GRADO EN MATEMÁTICA COMPUTACIONAL

PROYECTO FINAL DE GRADO

Los cuaterniones y su importancia en la representación gráfica por ordenador

Autor:
Rocío CARRATALÁ SÁEZ

Supervisor:
David RODRÍGUEZ SÁNCHEZ
Tutor académico:
Vicente José PALMER ANDREU

Fecha de lectura: 30 de julio de 2015
Curso académico 2014/2015

Resumen

El presente Proyecto Final de Grado aúna una recopilación de los aspectos más importantes de mi estancia en prácticas y un estudio sobre las isometrías lineales y afines y los cuaterniones.

Mi estancia en prácticas estuvo dedicada a la programación de videojuegos colaborando con el equipo Catness, uno de los que integran la empresa PaynoPain, situada en el Espaitec de la Universitat Jaume I de Castellón. Concretamente, programé funciones que permiten al usuario editar un cubo moviendo sus vértices o sus caras, así como realizar extrusiones del mismo.

Durante la estancia, descubrí la existencia de los ángulos de Euler, utilizados en la descripción de giros en el cubo, y de lo que constituye su principal defecto: el *Gimbal Lock*, que podría traducirse como "bloqueo giroscópico". La búsqueda de información sobre ellos me llevó hasta los cuaterniones; unos números, *creados* en el siglo XIX, que son de la forma:

$$q = a + bi + cj + dk$$

donde $a, b, c, d \in \mathbb{R}$ con $i^2 = j^2 = k^2 = ijk = -1$. Precisamente a su estudio dedico un capítulo completo en este documento.

Cabe añadir que actualmente los cuaterniones son muy útiles en la representación gráfica por ordenador, debido, entre otras cosas, a la posibilidad que ofrecen de representar con ellos rotaciones en el espacio tridimensional evitando el *Gimbal Lock*.

Palabras clave

Geometría afín, cuaterniones, Unreal Engine, videojuegos.

Keywords

Affine geometry, quaternions, Unreal Engine, videogames.

Índice general

Capítulo 1

Introducción.

1.1. Contexto y motivación del proyecto.

El Proyecto Final de Grado (en adelante, PFG) representa, junto con la estancia en prácticas, dieciocho de los doscientos cuