons* de la clase *UIManager* cuando se cambia el valor del *dropdown* correspondiente al tipo de luz.

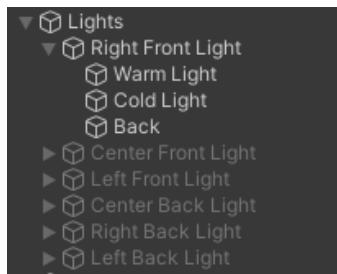
```

1 public void ChangeLightModeButtons(int newLightMode){
```

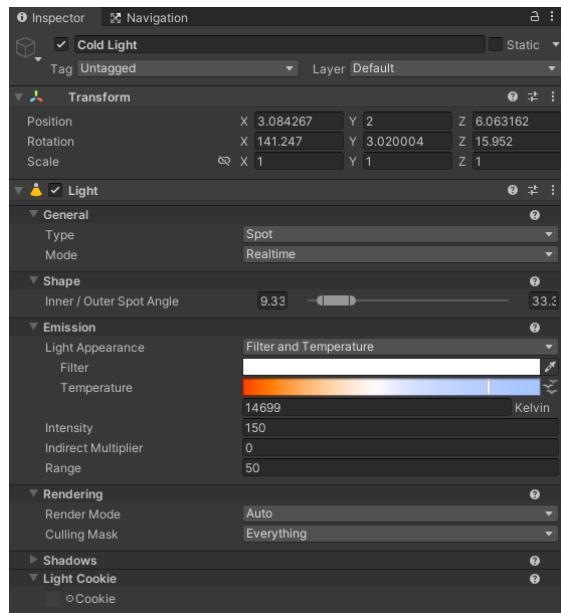
```

2     if ((LightMode)newLightMode == LightMode.Normal){
3         buttonSettingsLight.interactable = true;
4         toggleFollow.interactable = false;
5     }
6     else{
7         buttonSettingsLight.interactable = false;
8         toggleFollow.interactable = true;
9         settingsLights.SetActive(false);
10    }
11 }

```



(a) Jerarquía de los *GameObjects*.



(b) Ejemplo de una luz con filtro frío.

Figura 5.16.: Luces del método *McCandless* en Unity.

Asimismo, se debe cambiar el modo de luz en la clase *Manager* y desactivar o activar las luces normales que siguen el método de *McCandless*. La variable *normalLights* contiene un *GameObject* vacío que sirve de objeto padre para el resto de grupos de luces. Estos grupos son seis, las seis zonas de actuación. Dentro de cada grupo hay tres luces *spotlights*, una cálida, una fría y una trasera, que se activan o desactivan dependiendo de los *toggles* que se encuentran dentro de la configuración de estas luces. Al activarse o desactivarse dentro de cada uno de los objetos hijos cuando se vuelve al mismo modo de luz se guardará la configuración previa. Esto se realiza en el método *ChangeLightMode* de la clase *Manager* que es llamado cuando se cambia el valor del *dropdown* correspondiente al tipo de luz.

```

1 public void ChangeLightMode(int newLightMode){
2     lightMode = (LightMode)newLightMode;
3     if (lightMode == LightMode.Normal){
4         normalLights.SetActive(true);

```

```

5     }
6     else
7     {
8         normalLights.SetActive(false);
9     }
10 }
```

Para aportar un toque puramente estético al programa se ha añadido post-procesado. Esto funciona como un filtro para la cámara. Para ello se debe crear un volumen, este volumen servirá para indicar el lugar en el que la cámara va a usar este efecto. En este caso, la cámara nunca cambia de lugar por lo que podemos crear un volumen global. Dentro de este volumen se crea un perfil para el post-procesado. Simplemente se ha alterado un poco el color añadiendo contraste para que la iluminación quedase mejor. Tampoco se debe abusar de este tipo de filtros ya que puede ralentizar la ejecución.

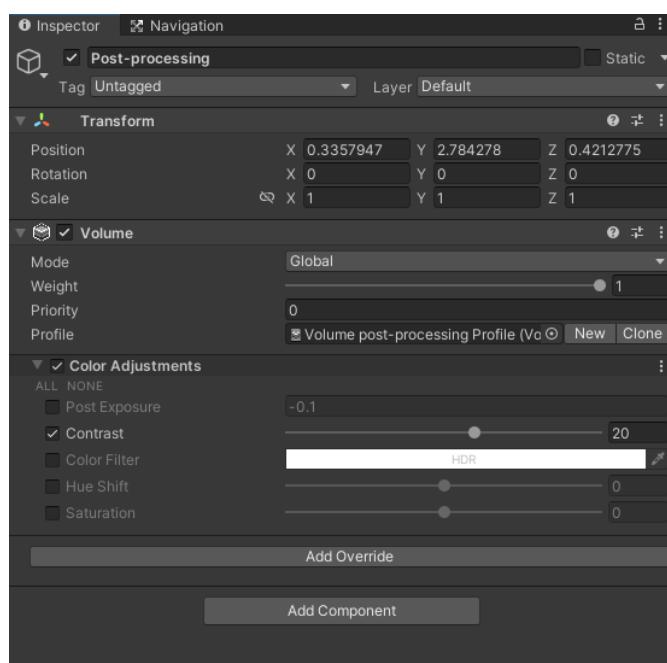


Figura 5.17.: Post-procesado.

### Creación de los focos

Como se dijo previamente, para los focos verticales que enfoquen a un actor se usa un foco *spotlight* guardado como un prefab. Este objeto se almacena con el componente del *script* que permitirá su movimiento. Cuando el usuario quiere reproducir la escena es cuando se crean los focos. Esto se encuentra en el método *CreateLights* de la clase *Manager*. Para crearlos se usa la posición del actor (solamente las coordenadas *x* y *z* ya que el foco deberá encontrarse a la misma altura a la que estaba el *prefab*) al que

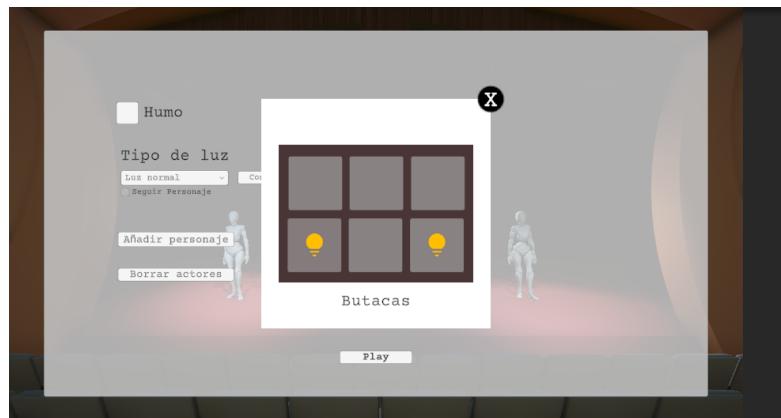


Figura 5.18.: Configuración de la luz normal por el usuario.

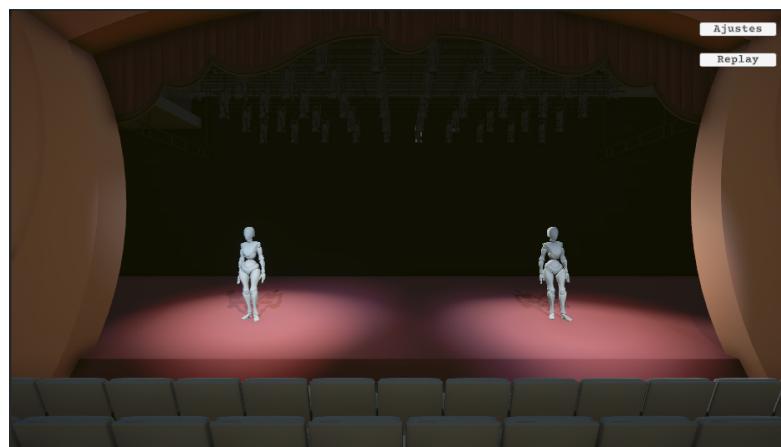


Figura 5.19.: Luz normal sobre el escenario.

están asociados y se les indica si deben seguir al actor, esta información se guarda en la variable `follow`. Este método será llamado cuando el usuario pulse el botón de *Play*.

```

1 public void CreateLights(){
2     // Si el modo de luz es el de foco en personaje
3     if (lightMode == LightMode.Spot){
4         Vector3 pos;
5         foreach (GameObject actor in actorScene){
6             // Vamos a crear un foco en el lugar donde se encuentra el actor
7             pos = new Vector3(actor.transform.position.x, spotLightPrefab.
8                 transform.position.y, actor.transform.position.z);
9
10            // Lo anadimos al vector de focos
11            spotScene.Add(Instantiate(spotLightPrefab, pos, spotLightPrefab.
12                transform.rotation));
13            // Le anadimos al actor y el modo de movimiento
14            spotScene[^1].GetComponent<SpotMove>().SetActor(actor);
15            spotScene[^1].GetComponent<SpotMove>().SetFollow(follow);
16        }
17    }
18}
```

```
15 }
16 }
```

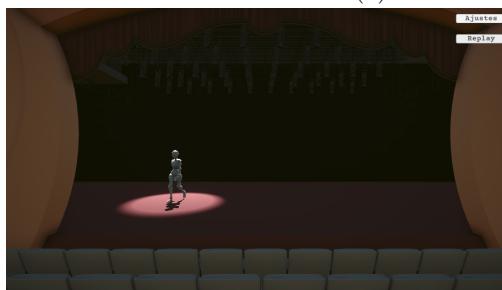
El movimiento de los focos debe realizarse en el método `Update` de la clase `SpotMove`. En este método se debe comprobar si el foco está configurado para que siga al actor y que tiene un actor asociado. Si es así, el foco se posiciona mirando al actor. Como se dijo previamente, este método es llamado una vez por frame.

```
1 void Update(){
2     if (follow && actor != null){
3         transform.LookAt(actor.transform);
4     }
5 }
```

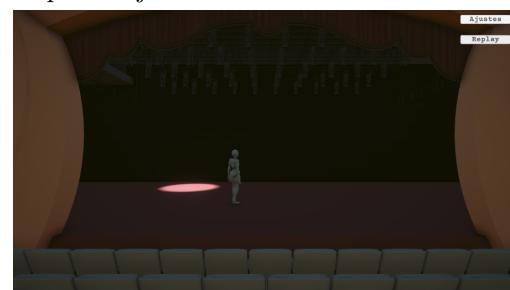
Encima de dichos focos se encuentra un elemento decorativo del modelo del escenario que simulan los focos reales de éste. Asociar una fuente de luz a cada foco y mover ambos elementos era más costoso que crear los focos de la forma explicada previamente. Sin embargo, era interesante el detalle visual que proporcionaban éstos. Para que los nuevos focos no creasen una sombra de los anteriores, creando un efecto alejado de la realidad, se han puesto los objetos del modelo en una nueva capa y se indica en el *prefab* del foco que no debe proyectar sombras sobre dicha capa.



(a) Modo de luz foco en personajes.



(b) Foco con el modo de seguimiento activado.



(c) Foco con el modo de seguimiento desactivado.

Figura 5.20.: Focos sobre los personajes.

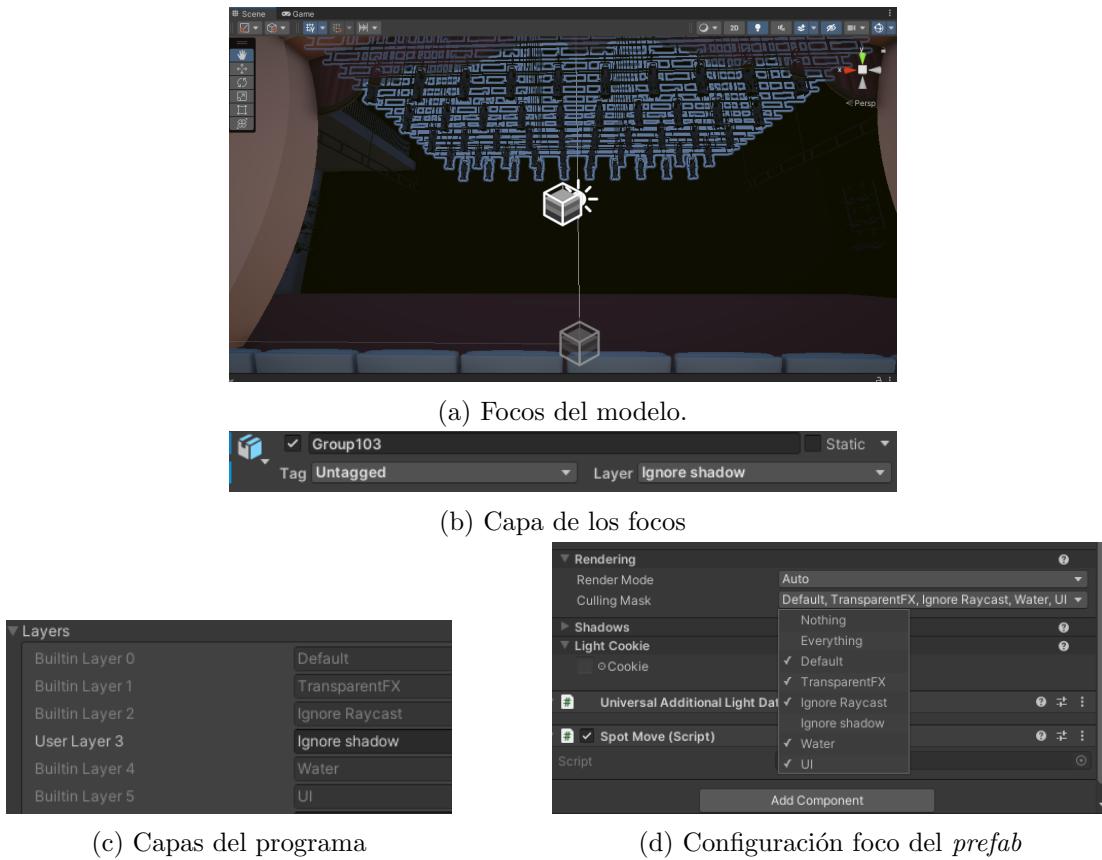
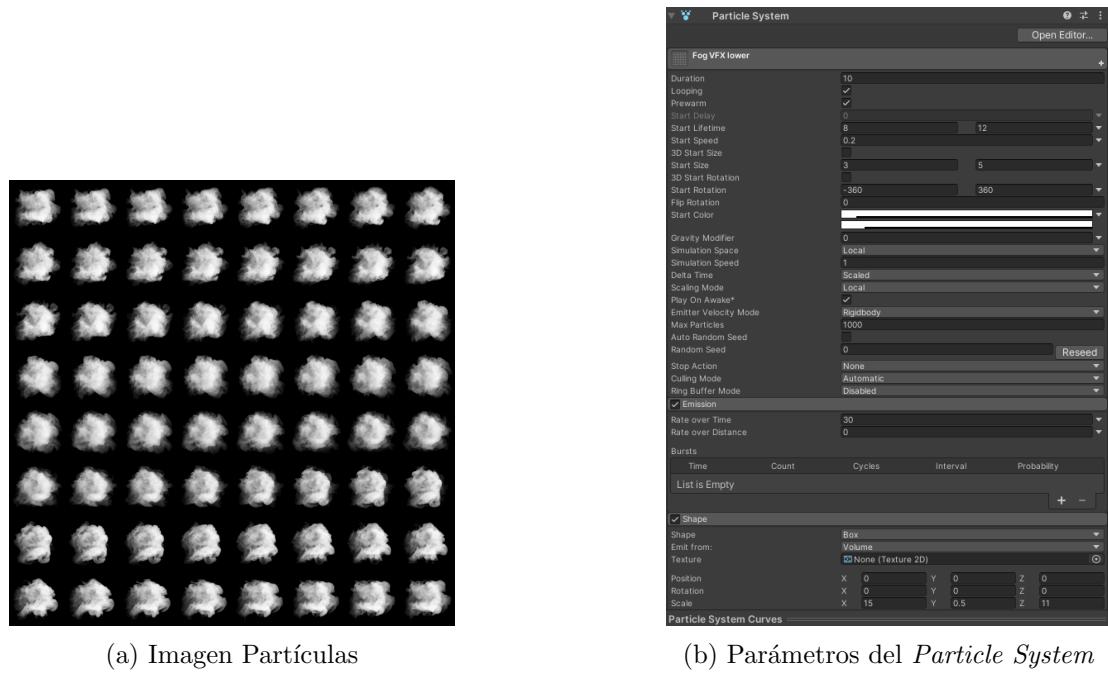


Figura 5.21.: Capas para la luz

### 5.3.3. Creación del sistema de partículas

Para crear el sistema de partículas, primero se debe crear un material específico para las partículas. En este caso lo hemos hecho con un *shader* de partículas lit, es decir, un *shader* específico para partículas que va a permitir que éstas reaccionen a la luz. En este material asignamos la imagen que va a conformar el aspecto que tengan.

El material creado se debe asignar al sistema de partículas creado. En la imagen hay 64 tipos de partículas distintas. Cada partícula irá pasando por cada una de ellas de forma aleatoria, para dar una mayor realidad al movimiento del humo. La forma en la que se emiten las partículas es una caja que ocupa el escenario. Para buscar la realidad en este sistema se alteran distintos parámetros. Uno de ellos es la velocidad a la que se mueven, ésta ha de ser muy lenta. También se añaden componentes que cambian a lo largo del tiempo como la rotación de las partículas y el color, permitiendo que al principio y al final la transparencia de las partículas sea total, para recrear un *fade in* y *out*. Cada partícula se inicializa con un tamaño y una transparencia distintas.

Figura 5.22.: *Particle System*

(a) Humo con la luz normal y personajes.



(b) Humo en el modo foco a personaje.



(c) Humo sin personajes con luz normal.

Figura 5.23.: *Humo*.

Para activar ese humo, en la interfaz de usuario se crea un *toggle*, que activará o desactivará ambos sistemas (ya que como se comenta en la introducción se han duplicado para crear una mayor sensación de realidad).

#### 5.3.4. Creación de los actores

Para crear a los actores primero debemos crear la trayectoria. Para ello, las posiciones del escenario son calculadas como se ha explicado previamente. Cuando la posición ha sido calculada, se debe añadir el *feedback* visual para que el usuario sepa dónde ha pinchado y el número de posición que es, para poder crear un camino visual. El siguiente método pertenece a la clase **StageManager** y es llamado cada vez que el usuario pincha en la imagen que simula el escenario.

```
1 public void OnPointerClick(PointerEventData eventData){  
2     // insertamos la posicion al vector  
3     pos.Add(CalcularPos(eventData));  
4     // creamos el circulo  
5     visualPath.Add(Instantiate(prefabCircle));  
6     visualPath[^1].transform.SetParent(this.transform.parent);  
7     visualPath[^1].GetComponent<RectTransform>().position = new Vector3(eventData.  
8         position.x,eventData.position.y,0);  
9     // cambiamos la etiqueta del circulo  
10    visualPath[^1].GetComponentInChildren<TMP_Text>().text = visualPath.  
        Count.ToString();  
11 }
```

Como el círculo es un elemento de la interfaz, debemos asignarle el padre el mismo que el del escenario para que pueda verse por pantalla encima de éste. También le asignamos la posición y le cambiamos el valor del número por el tamaño del vector de círculos ya que, al ser el último círculo creado, coincidirá con éste.

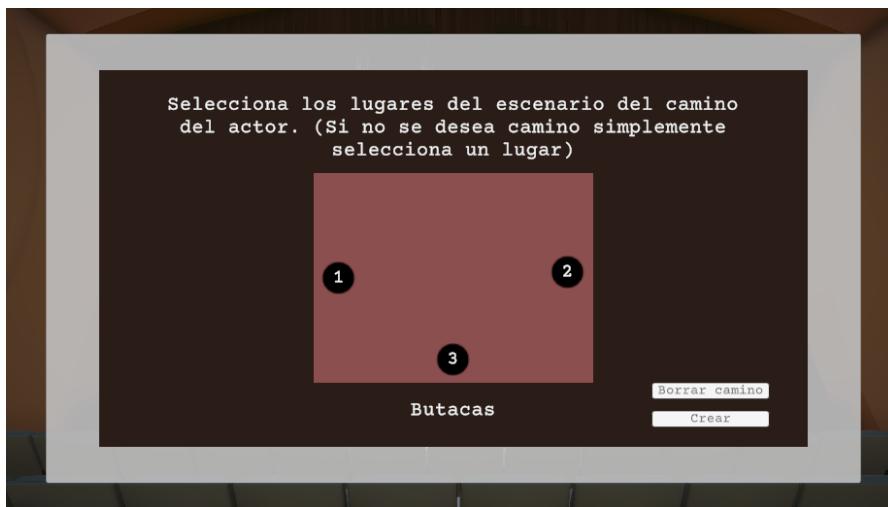


Figura 5.24.: Selección de la trayectoria para un actor.

Cuando todas las posiciones han sido almacenadas y el usuario selecciona el botón de crear al actor, se debe borrar el camino visual y vaciar el vector que contenía dicho camino, además se debe borrar el vector de posiciones. Es la clase `Manager` la que se va a encargar realmente de crear al actor. Esto se realiza en el método `CreateActor` de la clase `StageManager`.

```

1 public void CreateActor(){
2     foreach (GameObject paso in visualPath)
3     {
4         Destroy(paso);
5     }
6     visualPath.Clear();
7
8     manager.CreateActor(pos);
9     pos.Clear();
10 }
```

Para instanciar al actor debemos comprobar que el vector de posiciones contiene al menos una posición. Si es así, se instancia al actor en la primera posición y además le asignamos el vector de posiciones que conformará la trayectoria de éste. Esto se encuentra en el método `CreateActor` de la clase `StageManager`.

```

1 public void CreateActor(List<Vector3> pos){
2     if(pos.Count>0){
3         actorScene.Add(Instantiate(actorPrefab, pos[0], actorPrefab.transform.
4             rotation));
5         // le anadimos las posiciones al ultimo actor creado para ello debemos
6         // acceder al metodo que se encuentra en la clase Actor Manager
7         actorScene[^1].GetComponent<ActorManager>().SetPos(pos);
8     }
9 }
```

### 5.3.5. Movimiento del actor

Para que el movimiento del actor se realice de la manera más realista posible se deben tener en cuenta distintos factores. Para ello, cuando se crea el actor, algunas variables o componentes han de ser referenciadas. Esto va a realizarse antes del primer *frame* en el que se encuentre el actor, por lo que debe encontrarse en el método `Start` de la clase `ActorManager`.

```

1 void Start()
2     {
3         source = GetComponent< AudioSource >();
4         agent = GetComponent< NavMeshAgent >();
5         anim = GetComponent< Animator >();
6         rot = transform.rotation;
7         // primero el agente debe estar parado
8         agent.isStopped = true;
9     }
```

Las tres primeras líneas van a permitir que podamos realizar distintas funcionalidades de los componentes que tiene el actor mediante código. También debemos almacenar la

rotación inicial que tiene el actor para cuando necesitemos que vuelva a ésta, la posición inicial estaría guardada en el vector de posiciones por lo que no es necesario almacenarla. Además, cuando se cree nuestro actor necesitamos que esté parado (ya que comenzará el movimiento cuando el usuario pulse el botón de *Play*) por lo que debemos indicárselo al componente **agent** que es el que se encarga del movimiento del actor.

## Sonido de pasos

Uno de los detalles que introduce el realismo en el movimiento es el sonido de los pasos. Previamente a éste, ha de importarse la animación de andar. Cuando está animación se ha importado debe especificarse que es humanoide. Tras esto, debemos añadirle eventos en la animación para que se llame a la función **PlayStep** del *script* asociado al modelo que reproduzca la animación cuando ésta llega a un *frame* específico, que en este caso es **ActorManager**. Los *frames* seleccionados son en los que el modelo toca el suelo con la planta del pie.

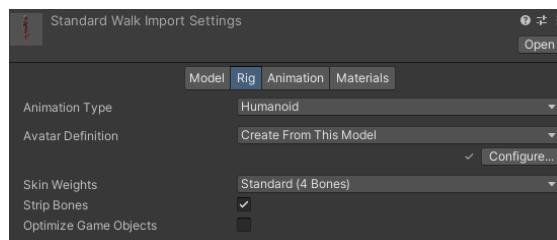


Figura 5.25.: Importación de la animación



Figura 5.26.: Importación de la animación

Cuando esto pase, el método **PlayStep** de la clase **ActorManager** reproducirá un sonido aleatorio de la lista de sonidos que almacena el actor. Esto se puede ver en el siguiente extracto de código:

```

1 public void PlayStep(){
2     AudioClip clip = soundSteps[Random.Range(0, soundSteps.Count)];
3     source.PlayOneShot(clip);
4 }
```

Tenemos que tener en cuenta que **source** es el componente  *AudioSource* que permite al actor reproducir sonidos como una fuente de audio.

### Animation Controller

Para poder controlar cuando el actor pasa de la animación de andar a la animación de estar parado, debemos crear un *Animator Controller*. Dentro del *Animator Controller* podemos tener distintos estados, esto *Unity* lo representa como nodos de un grafo.

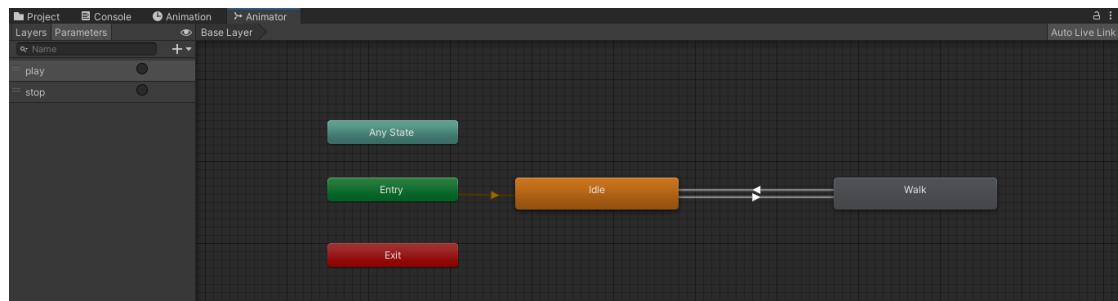


Figura 5.27.: Animator Controller del actor

Este controlador, llamado *ActorMove*, contiene dos nodos, uno con la animación de estar parado y otro con la animación de andar. Ambas animaciones serán reproducidas en bucle si el actor entra en alguno de los dos nodos. Cuando el actor se crea, comienza en el nodo de estar parado y transitará entre ambos nodos dependiendo de dos parámetros que podemos ver a la izquierda de la imagen. Estos parámetros son parámetros *trigger*, es decir, se activan cuando son llamados mediante código y sirven para que, cuando sean llamados, el actor pase de un estado al otro.

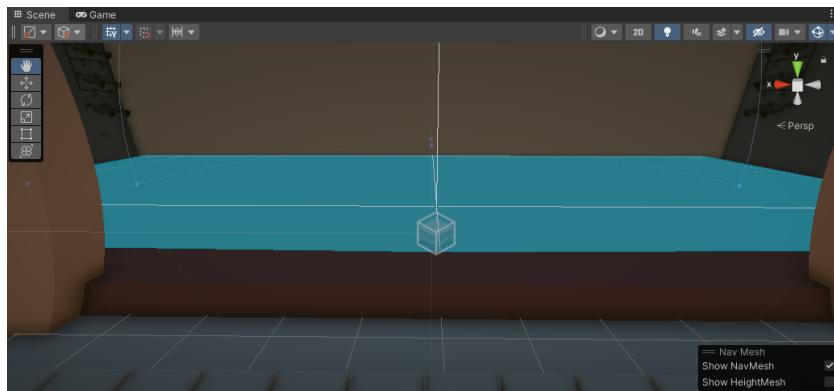
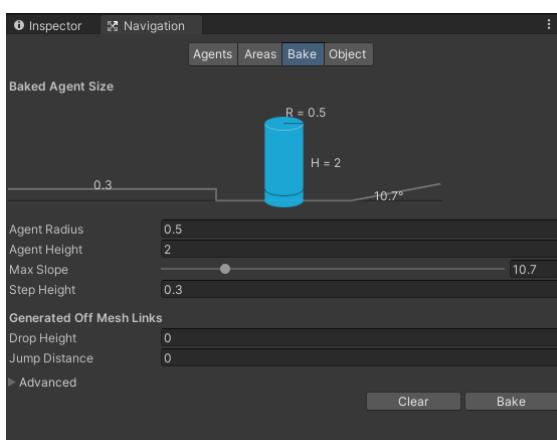
### Desplazamiento mediante NavMeshAgent

Como se comentó previamente, se usa un componente de *Unity* para realizar la trayectoria del actor. Este componente usa la Inteligencia Artificial para calcular la trayectoria más corta a un punto, volviendo a calcular el camino en el caso en el que el objeto encuentre un obstáculo. Como su nombre indica, el actor pasa a ser un agente de navegación. Para indicarle que no queremos que se desplace a un punto, podemos cambiar la variable que contiene el agente denominada *isStopped* por el valor *true*.

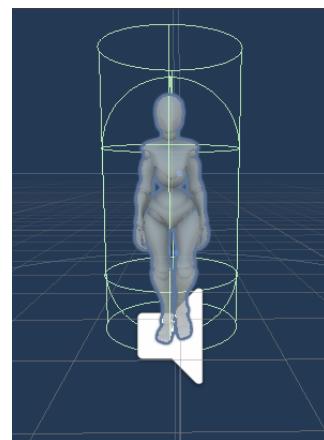
Para que el actor navegue por el mundo debemos indicarle por qué espacio puede navegar, denominándose esta zona *NavMesh*. Para crearlas, debemos hacerlo en la ventana de navegación del editor de *Unity*. Una vez creadas, se muestran en la ventana de *Scene* coloreadas de azul.

En el caso de este programa, el *NavMesh* corresponde con el área del escenario. Para evitar que los actores se choquen entre sí debemos añadirle un *Collider*, que delimita la zona en la que el actor sería considerado un obstáculo. En este caso, se usa un *Capsule collider*, que es un *collider* con forma de cápsula.

Cada vez que queramos que el actor se dirija a un punto, debemos indicarle el punto en la variable *destination*. Cuando el actor haya llegado a dicho punto se le indicará

Figura 5.28.: *NavMesh* del programa

(a) Ventana de navegación

(b) *Capsule Collider* del actorFigura 5.29.: *NavMesh* y *Capsule Collider*

la siguiente posición a la que acudir.

Para que el actor comience a andar, se encuentra el método `PlayAnim` en la clase `ActorManager`. Hay ocasiones en las que el actor no tiene trayectoria y siempre está parado en un punto, esto se puede comprobar si el vector de posiciones del actor es de tamaño uno. Si ocurriese este caso, el actor no realizaría ninguna trayectoria y permanecería parado.

```

1 public void PlayAnim()
2 {
3     // Si tiene mas de dos posiciones se empezaria por la primera
4     i_anim = 1;
5     if (i_anim < pos.Count)
6     {
7         // El agente deja de estar parado
8         agent.isStopped = false;
9         anim.SetTrigger("play");

```

```

10         agent.destination = pos[i_anim];
11     }
12 }
13 }
```

La variable `i_anim` indica el índice del vector de posiciones de la posición a la que debe ir el actor. Para que el actor realice el movimiento tenemos que poner al agente en movimiento e indicarle el destino. Además, activamos el *trigger play* para que comience la animación de andar.

En el método `Update` de la clase `ActorManager` se va a controlar el movimiento del actor. Recordamos que este método es llamado una vez por *frame*. Los momentos cruciales de la trayectoria son en los que el actor llega a uno de los destinos de ésta, es ahí cuando se tendrán que hacer los cambios correspondientes. Por lo que, se comprueba si el actor ha llegado a su destino. También se debe comprobar que el actor no esté parado, bien porque no tenga movimiento o bien porque ya ha llegado al destino final.

Cuando estas comprobaciones han sido realizadas, cambiamos la animación del actor a esta parado porque, aunque no haya terminado su trayectoria, simula un movimiento más natural el pararse en el punto de destino y después comenzar a andar. Además, puede darse el caso en el que el actor ha llegado al destino final, para ello el paso siguiente es comprobar que el siguiente índice de la animación sigue estando dentro de los posibles en el vector de posiciones. Para ello vemos que sea menor que el tamaño del vector. Si es así, se seleccionará un nuevo destino para el agente y comenzará la animación de andar. En caso contrario, el agente se parará, puesto que ya ha llegado al final de la trayectoria.

```

1 // Update is called once per frame
2 void Update(){
3     if ((transform.position.x == agent.destination.x) && (transform.position.z ==
4         agent.destination.z) && !agent.isStopped){
5         anim.SetTrigger("stop");
6         if (i_anim + 1 < pos.Count){
7             i_anim++;
8             agent.destination = pos[i_anim];
9             anim.SetTrigger("play");
10        }
11    else{
12        agent.isStopped = true;
13    }
14 }
```

En el caso en el que el actor vuelva a comenzar el movimiento desde el principio, hay que tener en cuenta ciertas cosas. El actor debe volver a la posición inicial, el actor debe tener la rotación inicial, el actor debe comenzar parado y si el actor estaba andando (ya que puede darse el caso en el que el usuario pulse el botón de *Replay* justo en medio de la animación) debe volver a la animación de estar parado, para ello, comprobamos en qué estado del *Animator Controller* se encuentra. Todo esto se encuentra en el método `Inicio` de la clase `ActorManager`.

```

1 public void Inicio(){
```

```
2     transform.position = pos[0];
3     transform.rotation = rot;
4     agent.isStopped = true;
5     if(anim.GetCurrentAnimatorStateInfo(0).IsName("Walk")){
6         anim.SetTrigger("stop");
7     }
8 }
```

Por lo tanto, para volver a reproducir el movimiento de los actores en escena, simplemente hay que llamar al método de `Inicio` y posteriormente al método de `PlayAnim`. Esto se realiza en la clase `Manager` en los métodos `PlayActors`, `InicioActors` y `Replay`. Estos métodos son llamados cuando el usuario pulsa el botón de *Play* o de *Replay* o cuando vuelve al menú inicial.

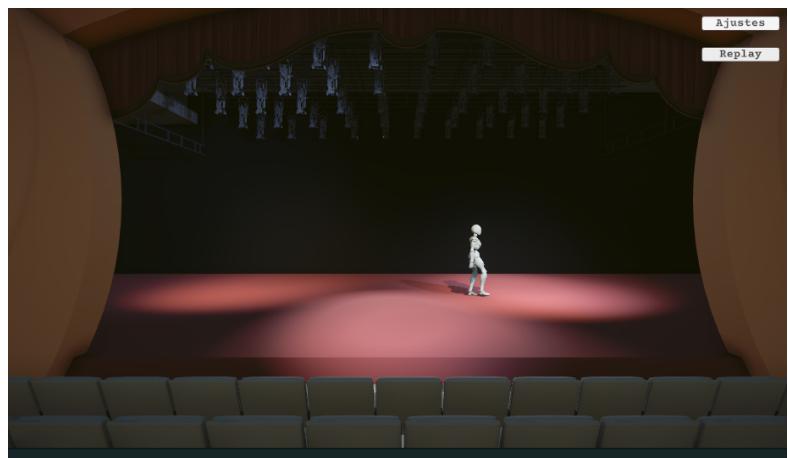
```
1 // para que los actores inicien el movimiento
2 public void PlayActors(){
3     foreach (GameObject actor in actorScene){
4         actor.GetComponent<ActorManager>().PlayAnim();
5     }
6 }
7
8 // los actores vuelven a la posicion inicial
9 public void InicioActors(){
10    foreach (GameObject actor in actorScene){
11        actor.GetComponent<ActorManager>().Inicio();
12    }
13 }
14
15 public void Replay(){
16     InicioActors();
17     PlayActors();
18 }
```

Se separan en dos funciones distintas ya que, cuando el usuario pulse el botón de Ajustes y vuelva al menú donde se configuran los parámetros, los actores deben volver a la posición inicial pero no deben iniciar el movimiento. Al igual que, cuando el usuario pulsa el botón de *Play*, los actores simplemente deben iniciar el movimiento ya que se encuentran en la posición inicial.

En las siguientes imágenes se puede observar la trayectoria del personaje según las posiciones elegidas en la Figura 5.24.

## 5.4. Manual de instalación

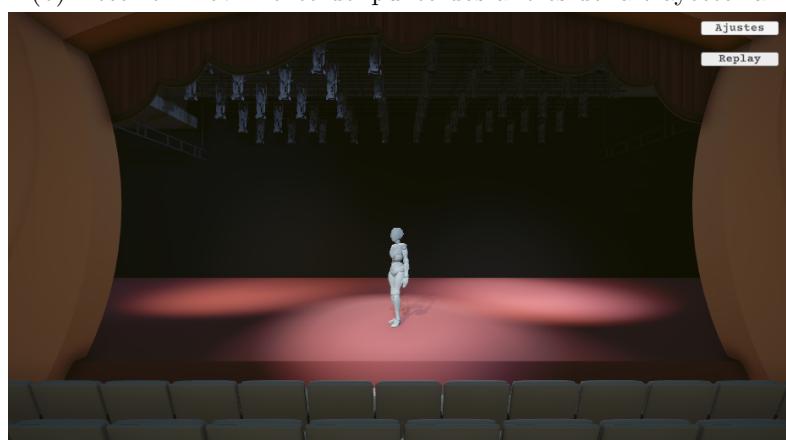
Para ejecutar la aplicación se debe descomprimir el *zip* con el nombre `Staging.exe` y ejecutar el archivo `TFG-Staging.x86_64`. Se debe recordar que sólo se podrá ejecutar en ordenadores con sistema operativo Linux.



(a) Actor en movimiento del punto uno al dos de la trayectoria.



(b) Actor en movimiento del punto dos al tres de la trayectoria.



(c) Fin de la trayectoria.

Figura 5.30.: Trayectoria de la Figura 5.24

## 6. Conclusiones y Vías Futuras

La llegada de las nuevas tecnologías puede permitir que el director teatral abandone las viejas técnicas de planificación de la puesta en escena de una obra para pasar a ser generador de nuevos contenidos, lo que exige una predisposición hacia una renovación en las formas de hacer. El empleo de una serie de instrumentos alternativos le facilitará, sin duda alguna, no solo la tarea de bosquejo y preparación, sino también de comunicación de lo que quiere o requiere su montaje a su equipo técnico y actoral. Se trata de un uso de programas que no solo están llegando sino que se están instalando en nuestras vidas. La irrupción de las nuevas tecnologías se convierte en el detonante que cataliza la ruptura con los modelos convencionales de hacer o trabajar. Y en este escenario es donde nuestra propuesta alcanza su funcionalidad.

No obstante, a partir de la elaboración del programa surgen diversas vías para el aumento de la eficacia o viabilidad. Unas de ellas está enfocada hacia la mejora de la funcionalidad ya existente y otras, en la creación de una ambientación para la escena.

En relación a la primera vía comentada, sería interesante añadir una mayor personalización en lo que se refiere a la iluminación, permitiendo que en las luces denominadas “normales” se pudiera configurar la intensidad de cada uno de los tres focos que la componen. Además, con respecto a la iluminación pero perteneciendo a la segunda vía, sería interesante incluir una luz de fondo en la que se pueda cambiar el color de ésta, aportando así diferentes matices a la escena, lo que proporcionaría distintas sensaciones en el espectador y podría incluso, llegar a representar la emoción de la escena.

Asimismo, una de las formas más comunes en el teatro y en el cine de crear una sensación específica en el espectador es mediante la música. Ésta tiene una intención clara subrayar un momento clave o bien, de transportar al público a una emoción concreta. Esto podría representarse en el programa mediante un *dropdown* con diversos sonidos o músicas ambiente que el usuario pudiera escoger para que se escuchase de fondo durante toda la escena. Para ello, simplemente habría que buscar sonidos o música con la licencia adecuada y añadirlos a la clase **Manager** y reproducirlos mediante un  *AudioSource*, que en este caso sería 2D, para que se escuchase por igual por toda la escena, ya que la distancia entre los altavoces del teatro y el sitio del espectador va a ser siempre el mismo.

Otra manera de personalizar, ampliar o diversificar la funcionalidad existente sería con el movimiento de los personajes. Podría personalizarse la velocidad de éstos aportando una urgencia distinta a cada uno de los actores. Para conseguir eso, deberían cambiarse las velocidades tanto de la animación como del movimiento en la trayectoria.

Por tanto, nuestro trabajo ha podido significar una pequeña contribución para la

mejora de cierta tarea de planificación del montaje de obras teatrales pero vemos que aún quedan muchas vías por explorar que creemos interesantes y pertinentes.

## **Parte II.**

# **Cuaterniones**

En esta parte se presentará la parte matemática del trabajo. En ésta se abordará la definición del álgebra sobre los reales conocida como los cuaterniones. También se incluirá un contexto histórico de éstos y la aplicación que tienen en las rotaciones para la informática gráfica. Además, se presentará una demostración del Teorema de Frobenius.



## 7. Introducción

En esta parte del trabajo se profundizará en las álgebras de división finito dimensionales sobre los reales, en especial, en los cuaterniones  $\mathbb{H}$ .

En el capítulo 8, se hablará de la historia de los cuaterniones. Siendo de relevante importancia la vida del descubridor de éstos, William Rowan Hamilton. Se hará un breve repaso de todos los aspectos de su vida, así como de la etapa en la que desarrolló su mayor obra, los cuaterniones. También se comentará la utilidad de los cuaterniones a lo largo de los años y su actual aplicación en la informática gráfica. Se mencionarán las álgebras de Grassmann así como la relación de éstas con los cuaterniones dada por Clifford.

En el siguiente capítulo 9 se hará una introducción a los cuaterniones definiendo los anillos de división, los módulos por la izquierda sobre un anillo y el álgebra de un anillo sobre un cuerpo. Posteriormente, a partir de los complejos, debido a la similitud con éstos, se definirán los cuaterniones, mostrando los vectores base, las propiedades de éstos y las operaciones de suma y de multiplicación. También se definirá la norma y el inverso de un cuaternion para finalmente comprobar que forman un álgebra de división sobre los reales.

El teorema de Frobenius, que caracteriza las álgebras de división finito dimensionales sobre los reales, será enunciado y demostrado en el capítulo 10.

Finalmente, en el capítulo 11, se explicará el uso de los cuaterniones en las rotaciones espaciales y también se comentará el uso de éstos en *Unity*.

La principal fuente bibliográfica para la elaboración de este trabajo ha sido la tesis de Johannes C. Familton titulada *Quaternions: A History of Complex Noncommutative Rotation Groups in Theoretical Physics* [15]. Siendo de relevante importancia para la historia algunos enlaces de la página *mathshistory* [16, 17, 18] y un capítulo del libro de *Historia de la matemática* [19].

Para el teorema de Frobenius, la demostración ha sido sacada de Wikipedia [20], ampliándola con diversas demostraciones. Las pequeñas referencias a *Unity* provienen de la documentación del programa [3].

Finalmente, para las rotaciones mediante cuaterniones ha sido crucial el artículo [1] de G.F. Torres del Castillo.



## 8. Historia de los cuaterniones

Para hablar de la historia de los cuaterniones, debemos hablar de su descubridor: William Rowan Hamilton (1805-1865).

William Rowan Hamilton nació en Dublín el 4 de agosto de 1805. Su padre era abogado en ejercicio y de su madre se dice que heredó las altas cualidades intelectuales. Sin embargo, su educación estaba en manos de su tío Rev James Hamilton, lingüista.

Con sólo cinco años ya sabía griego, hebreo y latín, y a los diez una media docena de lenguas orientales pero no fue hasta que conoció a Zerah Colburn, prodigo en el cálculo numérico, que su interés por las matemáticas no despertó.

Durante su adolescencia estudió el Álgebra de Clairaut y empezó a estudiar los trabajos de Newton y Laplace, encontrando un error en la *Mecánica Celeste* de Laplace lo que hizo que John Brinkley, Astrónomo Real de Irlanda, se fijase en él.

Comenzó sus estudios en el Trinity College con 18 años, obteniendo un *Optime* en su primer año, distinción raramente dada. Durante su periodo en Trinity conoció a la persona que más influiría en su vida personal, la que lo llevaría a diversos períodos de alcoholismo y depresión: Catherine. Se podría decir que Catherine fue el gran amor frustrado de su vida.

En 1824 presenta su primer artículo a la Real Academia Irlandesa, titulado *On Causistics*. Meses más tarde, el compromiso de Catherine con otro hombre lo lleva a un periodo oscuro en el cual comienza una de las actividades que lo acompañará para el resto de su vida, la poesía.

Antes de graduarse, presenta un trabajo titulado *Theory of Systems of Rays*, donde introduce la función característica aplicada a óptica, y es nombrado Astrónomo Real de Irlanda, director del Observatorio de Dunsink y profesor de astronomía. Esto fue motivo de controversia debido a que Hamilton no tenía experiencia en este ámbito.

Hamilton continúa trabajando en su *Theory of Systems of Rays* durante un largo periodo de tiempo. Tras publicar el tercer suplemento del trabajo en 1832, le pide al profesor de Física del Trinity College, Humphrey Lloyd, que compruebe experimentalmente la predicción que hacía en el trabajo de la refracción cónica en ciertos cristales. Esta comprobación satisfactoria le proporcionó fama y le aseguró su reputación.

Durante esos años se casa con la que será su mujer y madre de sus tres hijos: Helen Maria Bayly. Sin embargo, su matrimonio estará marcado por un continuo distanciamiento.

En 1833, presenta un artículo a la Real Academia Irlandesa en el que introducía y

estudiaba un álgebra formal de parejas de números reales con las reglas de combinación que existen hoy para los números complejos. Hamilton interpreta la multiplicación de estas parejas dada por

$$(a, b) * (\alpha, \beta) = (a\alpha - b\beta, a\beta + b\alpha)$$

como una operación en la que interviene una rotación. Con esto se puede ver la versión definitiva del concepto de número complejo como un par ordenado de números reales (idea implícita en la representación gráfica tanto de Wessel como de Argand y Gauss) que se hacía explícita por primera vez.

En 1835, publica *Algebra as the Science of Pure Time*, donde identifica a las parejas algebraicas como pasos en el tiempo. Ese mismo año fue nombrado caballero. Tras este descubrimiento, intentó extender la idea a tres dimensiones, es decir, a las ternas  $a+bi+cj$  donde  $i^2 = j^2 = -1$ . A pesar de que la suma no le producía ninguna dificultad, la multiplicación de n-uplas para un n mayor a dos lo tuvo obsesionado y atormentado durante unos diez años. El problema residía en que debía decidir cuál sería el producto de  $ij$ . Él sabía que la fórmula sería inconsistente a menos que satisfaciera lo que él llamaba la “ley del módulo”, es decir, que si la multiplicación de dos ternas daba lugar a una tercera  $(a+bi+cj)(d+ei+fj) = u+vi+wj$ , entonces la “ley del módulo” exigiría que  $(a^2 + b^2 + c^2)(d^2 + e^2 + f^2) = u^2 + v^2 + w^2$ . Probó con muchas combinaciones sin embargo no pudo encontrar ninguna que funcionase.

Cada mañana, en el desayuno, uno de sus hijos le preguntaba si había conseguido multiplicar las ternas, lo que a Hamilton respondía que sólo podía sumarlas y restarlas. Esta dificultad residía en que no podía encontrar una identidad cúbica, debido a que dicha identidad no existía.

No fue hasta el 16 de octubre de 1843, dando un paseo en el Royal Canal con su mujer, cuando le llegó la inspiración. Debía usar cuádruplas en lugar de ternas y debía desprenderse de la idea de la commutatividad en la multiplicación. No pudo resistirse a grabar en el puente de Brougham la fórmula de los cuaterniones.

$$i^2 = j^2 = k^2 = ijk = -1.$$

Esta idea pudo llegar a su mente debido a la existencia de la identidad de los cuatro cuadrados descubierta por Euler en 1748. Esta identidad afirma que

$$(a_1^2 + a_2^2 + a_3^2 + a_4^2)(b_1^2 + b_2^2 + b_3^2 + b_4^2) = (a_1b_1 - a_2b_2 - a_3b_3 - a_4b_4)^2 + (a_1b_2 + a_2b_1 + a_3b_4 - a_4b_3)^2 + (a_1b_3 - a_2b_4 + a_3b_1 + a_4b_2)^2 + (a_1b_4 + a_2b_3 - a_3b_2 + a_4b_1)^2.$$

Para poder preservar la distancia y el valor absoluto, Hamilton tuvo que abandonar la commutatividad.

Ese mismo día Hamilton pidió permiso a la Real Academia Irlandesa para leer un comunicado sobre los cuaterniones en la siguiente sesión.

Hamilton sintió que los cuaterniones revolucionarían la física matemática y pasó el

resto de su vida trabajando en ellos, considerando el descubrimiento de los cuaterniones como su obra más importante.

Poco después del descubrimiento de los cuaterniones, Hamilton conectó el álgebra de los cuaterniones con las rotaciones espaciales. Aunque, esta relación fue descubierta previamente por Olindo Rodrigues.

Los siguientes años de su vida estarían marcados también por el alcoholismo y la depresión derivados de diversos problemas personales. Publicó su libro *Lectures on Quaternions* en 1853. Sin embargo, no era un buen libro para aprender sobre cuaterniones y Hamilton era consciente de ello, por lo que decidió trabajar en un nuevo libro cuya calidad durase en el tiempo: *Elements of Quaternions*.

Al principio pensó que esta tarea sólo le llevaría dos años. Sin embargo, necesitó siete años y dejó el último capítulo sin acabar debido a que sufrió un ataque de gota que le causó la muerte.

El libro fue publicado en 1866 por su hijo mayor, William Edwin Hamilton. Al principio del libro indica que no ha cambiado nada de éste salvo algunos errores tipográficos, ya que no poseía ninguna instrucción porque se suponía que el libro sería publicado por el padre. También cuenta que su padre hablaba de la aplicación de los cuaterniones a la electricidad y a todas las cuestiones en la que la polaridad se daba, pero que no pudo desarrollarlas.

Poco antes de su muerte, la Academia Nacional de Ciencias de Estados Unidos nombró a Hamilton su primer miembro extranjero asociado.

Hamilton estaba tan satisfecho con el resultado de su sistema de cuaterniones que fundó una escuela dedicada a su estudio a la que llamó “*quaternionists*”.

Parece ser que Hamilton no llegó a encontrar la relación entre los cuaterniones y los vectores. Esto sería aclarado más tarde por Gibbs y Heaviside.

Al principio los cuaterniones se enseñaban por las universidades de todo el mundo, sin embargo, a mediados de la década de 1880 fueron reemplazados por el análisis vectorial que Gibbs y Heaviside desarrollaron. Temas que se habían escrito en término de los cuaterniones ahora usaban vectores. Esto es debido a que el análisis vectorial es conceptualmente más sencillo y la notación es más clara que la de los cuaterniones. Para la física del siglo XIX era suficiente, pero con la llegada de temas como la mecánica cuántica las limitaciones del cálculo vectorial se hicieron más evidentes y los cuaterniones fueron redescubiertos en forma de matrices de Pauli.

A día de hoy es más complicado sentirse cómodo con los cuaterniones ya que pensamos en términos del análisis vectorial aprendido en el colegio. Así, hacia finales del siglo XIX los cuaterniones pasaron a un “segundo plano” frente a otros métodos para muchas aplicaciones clásicas.

Una de las razones por las que reaparecieron los cuaterniones a finales del siglo XX fue por la animación por ordenador y otras aplicaciones que necesitaban programación. Esto es debido a que los cuaterniones usan el álgebra para describir las rotaciones espaciales,

lo que hace que sean más “compactos” que las matrices, por lo que son más rápidos computarlos que las matrices, lo que los hace más útiles. Para los cuaterniones sólo se necesitan cuatro escalares para almacenarlos, mientras que las matrices que se suelen usar para este tipo de programas necesitan nueve. Esto también aumenta la rapidez de la multiplicación de cuaterniones.

Mirando con retrospectiva queda claro que lo fundamental no fue la importancia de este álgebra si no el descubrimiento de la libertad de la matemática para construir álgebras que no necesitan satisfacer las restricciones de las «leyes fundamentales».

Cabe destacar la figura de otro matemático: Hermann Günter Grassmann (1809-1877). Grassmann fue un matemático autodidacta de los pocos que aplicó con éxito sus descubrimientos matemáticos a los problemas de la física. Grassmann nunca recibió formación matemática, de hecho estudió en la Universidad de Berlín lingüística y teología. Trabajaba de profesor de escuela e investigaba en su tiempo libre, siendo profesor asistente en el *Gymnasium* de su ciudad natal, Stettin.

Mientras que Hamilton desarrollaba los cuaterniones, Grassmann trabajaba en un enfoque más algebraico para cuestiones similares. Grassmann desarrolló un sistema que ponía la geometría en forma algebraica. Este enfoque algebraico, a diferencia de los cuaterniones, no está limitado por un espacio tridimensional. Para la época, ésta era una idea muy inusual ya que la mayoría de matemáticos y físicos estaban limitados por un espacio de dos o tres dimensiones. A su sistema incorporó la multiplicación no conmutativa, idea muy innovadora para su tiempo. Sin embargo, uno de los problemas para entender su trabajo fue la terminología que usó, muy distinta a los estándares de la época.

Grassmann usaba ideas que forman parte del análisis vectorial moderno antes de que Hamilton desarrollara los cuaterniones. También desarrolló un producto multidimensional análogo al producto escalar que llamó producto interno y a lo que hoy se llama producto vectorial lo llamó producto externo. En su trabajo, Grassmann incluyó temas que hoy se consideran parte del análisis vectorial.

Una vez pasada la barrera de la terminología, resultaba que, en general, sus álgebras eran más sencillas de utilizar para las aplicaciones que los cuaterniones de Hamilton. Por ejemplo, un vector está asociado usualmente a un punto o a una recta desde el origen al punto, lo que Grassmann hizo fue discutir situaciones en las que las rectas no pasaban necesariamente por el origen, permitiendo una mayor generalidad. Esto permitía que pudiesen ser “desplazadas”.

Grassmann empezó a aplicar sus ideas en su *Theorie der Ebbe und Flut: Prüfungsarbeit* en 1840 y posteriormente, en 1844 publicó *Die Lineale Ausdehnungslehre* que contiene el análisis vectorial todavía en uso a día de hoy. En 1861, publicó una segunda versión de éste llamada *Die Lineale Ausdehnungslehre, ein neuer Zweig der Mathematik*, intentando buscar atraer la atención de un mayor número de matemáticos y científicos, sin embargo, esto no ocurrió.

Fue otro matemático, William Kingdon Clifford (1845-1879), quien combinó los cu-

terniones con las álgebras de Grassmann. En 1878, publicó *Applications of Grassmann's Extensive Algebra*. En este trabajo, Clifford busca una forma más simple y general de ver las álgebras en dimensiones superiores. Se dio cuenta que las álgebras de Grassmann y los cuaterniones no estaban realmente en conflicto entre sí como se pensaba. Así, con un pequeño ajuste, Clifford fue capaz de resolver los aparentes problemas que existían entre ambos.

Lo que hizo Clifford fue relacionar las álgebras de Grassmann con los cuaterniones destacando primero sus diferencias. Clifford deja claro que su trabajo es una forma de conectar las ideas de Hamilton y Grassmann. No tenía la intención de cambiar los cuaterniones, simplemente quería dar una presentación más completa y sencilla de las ideas que tanto Hamilton y Grassmann presentaron en su trabajo, haciendo que las ideas de Hamilton fuesen más fáciles de usar para los físicos. Esto lo hizo creando un método que pudiese ser usado de manera más sencilla en los cálculos sin tener que abandonar las partes conceptualmente interesantes tanto de los cuaterniones como de las álgebras de Grassmann. Las álgebras geométricas, a diferencia de los vectores, no son simplemente un “caso especial” de los cuaterniones, pero incorporan la naturaleza conceptual profunda de los cuaterniones que se perdió con los vectores.

En 1878, se podría decir que Clifford reinventó y generalizó los cuaterniones de Hamilton.



## 9. Cuaterniones

Para poder definir los cuaterniones es necesario recordar algunas definiciones previas.

Un anillo  $(A, +, \cdot)$  es un conjunto con dos operaciones binarias  $A \times A \rightarrow A$  denotadas por suma (+) y producto ( $\cdot$ ) que verifican los axiomas de asociatividad de la suma (para todo  $a, b, c \in A$ , se tiene que  $a + (b + c) = (a + b) + c$ ), la existencia de un elemento neutro para la suma, conocido como 0 (existe  $0 \in A$  tal que, para todo  $a \in A$ , se cumple que  $0 + a = a = a + 0$ ), la existencia de un opuesto (para todo  $a \in A$  existe su opuesto denotado por  $-a \in A$ , tal que  $a + (-a) = 0 = (-a) + a$ ) y la comutatividad de la suma (para todo  $a, b \in A$ , se verifica que  $a + b = b + a$ ). Estos primeros cuatro axiomas se resumen en uno:  $(A, +)$  es un grupo abeliano. El producto también debe verificar los axiomas de asociatividad del producto (para todo  $a, b, c \in A$ , se cumple que  $a(bc) = (ab)c$ ), la propiedad distributiva (para todo  $a, b, c \in A$ , se tiene que  $a(b + c) = ab + ac$  y que  $(b + c)a = ba + ca$ ) y la existencia de un elemento neutro para el producto, denotado por 1 (existe  $1 \in A$ , tal que para todo  $a \in A$ , se verifica que  $1a = a = a1$ ). Un anillo se llama comunitativo si verifica el axioma de la comutatividad para el producto (para todo  $a, b \in A$ , se cumple que  $ab = ba$ ). Un anillo de división es un anillo que verifica el axioma adicional de la existencia de inverso (para todo elemento no nulo  $a \in A$ , existe un inverso denotado por  $\exists a^{-1} \in A$ , donde  $aa^{-1} = 1 = a^{-1}a$ ). Un cuerpo es un anillo de división comunitativo.

Sea  $A$  un anillo. Un módulo por la izquierda sobre  $A$ , o  $A$ -módulo,  $(M, +, \cdot)$  es un conjunto  $M$  junto con una ley de composición interna  $M \times M \rightarrow M$  dada por  $(x, y) \mapsto x + y$  y una ley de composición externa  $A \times M \rightarrow M$  denotada  $(a, x) \mapsto ax$  que verifica que  $(M, +)$  es un grupo abeliano. Además cumple la distributividad respecto a escalares (para todo  $a, b \in A$  y para todo  $x \in M$  se verifica que  $(a + b)x = ax + bx$ ), la distributividad respecto a vectores (para todo  $a \in A$  y para todo  $x, y \in M$ , se tiene que  $a(x + y) = ax + ay$ ), la pseudoasociatividad (para todo  $a, b \in A$  y para todo  $x \in M$ , se cumple que  $a(bx) = (ab)x$ ) y la acción trivial del uno (para todo  $x \in M$ , se cumple que  $1x = x$ ). A los elementos de  $M$  se les llama vectores y a los de  $A$  escalares. En el caso particular en que  $A$  es un cuerpo a  $M$  se le llama espacio vectorial sobre  $A$ . Una base  $\mathcal{B}$  de un espacio vectorial es un subconjunto de  $M$  que cumple que los elementos de dicha base son linealmente independientes, es decir, ninguno de ellos es combinación lineal del resto y, además, forman un sistema generador del espacio vectorial, lo que equivale a decir que todo elemento del espacio puede ser descrito de forma única como combinación lineal de los elementos de la base. La dimensión de un espacio vectorial estará conformada por el número de vectores que forman una base de dicho espacio.

Sea  $K$  un anillo comunitativo. Un álgebra sobre  $K$  es un conjunto  $A$  junto con dos

leyes de composición internas  $A \times A \rightarrow A$  denotadas por  $a + b$  y  $ab$  y una ley de composición externa  $K \times A \rightarrow A$  denotada por  $\lambda * a$  que verifican que  $(A, +, *)$  es un  $K$ -módulo,  $(A, +, \cdot)$  es un anillo y cumple la pseudoasociatividad (para todo  $\lambda \in K$  y para todos  $a, b \in A$ , se cumple que  $(\lambda * (ab)) = ((\lambda * a)b) = a(\lambda * b)$ ).

Los números complejos ( $\mathbb{C}$ ) se pueden ver como un espacio vectorial real de dos dimensiones. Los vectores base de este espacio vectorial son  $\{1, i\}$ . Los números complejos se suman componente a componente:

$$(a + bi) + (c + di) = (a + c) + (b + d)i.$$

La multiplicación está determinada por la ley distributiva y por la propiedad  $i^2 = -1$ :

$$(a + bi)(c + di) = ac + adi + bci + bdi^2 = (ac - bd) + (ad + bc)i.$$

Los números complejos con estas operaciones forman un álgebra sobre los reales.

Los cuaterniones,  $\mathbb{H}$ , son una extensión de este sistema de números complejos. Son llamados así porque tienen cuatro vectores como base  $\{1, i, j, k\}$ , donde  $i, j$  y  $k$  son las componentes imaginarias, que son tres raíces diferentes de  $-1$ , es decir,  $i^2 = j^2 = k^2 = -1$ . Los cuaterniones se pueden escribir como  $q_0 + q_1i + q_2j + q_3k$  o como  $(q_0, q_1, q_2, q_3)$  donde  $q_0, q_1, q_2, q_3$  son números reales.

La suma de los cuaterniones está definida componente a componente y es asociativa y commutativa, ya que los coeficientes son números reales cuya suma cumple esta propiedad. Sean

$$p = (p_0 + p_1i + p_2j + p_3k), \quad q = (q_0 + q_1i + q_2j + q_3k)$$

dos cuaterniones,

$$\begin{aligned} p + q &= (p_0 + p_1i + p_2j + p_3k) + (q_0 + q_1i + q_2j + q_3k) \\ &= (p_0 + q_0) + (p_1 + q_1)i + (p_2 + q_2)j + (p_3 + q_3)k. \end{aligned}$$

El cero en la suma es el cuaternion donde  $q_0 = q_1 = q_2 = q_3 = 0$  y el opuesto de un cuaternion está dado por el opuesto de cada una de sus componentes, es decir, el opuesto de  $p = (p_0 + p_1i + p_2j + p_3k)$  sería  $-p = (-p_0 - p_1i - p_2j - p_3k)$ .

Para la multiplicación debemos tener en cuenta las siguientes propiedades de las unidades imaginarias:

$$\begin{aligned} i^2 &= j^2 = k^2 = -1, \\ ij &= -ji = k, \\ jk &= -kj = i, \\ ki &= -ik = j. \end{aligned}$$

La tabla de multiplicación de Cayley para los cuaterniones sería:

.	<b>1</b>	<b>-1</b>	<b>i</b>	<b>-i</b>	<b>j</b>	<b>-j</b>	<b>k</b>	<b>-k</b>
<b>1</b>	1	-1	i	-i	j	-j	k	-k
<b>i</b>	i	-i	-1	1	k	-k	-j	j
<b>-i</b>	-i	i	1	-1	-k	k	j	-j
<b>j</b>	j	-j	-k	k	-1	1	i	-i
<b>-j</b>	-j	j	k	-k	1	-1	-i	i
<b>k</b>	k	-k	j	-j	-i	i	-1	1
<b>-k</b>	k	-k	j	-j	-i	i	-1	1

Tabla 9.1.: Tabla de Cayley

Evidentemente, por las propiedades previas, la multiplicación de los cuaterniones no es conmutativa. Para dos cuaterniones  $p, q$  definidos previamente tenemos que

$$\begin{aligned} pq &= (p_0 + p_1i + p_2j + p_3k)(q_0 + q_1i + q_2j + q_3k) \\ &= p_0q_0 + p_0q_1i + p_0q_2j + p_0q_3k + p_1q_0i + p_1q_1i^2 + p_1q_2ij + p_1q_3ik \\ &\quad + p_2q_0j + p_2q_1ji + p_2q_2j^2 + p_2q_3jk + p_3q_0k + p_3q_1ki + p_3q_2kj + p_3q_3k^2, \end{aligned}$$

por las propiedades mencionadas tendríamos que

$$\begin{aligned} pq &= [\dots] = p_0q_0 + p_0q_1i + p_0q_2j + p_0q_3k + p_1q_0i - p_1q_1 + p_1q_2k - p_1q_3j \\ &\quad + p_2q_0j - p_2q_1k - p_2q_2 + p_2q_3i + p_3q_0k + p_3q_1j - p_3q_2i - p_3q_3 \\ &= p_0q_0 - p_1q_1 - p_2q_2 - p_3q_3 + (p_0q_1 + p_1q_0 + p_2q_3 - p_3q_2)i \\ &\quad + (p_0q_2 - p_1q_3 + p_2q_0 + p_3q_1)j + (p_0q_3 + p_1q_2 - p_2q_1 + p_3q_0)k. \end{aligned}$$

La unidad en la multiplicación sería la misma que la de los reales, es decir, el cuaternión donde la parte real es 1 y los coeficientes que acompañan a las unidades imaginarias son 0, esto es,  $1 = 1 + 0i + 0j + 0k$ .

De la misma manera que en los números complejos, en los cuaterniones también se define el conjugado. Recordemos que el conjugado de un número complejo  $z = a + bi$  se define como  $\bar{z} = a - bi$ . El conjugado de un cuaternión  $q = q_0 + q_1i + q_2j + q_3k$  se define de manera análoga  $\bar{q} = q_0 - q_1i - q_2j - q_3k$ . Como en el caso de los números complejos, se tiene que  $\bar{\bar{q}} = q$ ,  $q\bar{q} = \bar{q}q = q_0^2 + q_1^2 + q_2^2 + q_3^2$ ,  $\overline{(p+q)} = \bar{p} + \bar{q}$  y  $\overline{(pq)} = \bar{q}\bar{p}$ . Cabe destacar que, en el caso de la multiplicación de conjugados, la multiplicación es conmutativa.

Tomando la norma de un cuaternión como

$$|q| = \sqrt{q\bar{q}} = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2} = |\bar{q}|.$$

Además,

$$|pq| = \sqrt{pq(\bar{pq})} = \sqrt{pq\bar{q}\bar{p}} = \sqrt{p|q|^2\bar{p}} = \sqrt{p\bar{p}|q|^2} = \sqrt{|p|^2|q|^2} = |p||q|.$$

Esto nos permite calcular el inverso de un cuaternión no nulo como

$$q^{-1} = \frac{q_0 - q_1 i - q_2 j - q_3 k}{q_0^2 + q_1^2 + q_2^2 + q_3^2} = \frac{\bar{q}}{|q|^2},$$

$$qq^{-1} = \frac{q\bar{q}}{|q|^2} = \frac{\bar{q}q}{|q|^2} = q^{-1}q = \frac{|q|^2}{|q|^2} = 1.$$

El producto también cumple la propiedad asociativa y distributiva, por lo que  $(\mathbb{H}, +, \cdot)$  es un anillo de división no commutativo. Tenemos que tener en cuenta que los reales se embeben de forma natural en los cuaterniones, por lo que la ley de composición externa  $\mathbb{R} \times \mathbb{H} \rightarrow \mathbb{H}$   $(\lambda, q) \mapsto \lambda q$  forma parte de la multiplicación definida previamente. Esta operación cumple la pseudoasociatividad. Sea  $\lambda = \lambda_0 + 0i + 0j + 0k$  y  $p, q$  los cuaterniones definidos previamente tenemos

$pq = p_0q_0 - p_1q_1 - p_2q_2 - p_3q_3 + (p_0q_1 + p_1q_0 + p_2q_3 - p_3q_2)i + (p_0q_2 - p_1q_3 + p_2q_0 + p_3q_1)j + (p_0q_3 + p_1q_2 - p_2q_1 + p_3q_0)k$  por lo que

$$\lambda(pq) = \lambda_0(p_0q_0 - p_1q_1 - p_2q_2 - p_3q_3) + \lambda_0(p_0q_1 + p_1q_0 + p_2q_3 - p_3q_2)i + \lambda_0(p_0q_2 - p_1q_3 + p_2q_0 + p_3q_1)j + \lambda_0(p_0q_3 + p_1q_2 - p_2q_1 + p_3q_0)k.$$

Como  $\lambda q = \lambda_0q_0 + \lambda_0q_1i + \lambda_0q_2j + \lambda_0q_3k$ , sacando factor común en cada suma se tiene que  $\lambda(pq) = p\lambda q$ . Por tanto, los cuaterniones forman un álgebra sobre los reales.

# 10. Teorema de Frobenius

El teorema de Frobenius, demostrado por Ferdinand Georg Frobenius en 1877, caracteriza las álgebras asociativas de división de dimensión finita sobre los números reales.

**Teorema 10.1.** *Toda álgebra asociativa de división de dimensión finita sobre los reales es isomorfa a una de las siguientes: los números reales ( $\mathbb{R}$ ), los números complejos ( $\mathbb{C}$ ) o los cuaterniones ( $\mathbb{H}$ ).*

Antes de comenzar la demostración vamos a introducir la notación a usar.

Sea  $\mathcal{D}$  el álgebra de división. Sea  $n$  la dimensión de  $\mathcal{D}$ , que suponemos finita como espacio vectorial.

Identificaremos los múltiplos reales de 1 con  $\mathbb{R}$ . Cuando se escribe  $a \leq 0$  para un elemento  $a$  en  $\mathcal{D}$ , asumimos que  $a$  está contenido en  $\mathbb{R}$ .

Cualquier elemento  $d$  de  $\mathcal{D}$  define un endomorfismo de  $\mathcal{D}$  con la multiplicación por la izquierda, identificamos  $d$  con ese endomorfismo. Por lo tanto, podemos hablar de la traza de  $d$ , y de su polinomio característico y mínimo. Si recordamos, la traza de  $d$  corresponde con la traza de la matriz asociada al endomorfismo en una base cualquiera, el polinomio característico corresponde con el polinomio dado por  $p(\lambda) = \det(A - \lambda \text{Id})$  donde  $A$  corresponde con la matriz del endomorfismo y el polinomio mínimo corresponde con el polinomio mónico  $p(x)$ , es decir, el polinomio cuyo coeficiente líder es uno, de menor grado que cumple que  $p(A) = 0$ .

Para cada  $z$  en  $\mathbb{C}$  se define el siguiente polinomio cuadrático real:

$$Q(z; x) = x^2 - 2\operatorname{Re}(z)x + |z|^2 = (x - z)(x - \bar{z}) \in \mathbb{R}[x].$$

Nótese que si  $z \in \mathbb{C} \setminus \mathbb{R}$ , entonces  $Q(z; x)$  es irreducible sobre  $\mathbb{R}$ .

Para la demostración del Teorema de Frobenius van a ser clave dos teoremas: el Teorema de Cayley Hamilton y el Teorema de Rango Nulidad.

**Teorema 10.2. Teorema de Cayley-Hamilton.** *Sea  $A$  una matriz cuadrada de orden  $n$  y  $p_A$  su polinomio característico. Entonces se verifica que  $p_A(A) = 0_n$ , donde  $0_n$  es la matriz nula de orden  $n$ . Es decir, cada matriz cuadrada es solución de su ecuación característica.*

*Demostración.* Si  $A$  es una matriz cuadrada entonces:

$$A \cdot \operatorname{Adj}(A)^T = \det(A) \cdot \text{Id}_n,$$

donde  $\text{Adj}(A)$  es la matriz adjunta de  $A$ , es decir,

$$\text{Adj}(A) = (\alpha_{ij}) \quad 1 \leq i, j \leq n, \quad (\alpha_{ij}) = (-1)^{i+j} \cdot \det(A_{ij}),$$

donde  $A_{ij}$  es la matriz de orden  $n - 1$  que se obtiene de  $A$  al eliminar la fila  $i$ -ésima y la columna  $j$ -ésima.

Tomando ahora la matriz  $A - \lambda \text{Id}_n$  tendríamos que

$$(A - \lambda \text{Id}_n) \cdot \text{Adj}(A - \lambda \text{Id}_n)^T = \det(A - \lambda \text{Id}_n) \cdot \text{Id}_n = p_A(\lambda) \text{Id}_n.$$

$$\text{Adj}(A - \lambda \text{Id}_n)^T = B_{n-1}\lambda^{n-1} + B_{n-2}\lambda^{n-2} + \cdots + B_1\lambda + B_0,$$

donde  $B_i$  son matrices cuadradas de orden  $n$  para  $i = 0, \dots, n - 1$ . Esto es así porque cada elemento de la matriz adjunta está formado por los determinantes de matrices de orden  $n - 1$  que son polinomios de orden menor o igual que  $n - 1$ .

$$\begin{aligned} & (A - \lambda \text{Id}_n)(B_{n-1}\lambda^{n-1} + B_{n-2}\lambda^{n-2} + \cdots + B_1\lambda + B_0) \\ &= (-1)^n \text{Id}_n \lambda^n + c_{n-1} \text{Id}_{n-1} \lambda^{n-1} + \cdots + c_1 \text{Id}_1 \lambda + c_0 \lambda = p_A(\lambda) \text{Id}_n, \end{aligned}$$

donde  $c_i \in \mathbb{R}$ ,  $i = 0, \dots, n - 1$ . Si identificamos los coeficientes según los distintos monomios de  $\lambda^i$ ,  $i = 0, \dots, n$  y sustituyendo  $\lambda$  por  $A$  tendríamos:

$$\cancel{-A^n B_{n-1}} = (-1)^n \text{Id}_n A^n,$$

$$\cancel{A^n B_{n-1}} - \cancel{A^{n-1} B_{n-2}} = c_{n-1} A^{n-1},$$

$$\cancel{A^{n-1} B_{n-2}} - \cancel{A^{n-2} B_{n-3}} = c_{n-2} A^{n-2},$$

⋮

$$\cancel{A^2 B_1} - \cancel{A B_0} = c_1 A,$$

$$\cancel{A B_0} = c_0 \text{Id}_n.$$

Al sumar, vemos que se van cancelando los términos con los anteriores, obteniendo  $0_n = p_A(A)$ .  $\square$

**Teorema 10.3. Teorema de Rango Nulidad.** Sea  $f : V \rightarrow V'$  una aplicación lineal tal que  $V$  y  $V'$  son de dimensión finita. Entonces,

$$\dim(V) = \dim(\text{Ker}(f)) + \dim(\text{Im}(f)).$$

*Demostración.* Denotemos  $\dim(V') = m$ . Tomamos la matriz asociada a la aplicación lineal  $f$  respecto de ciertas bases  $B$  y  $B'$  de  $V$  y  $V'$  respectivamente y denotemos  $r$  al rango de  $A$ . Entonces, las columnas de  $A$  son las coordenadas respecto de  $B'$  de un sistema de generadores de  $\text{Im}(f)$ , y en particular  $\dim(\text{Im}(f)) = r$ .

Por otra parte, un vector  $x \in V$  de coordenadas  $x = (x_1, \dots, x_n)_B$  está en el núcleo de  $f$  si, y sólo si,  $f(x) = 0$  o equivalentemente si, y sólo si,  $Ax = 0$ , por lo tanto, se obtienen las coordenadas cartesianas de  $\text{Ker}(f)$  a partir del sistema homogéneo cuya matriz de coeficientes es A:

$$\begin{cases} a_{11}x_1 + \cdots + a_{1n}x_n = 0, \\ a_{21}x_1 + \cdots + a_{2n}x_n = 0, \\ \vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n = 0. \end{cases}$$

De estas  $m$  ecuaciones, el número de ecuaciones independientes es igual al rango de la matriz, es decir,  $r$ . Por lo tanto,  $\dim(\text{Ker}(f)) = n - r$ . Si sumamos  $\dim(\text{Ker}(f)) + \dim(\text{Im}(f)) = n - r + r = n$ , que es lo queríamos demostrar.  $\square$

Además de los dos teoremas previos se van a necesitar los dos siguientes lemas.

**Lema 10.1.** *Sea A una matriz cuadrada de orden m y  $p_A(\lambda)$  su polinomio característico. Entonces se verifica que*

$$p_A(\lambda) = (-1)^m \lambda^m + (-1)^{m-1} \text{tr}(A) \lambda^{m-1} + q(\lambda)$$

donde  $q(\lambda)$  es un polinomio de grado menor que  $m - 1$ .

*Demostración.* Vamos a demostrarlo por inducción, para ello vamos a ver primero que se cumple para  $m = 1$ .

Si  $m = 1$ ,  $A_1 = (a_{11})$ , entonces

$$p_{A_1}(\lambda) = \det(a_{11} - \lambda) = -\lambda + a_{11} = (-1)^1 \lambda + \text{tr}(A_1) \lambda^{1-1}.$$

Suponiendo que se cumple para  $m$  vamos a ver que se cumple para  $m + 1$ .

$$A_{m+1} = \begin{pmatrix} a_{11} & \cdots & a_{1m+1} \\ \vdots & \ddots & \vdots \\ a_{m+11} & \cdots & a_{m+1m+1} \end{pmatrix} \text{ entonces } p_{A_{m+1}}(\lambda) = \begin{vmatrix} a_{11} - \lambda & \cdots & a_{1m+1} \\ \vdots & \ddots & \vdots \\ a_{m+11} & \cdots & a_{m+1m+1} - \lambda \end{vmatrix}.$$

Para calcular el determinante desarrollamos por la primera columna. Si nos fijamos, los elementos de las filas posteriores a la primera van a multiplicar su valor por un determinante donde sólo intervienen  $(m + 1) - 2$  variables  $\lambda$  ya que se “eliminarán” las de la primera columna y la de la fila correspondiente, por lo que, a lo sumo, formarán un polinomio de grado menor o igual a  $m - 2$ . Es decir, para calcular los coeficientes de grado  $m + 1$  y  $m$ , sólo intervendrá el primer elemento de la columna.

$$(a_{11} - \lambda) \cdot \begin{vmatrix} a_{22} - \lambda & \cdots & a_{2m+1} \\ \vdots & \ddots & \vdots \\ a_{m+12} & \cdots & a_{m+1m+1} - \lambda \end{vmatrix}.$$

Podemos notar que el determinante sería el polinomio característico de una matriz de orden  $m$  que, por lo tanto, cumple que  $p_{A_m}(\lambda) = (-1)^m \lambda^m + (-1)^{m-1} \text{tr}(A_m) \lambda^{m-1} + q(\lambda)$  donde  $q(\lambda)$  es un polinomio de grado menor que  $m - 1$ . Tenemos entonces que

$$(a_{11} - \lambda) \cdot \begin{vmatrix} a_{22} - \lambda & \cdots & a_{2m+1} \\ \vdots & \ddots & \vdots \\ a_{m+12} & \cdots & a_{m+1m+1} - \lambda \end{vmatrix} =$$

$$\begin{aligned} & (a_{11} - \lambda)((-1)^m \lambda^m + (-1)^{m-1}(a_{22} + \cdots + a_{m+1m+1})\lambda^{m-1} + q(\lambda)) \\ &= (-1)^m a_{11} \lambda^m + (-1)(-1)^m \lambda^{m+1} + (-1)(-1)^{m-1}(a_{22} + \cdots + a_{m+1m+1})\lambda^{m-1} + r(\lambda) \\ &= (-1)^{m+1} \lambda^{m+1} + (-1)^m (a_{11} + a_{22} + \cdots + a_{m+1m+1})\lambda^{m-1} + r(\lambda) \\ &= (-1)^{m+1} \lambda^{m+1} + (-1)^m \text{tr}(A_{m+1}) \lambda^{m-1} + r(\lambda). \end{aligned}$$

Donde  $r(\lambda)$  es un polinomio de grado menor que  $m$ . Por lo que se cumple para  $m+1$ .  $\square$

La clave para la demostración del teorema de Frobenius reside en el siguiente lema.

**Lema 10.2.** *El conjunto  $V$  de todos los elementos  $a$  de  $\mathcal{D}$  tales que  $a^2 \leq 0$  es un subespacio vectorial de  $\mathcal{D}$  de dimensión  $n - 1$ . Además  $\mathcal{D} = \mathbb{R} \oplus V$  como espacios vectoriales reales, lo que implica que  $V$  genera  $\mathcal{D}$  como álgebra.*

*Demostración.* Sea  $m$  la dimensión del espacio vectorial real  $\mathcal{D}$ , tomamos  $a$  en  $\mathcal{D}$  con polinomio característico  $p(x)$ . Por el Teorema Fundamental del Álgebra, podemos escribir  $p(x)$  como:

$$p(x) = (x - t_1) \cdots (x - t_r)(x - z_1)(x - \bar{z}_1) \cdots (x - z_s)(x - \bar{z}_s), \quad t_i \in \mathbb{R}, \quad z_j \in \mathbb{C} \setminus \mathbb{R}.$$

Podemos reescribir  $p(x)$  en términos de los polinomios  $Q(z; x)$ :

$$p(x) = (x - t_1) \cdots (x - t_r)Q(z_1; x) \cdots Q(z_s; x).$$

Como  $z_j \in \mathbb{C} \setminus \mathbb{R}$ , los polinomios  $Q(z_j; x)$  son todos irreducibles en  $\mathbb{R}$ . Por el Teorema de Cayley-Hamilton,  $p(a) = 0$  y como  $\mathcal{D}$  es un álgebra de división, se tiene que o  $a - t_i = 0$  para algún  $i$ , o  $Q(z_j; a) = 0$  para algún  $j$ . El primer caso implica que  $a$  es real. En el segundo caso, se tendría que  $Q(z_j; x)$  es el polinomio mínimo de  $a$ . Como  $p(x)$  es el polinomio característico,  $p(a) = 0$ , como hemos dicho previamente, al ser el polinomio mínimo el polinomio mónico  $q(x)$  de menor grado que cumple que  $q(a) = 0$ , esto quiere decir que  $q(x)|p(x)$  luego  $p(x) = q(x) \cdot c(x)$  para algún  $c(x)$ . Las raíces de  $q(x)$  son raíces de  $p(x)$  ya que si  $\lambda$  es un valor propio de  $A$ ,  $Av = \lambda v$  para  $v$  un vector propio. Así,  $A^2v = A\lambda v = \lambda Av = \lambda^2 v$ . Del mismo modo,  $A^k \lambda = \lambda^k v$ . De esta forma  $q(A) \cdot v = q(\lambda) \cdot v$ . Pero  $q(A) = 0$ , lo que lleva a  $q(\lambda) \cdot v = 0$  y esto fuerza  $q(\lambda) = 0$ . Por lo tanto el polinomio característico tiene las mismas raíces complejas que el polinomio mínimo y como es real

se sigue que

$$p(x) = Q(z_j; x)^k = (x^2 - 2 \operatorname{Re}(z_j)x + |z_j|^2)^k. \quad (10.1)$$

Ya que  $p(x)$  es el polinomio característico de  $a$ , el coeficiente de  $x^{2k-1}$  en  $p(x)$  es la traza de  $a$  salvo signo. Por lo tanto, de la ecuación tenemos que la traza de  $a$  se anula si y sólo si  $\operatorname{Re}(z_j) = 0$ , en otras palabras,  $\operatorname{tr}(a) = 0$  si y sólo si  $a^2 = -|z_j|^2 \leq 0$ .

Así que  $V$  es el subconjunto de todos los  $a$  con traza nula. En particular, es un subespacio vectorial. El Teorema de Rango-Nulidad implica que  $V$  tiene dimensión  $n-1$  ya que es el núcleo de  $\operatorname{tr} : \mathcal{D} \rightarrow \mathbb{R}$ . Como  $\mathbb{R}$  y  $V$  son disjuntos (esto es, satisfacen  $\mathbb{R} \cap V = \{0\}$ ), y la suma de sus dimensiones es  $n$ , tenemos que  $\mathcal{D} = \mathbb{R} \oplus V$ .  $\square$

*Demostración.* (*Teorema de Frobenius*). Para  $a, b$  en  $V$  definimos  $B(a, b) = (-ab - ba)/2$ . Por la identidad  $(a + b)^2 - a^2 - b^2 = ab + ba$  se tiene que  $B(a, b)$  es real. Además, como  $a^2 \leq 0$  para  $a \in V$ , tenemos que  $B(a, a) > 0$  para  $a$  no nulo. Así pues,  $B$  es una forma bilineal simétrica definida positiva, en otras palabras, un producto escalar en  $V$ .

Sea  $W$  un subespacio de  $V$  que genera  $\mathcal{D}$  como un álgebra de forma minimal. Sea  $e_1, \dots, e_n$  una base ortonormal de  $W$  con respecto de  $B$ . La ortonormalidad implica que:

$$e_i^2 = -1, \quad e_i e_j = -e_j e_i.$$

Si  $n = 0$ , entonces  $\mathcal{D}$  es isomorfo a  $\mathbb{R}$ .

Si  $n = 1$ , entonces  $\mathcal{D}$  está generado por 1 y  $e_1$  está determinado por la propiedad  $e_1^2 = -1$ . Por lo tanto, es isomorfo a  $\mathbb{C}$ .

Si  $n = 2$ , se ha demostrado anteriormente que  $\mathcal{D}$  está generado por  $1, e_1, e_2$  con las siguientes propiedades

$$e_1^2 = e_2^2 = -1, \quad e_1 e_2 = -e_2 e_1, \quad (e_1 e_2)(e_1 e_2) = -1,$$

que son precisamente las propiedades de  $\mathbb{H}$ .

Si  $n > 2$ , entonces  $\mathcal{D}$  no puede ser un álgebra de división. Supongamos que  $n > 2$ . Sea  $u = e_1 e_2 e_n$ . Podemos ver que  $u^2 = 1$  (esto sólo funciona si  $n > 2$ ) ya que  $u^2 = (e_1 e_2 e_n)(e_1 e_2 e_n) = e_1 e_2 - e_1 e_n e_2 e_n = e_1 e_2 - e_1 - e_2 e_n e_n = (e_1 e_2)(e_1 e_2)e_n^2 = -1 \cdot -1 = 1$ . Si  $\mathcal{D}$  fuese un álgebra de división,  $0 = u^2 - 1 = (u - 1)(u + 1)$  lo que implica que  $u = \pm 1$ , que a su vez significa que  $e_n = \pm e_1 e_2$  y, por lo tanto,  $e_1, \dots, e_{n-1}$  generan  $\mathcal{D}$ . Lo que contradice la minimalidad de  $W$ .  $\square$

Como consecuencia de esta demostración, las únicas álgebras de división conmutativas son  $\mathbb{R}$  y  $\mathbb{C}$ . Notemos que  $\mathbb{H}$  no es un álgebra sobre  $\mathbb{C}$ . Si lo fuera, el centro de  $\mathbb{H}$  debería contener a  $\mathbb{C}$ , pero el centro de  $\mathbb{H}$  es  $\mathbb{R}$  (el centro es el conjunto de elementos de  $\mathbb{H}$  que comutan con todos los elementos de  $\mathbb{H}$ , es decir, los  $x \in \mathbb{H}$  que cumplen que  $xq = qx$  para todo  $q \in \mathbb{H}$ ). Por lo tanto, la única álgebra de división de dimensión finita sobre  $\mathbb{C}$  es  $\mathbb{C}$ .

$\square$



# 11. Cuaterniones y rotaciones en 3D

Para entender cómo los números complejos se relacionan con las rotaciones usamos la fórmula de Euler:

$$e^{i\varphi} = \cos \varphi + i \sin \varphi.$$

Un número complejo  $z = x+iy$  se puede escribir como  $z = |z|e^{i\varphi}$  donde  $|z|$  es el módulo de  $z$  y  $\varphi$  es el ángulo que del vector  $(x, y)$  con respecto al eje positivo de la  $x$ . El número  $z$  se puede rotar un ángulo  $\theta$  multiplicando  $z$  por  $e^{i\theta}$  ya que  $ze^{i\theta} = |z|e^{i\varphi}e^{i\theta} = |z|e^{i(\varphi+\theta)}$ . Tanto  $z$  como  $ze^{i\theta}$  tienen el mismo módulo pero varía el ángulo que forma con el eje  $x$ . También podemos denotar la multiplicación como

$$\begin{aligned} z' &= ze^{i\theta} = (x + iy)(e^{i\theta}) = (x + iy)(\cos \theta + i \sin \theta) \\ &= x \cos \theta - y \sin \theta + i(x \sin \theta + y \cos \theta). \end{aligned}$$

Podemos escribir esta rotación entonces como la matriz  $2 \times 2$ :

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$

Para las rotaciones en el espacio de tres dimensiones se usan los cuaterniones unidad, es decir, los cuaterniones cuya norma vale 1, esto es,  $|q| = |\bar{q}| = |q^{-1}| = 1$ . Para realizar una rotación completa  $\theta$  se necesitan dos cuaterniones, donde cada cuaternion aportará una rotación de ángulo  $\theta/2$ . La fórmula de Euler se puede sustituir por una variante para el cuaternion donde  $q = e^{v\theta/2} = \cos(\theta/2) + v \sin(\theta/2)$  y  $v = v_x i + v_y j + v_z k$ . Al tipo de cuaternion  $v$  se le denomina cuaternion puro imaginario ya que la parte escalar del cuaternion es cero.

Vamos a ver cuál es la fórmula y de dónde viene. Para ello es necesario recordar la definición del producto vectorial y algunas propiedades de éste.

El producto vectorial es una operación binaria sobre dos vectores en un espacio vectorial euclíadiano tridimensional orientado denotada por  $\times$ . Dados dos vectores linealmente independientes  $a$  y  $b$ , el producto vectorial,  $a \times b$ , es un vector perpendicular a ambos, y, por lo tanto, normal al plano que los contiene. Una de las propiedades de este producto es que es anticomutativo, es decir,  $a \times b = -b \times a$ . Además, si tenemos un doble producto vectorial éste se puede calcular como  $a \times (b \times c) = b(a \cdot c) - c(a \cdot b)$ , donde  $\cdot$  denota el producto escalar. Si el doble producto fuese de la forma  $(a \times b) \times c$  se podría calcular de la misma manera ya que  $(a \times b) \times c = -c \times (a \times b) = (a \cdot c)b - (b \cdot c)a$ .

Una de las formas de calcular el vector que surge del producto vectorial de dos vectores

es mediante el determinante de la matriz que se muestra a continuación. Si tomamos  $a = (a_1, a_2, a_3)$  y  $b = (b_1, b_2, b_3)$ , entonces

$$\begin{aligned} a \times b &= \begin{vmatrix} i & j & k \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = \begin{vmatrix} a_2 & a_3 \\ b_2 & b_3 \end{vmatrix} i - \begin{vmatrix} a_1 & a_3 \\ b_1 & b_3 \end{vmatrix} j + \begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix} k \\ &= (a_2 b_3 - a_3 b_2) i + (a_3 b_1 - a_1 b_3) j + (a_1 b_2 - a_2 b_1) k. \end{aligned}$$

Si recordamos del capítulo 9, la multiplicación de dos cuaterniones  $P = (p_0 + p_1 i + p_2 j + p_3 k)$  y  $Q = (q_0 + q_1 i + q_2 j + q_3 k)$  viene dada por

$$\begin{aligned} PQ &= p_0 q_0 - p_1 q_1 - p_2 q_2 - p_3 q_3 + (p_0 q_1 + p_1 q_0 + p_2 q_3 - p_3 q_2) i \\ &\quad + (p_0 q_2 - p_1 q_3 + p_2 q_0 + p_3 q_1) j + (p_0 q_3 + p_1 q_2 - p_2 q_1 + p_3 q_0) k \\ &= p_0 q_0 - (p_1 q_1 + p_2 q_2 + p_3 q_3) + p_0 q_1 i + p_1 q_0 i + (p_2 q_3 - p_3 q_2) i \\ &\quad + p_0 q_2 j + p_2 q_0 j + (p_3 q_1 - p_1 q_3) j + p_0 q_3 k + p_3 q_0 k + (p_1 q_2 - p_2 q_1) k \\ &= p_0 q_0 - (p_1 q_1 + p_2 q_2 + p_3 q_3) + p_0 (q_1 i + q_2 j + q_3 k) + q_0 (p_1 i + p_2 j + p_3 k) \\ &\quad (p_2 q_3 - p_3 q_2) i + (p_3 q_1 - p_1 q_3) j + (p_1 q_2 - p_2 q_1) k. \end{aligned}$$

Un cuaternion puede denotarse como  $q_0 + q$ . Donde  $q_0$  es la parte escalar del cuaternion y  $q$  es la parte vectorial ( $q = q_1 i + q_2 j + q_3 k$ ). Entonces, si tomamos dos cuaterniones  $Q = q_0 + q$  y  $P = p_0 + p$  la multiplicación de éstos se puede ver como

$$PQ = p_0 q_0 - p \cdot q + p_0 q + q_0 p + p \times q.$$

Donde  $p \cdot q$  es el producto escalar y  $p \times q$  es el producto vectorial. Tendríamos entonces que la parte escalar del producto sería  $p_0 q_0 - p \cdot q$  y la parte vectorial  $p_0 q + q_0 p + p \times q$ .

Tenemos que tener en cuenta que

$$p \cdot q = |p||q| \cos \theta,$$

donde  $\theta$  es el ángulo que forman los vectores  $p$  y  $q$ .

Un cuaternion  $r$  imaginario puro, es decir,  $r = xi + yj + zk$ , puede identificarse con el punto  $(x, y, z) \in \mathbb{R}^3$ . Tomamos también un cuaternion imaginario puro unidad  $q = q_1 i + q_2 j + q_3 k$ . Entonces,

$$qr\bar{q} = q(-r \cdot \bar{q} + r \times \bar{q}) = \cancel{-q(r \times q)}^0 + (-r \cdot \bar{q})q + q \times (r \times \bar{q}),$$

el segundo producto escalar se anula ya que el producto vectorial produce un vector ortogonal a  $\bar{q}$ , que al ser proporcional a  $q$  (ya que  $\bar{q} = -q$ ), también es ortogonal a  $q$ . Al ser  $q \times (r \times \bar{q})$  un doble producto vectorial tenemos que

$$q \times (r \times \bar{q}) = (q \cdot \bar{q})r - (q \cdot r)\bar{q} = -r + (r \cdot q)q.$$

Por lo que, finalmente, tenemos que

$$\begin{aligned} qr\bar{q} &= (-r \cdot \bar{q})q + (-r + (r \cdot q)q) \\ &= (r \cdot q)q - r + (r \cdot q)q \\ &= -r + 2(r \cdot q)q. \end{aligned}$$

Debido a la fórmula del coseno con el producto escalar tendríamos que

$$qr\bar{q} = -r + 2|r||q|\cos(\theta)q = -r + 2|r|\cos(\theta)q,$$

donde  $\theta$  es el ángulo entre los vectores  $(x, y, z)$  y  $(q_1, q_2, q_3)$ . También cabe destacar que  $qr\bar{q}$  es un cuaternion imaginario puro que corresponde, por tanto, a algún punto de  $\mathbb{R}^3$ .

Sea  $Q$  el plano en  $\mathbb{R}^3$  que pasa por el origen y es normal a  $(q_1, q_2, q_3)$ . Tenemos que  $|r|\cos(\theta)q$  es la proyección de  $(x, y, z)$  en la dirección  $(q_1, q_2, q_3)$ , por lo que

$$r' \equiv -qr\bar{q} = r - 2|r|\cos(\theta)q$$

representa la imagen de  $r$  bajo simetría en el plano  $Q$  (Figura 11.1).

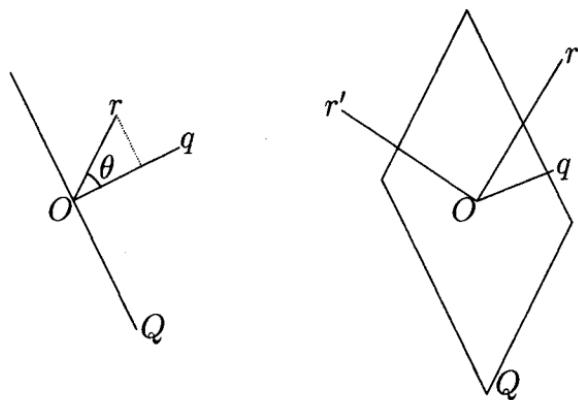


Figura 11.1.: En la imagen de la izquierda se puede observar la representación de los cuaterniones  $q$  y  $r$  como vectores del espacio y el ángulo  $\theta$  entre éstos siendo  $Q$  el plano normal a  $q$ . En la imagen de la derecha se muestra la representación de  $r'$ . Imagen extraída de [1].

En otras palabras, si identificamos los cuaterniones imaginarios puros con puntos del espacio, la aplicación  $r \mapsto -qr\bar{q}$  representa la simetría en el plano normal a  $q$  que pasa por el origen si  $q$  es unitario.

Vamos a ver ahora que esto se cumple para la composición de estas simetrías, para ello tomamos dos cuaterniones unitarios imaginarios puros  $q$  y  $p$ , tenemos que

$$r'' \equiv -pr'\bar{p} = -p(-qr\bar{q})\bar{p} = (pq)r\overline{(pq)},$$

$r''$  representa la imagen de  $r$  bajo simetría en el plano  $Q$ , normal a  $q$ , seguida de la simetría en el plano  $P$ , normal a  $p$ . Teniendo en cuenta que  $p \times q = |p||q| \sin(\phi)n$ , donde  $\phi$  es el ángulo entre los vectores  $p$  y  $q$  y  $n$  es un vector unitario ortogonal a  $p$  y  $q$  cuyo sentido viene dado por la *regla de la mano derecha*. Denotando  $p = p_1i + p_2j + p_3k$ , tenemos que

$$pq = -p \cdot q + q \times p = -|p||q| \cos \phi + |p||q| \sin(\phi)n = -\cos \phi + \sin(\phi)n, \quad (11.1)$$

donde  $n$  es un cuaternión imaginario puro unitario que representa una dirección normal a las direcciones de  $q$  y  $p$ , con el sentido dado por la *regla de la mano derecha* como se ha comentado previamente. Por lo tanto, la dirección de  $n$  se encuentra en la intersección de los planos  $Q$  y  $P$ , y  $\phi$  es uno de los ángulos entre dichos planos (Figura 11.2). A pesar de que  $q$  y  $p$  son cuaterniones imaginarios puros, en general,  $pq$  no lo es, sin embargo sí es unitario ya que  $|pq| = \cos^2 \phi + \sin^2 \phi |n| = \cos^2 \phi + \sin^2 \phi = 1$ .

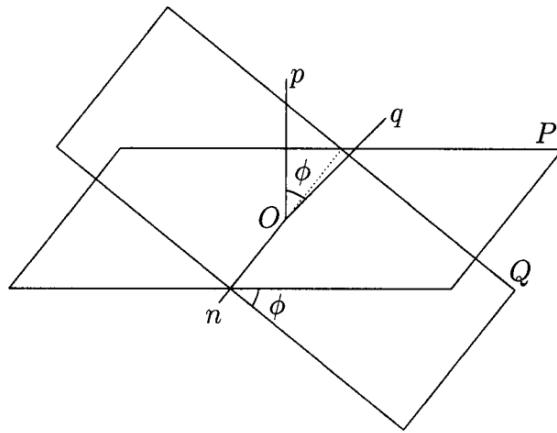


Figura 11.2.: Representación de los planos  $P$  y  $Q$ , la intersección de éstos  $n$  y el ángulo  $\phi$  entre ellos. Imagen extraída de [1].

La aplicación  $r \mapsto (pq)r\overline{(pq)}$ , siendo la composición de dos simetrías, representa una rotación en  $\mathbb{R}^3$  alrededor del origen. El eje de dicha rotación está en la dirección de  $n$ , ya que  $n$  queda invariante en la rotación por estar en los planos  $P$  y  $Q$  y los puntos de cada plano quedan invariantes por la simetría en su plano.

La ecuación 11.1 implica que la rotación depende sólo del ángulo  $\phi$  entre los planos  $P$  y  $Q$ , y de la dirección de la línea que interseca los planos. Podemos ver entonces que la aplicación  $r \mapsto (pq)r\overline{(pq)}$  representa una rotación en  $\mathbb{R}^3$  por un ángulo  $2\phi$ . Si tomamos un punto arbitrario  $(x, y, z)$ , que corresponde con el cuaternión  $r$ , y rotamos los planos  $Q$  y  $P$  alrededor de su línea de intersección, manteniendo fijo el ángulo entre los planos, hasta que el punto esté en el plano  $\tilde{Q}$  que se obtiene por esta rotación (Figura 11.3). Entonces,  $(x, y, z)$  queda fijo bajo simetría en  $\tilde{Q}$ , por lo que la imagen de  $(x, y, z)$  tras la simetría en  $\tilde{Q}$  seguida de la simetría en  $\tilde{P}$  lo que da un punto tal que la línea que

lo une con el origen forma un ángulo  $2\phi$  con la línea que une el origen con  $(x, y, z)$ . De este modo, se obtiene que la rotación alrededor del eje dado por el cuaternion unitario imaginario puro  $n$ , por un ángulo  $\alpha$ , está representada por la aplicación

$$r \mapsto (\cos \frac{\alpha}{2} + \sin \frac{\alpha}{2}n)r(\cos \frac{\alpha}{2} - \sin \frac{\alpha}{2}n) \quad (11.2)$$

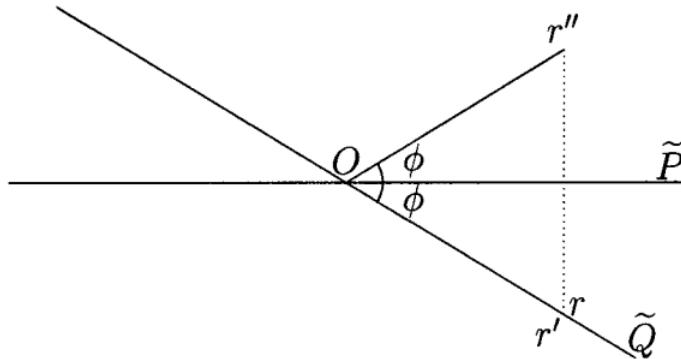


Figura 11.3.: Representación de la rotación de los planos  $P$  y  $Q$  en  $\tilde{P}$  y  $\tilde{Q}$ , siendo la intersección de éstos una línea perpendicular al plano de la imagen. Imagen extraída de [1].

Vamos a ver que cualquier cuaternion unitario se puede representar de la forma  $q = \cos(\alpha/2) + \sin(\alpha/2)n$ , para algún  $\alpha \in \mathbb{R}$  y para algún cuaternion unitario imaginario puro  $n$ .

Sea  $q = q_0 + q_1i + q_2j + q_3k$  con  $q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1$ , lo que implica que  $q_1^2 + q_2^2 + q_3^2 = 1 - q_0^2$ . Podríamos tomar entonces  $n = 1/\sqrt{1 - q_0^2}(q_1i + q_2j + q_3k)$ . La parte escalar equivale a  $q_0$  por lo que  $\cos(\alpha/2) = q_0$ , lo que implica que  $\alpha = 2 \arccos(q_0)$ . Podemos ver entonces que de esta manera

$$\begin{aligned} \cos(\alpha/2) + \sin(\alpha/2)n &= \cos(\alpha/2) + \sin(\alpha/2)(1/\sqrt{1 - q_0^2})(q_1i + q_2j + q_3k) \\ &= \cos(\alpha/2) + \sin(\alpha/2)(1/\sqrt{1 - \cos^2(\alpha/2)})(q_1i + q_2j + q_3k) \\ &= \cos(\alpha/2) + \sin(\alpha/2)(1/\sqrt{\sin^2(\alpha/2)})(q_1i + q_2j + q_3k) \\ &= \cos(\alpha/2) + \sin(\alpha/2)(1/\sin(\alpha/2))(q_1i + q_2j + q_3k) \\ &= \cos(\alpha/2) + (q_1i + q_2j + q_3k) \\ &= q_0 + q_1i + q_2j + q_3k = q. \end{aligned}$$

Podemos concluir que cualquier cuaternion unitario  $q_0 + q_1i + q_2j + q_3k$  se puede poner de la forma  $q = \cos(\alpha/2) + \sin(\alpha/2)n$  con  $\alpha = 2 \arccos(q_0)$  y  $n = 1/\sqrt{1 - q_0^2}(q_1i + q_2j + q_3k)$ . Por lo tanto, cada cuaternion unitario define una rotación en  $\mathbb{R}^3$ , alrededor del origen

dada por la ecuación 11.2.

Cabe destacar que un cuaternión unitario  $q$  y su negativo  $-q$  definen la misma rotación. Si  $q = \cos(\alpha/2) + \sin(\alpha/2)n$ , entonces  $-q = -\cos(\alpha/2) - \sin(\alpha/2)n = \cos[(\alpha + 2\pi)/2] - \sin[(\alpha + 2\pi)/2]n$ , es decir, forman el mismo ángulo salvo una vuelta completa, lo que equivale al mismo punto en  $\mathbb{R}^3$ . La correspondencia entre cuaterniones unitarios y rotaciones en  $\mathbb{R}^3$  es dos a uno, siendo ésta un homomorfismo de grupos con la multiplicación de cuaterniones y la composición de rotaciones. Si  $q$  y  $q'$  son dos cuaterniones unitarios, la composición de la rotación dada por  $q$  seguida de la rotación dada por  $q'$  se representa por

$$r \mapsto q'(qr\bar{q})\bar{q}' = (q'q)r\overline{(q'q)}.$$

Esto hace que la rotación de un punto a través de un eje arbitrario utilizando cuaterniones sea un proceso sencillo. Por ejemplo, si queremos realizar una rotación de ángulo  $\pi/3$  alrededor del eje  $(0, 0, 1)$  deberíamos tomar el cuaternión

$$q = \cos\left(\frac{\pi}{6}\right) + \sin\left(\frac{\pi}{6}\right)(0i + 0j + k) = \frac{\sqrt{3}}{2} + \frac{k}{2}.$$

Entonces para un punto del espacio  $(x, y, z)$  se tendría la rotación como  $r = (xi + yj + zk) \mapsto qr\bar{q}$ , que, como vimos previamente, es otro cuaternión imaginario puro que se corresponde con otro punto de  $\mathbb{R}^3$ .

Las rotaciones en aplicaciones 3D se representan generalmente en una de dos maneras, ángulos de cuaterniones o de Euler. Los ángulos de Euler son tres ángulos introducidos por Leonhard Euler para describir la orientación de un cuerpo rígido con respecto a un sistema de referencia fijo. Estos ángulos son los ángulos que forman los ejes del nuevo sistema de referencia con respecto a los ejes del sistema de referencia base fijo.

*Unity* usa cuaterniones internamente, pero muestra valores de los ángulos Euler equivalentes en el inspector para que sea fácil de editar.

Se usan los cuaterniones ya que los ángulos de Euler sufren de Gimbal Lock, esto es que cuando se aplican las tres rotaciones a su vez, es posible que la primera o segunda rotación dé lugar al tercer eje que apunta en la misma dirección que uno de los ejes anteriores. Esto significa que se ha perdido un “grado de libertad”, porque el tercer valor de rotación no puede aplicarse alrededor de un eje único. Un cuaternión no puede representar una rotación de más de 180 grados, por lo que, si se especifica una rotación que no se puede representar como un cuaternión, internamente *Unity* lo transformará en un cuaternión que oriente el objeto de la misma manera que el resultado de la rotación previa.

# Bibliografía

- [1] G.F. Torres del Castillo. La representación de rotaciones mediante cuaterniones. *Miscelánea Matemática*, 29, 1999. Departamento de Física Matemática del Instituto de Ciencias de la Universidad Autónoma de Puebla México. URL: [https://miscelaneamatematica.org/download/tbl\\_articulos.pdf2.b8d4415d314c3f7d.746f727265735f632e706466.pdf](https://miscelaneamatematica.org/download/tbl_articulos.pdf2.b8d4415d314c3f7d.746f727265735f632e706466.pdf).
- [2] Aristóteles. *Poética*. Alianza Editorial, 2004. Traducción de Alicia Villar.
- [3] Unity Documentation. <https://docs.unity3d.com/>, 2022. Documentación completa de Unity.
- [4] Unity Forum. <https://forum.unity.com/>, 2022. Foro de Unity.
- [5] Unity Learn. <https://learn.unity.com/>, 2022. Página para el aprendizaje de Unity proporcionada por la herramienta.
- [6] Salario Programador Junior. <https://es.indeed.com/career/programador-junior/salaries>, 2022. Página web donde se ha obtenido el salario medio de un programador junior.
- [7] Salario Alquiler Coworking. [https://cincodias.elpais.com/cincodias/2014/10/17/autonomos/1413556226\\_514749.html](https://cincodias.elpais.com/cincodias/2014/10/17/autonomos/1413556226_514749.html), 2014. Noticia donde se ha obtenido el precio medio de un espacio de co-working.
- [8] Unity Asset Store. <https://assetstore.unity.com/>, 2022. Página donde se encuentran assets para importarlos en los proyectos de Unity.
- [9] CGtrader. <https://www.cgtrader.com/>, 2022. Página donde se ha encontrado el modelo del escenario.
- [10] Mixamo. <https://www.mixamo.com>, 2022. Página donde se han encontrado las animaciones y el modelo del actor.
- [11] Freesound. <https://freesound.org/>, 2022. Página donde se han encontrado los sonidos de las pisadas.
- [12] Audacity. <https://audacity.es/>, 2022. Software libre de edición de audio usado para alterar los sonidos de las pisadas.
- [13] Google Fonts. <https://fonts.google.com/>, 2022. Fuentes e iconos de Google.
- [14] Iluminación en el teatro. <https://billwilliams.ca/resources/sld/sld-200.htm>, 1999. Información de la iluminación en el montaje de escenas de teatro.
- [15] Johannes C. Familton. *Quaternions: A History of Complex Noncommutative Rotation Groups in Theoretical Physics*. Columbia University, 2015. A thesis submitted in partial fulfillment of the requirements for the degree of Ph.D of Columbia University. URL: <https://arxiv.org/pdf/1504.04885.pdf>.
- [16] JJ O'Connor and EF Robertson. Biografía de Hamilton. <https://mathshistory.st-andrews.ac.uk/Biographies/Hamilton/>, Junio 1998.

- [17] William Edwin Hamilton. Advertisement Elements of Quaternions. [https://mathshistory.st-andrews.ac.uk/Extras/Hamilton\\_elements/](https://mathshistory.st-andrews.ac.uk/Extras/Hamilton_elements/), Agosto 2006.
- [18] Cesare Burali-Forti. Problems of acceptance of the vector calculus. [https://mathshistory.st-andrews.ac.uk/Extras/Vector\\_calculus\\_problems/](https://mathshistory.st-andrews.ac.uk/Extras/Vector_calculus_problems/), Enero 2015.
- [19] Carl B. Boyer. *Historia de la matemática*. Alianza Editorial, 1968. Versión española de Mariano Martínez Pérez.
- [20] Frobenius Theorem. [https://en.wikipedia.org/wiki/Frobenius\\_theorem\\_\(real\\_division\\_algebras\)](https://en.wikipedia.org/wiki/Frobenius_theorem_(real_division_algebras)), Octubre 2021.
- [21] Luis Merino y Evangelina Santos. *Álgebra Lineal con métodos elementales*. Thomson Editores Spain, 2006. Versión española de Mariano Martínez Pérez.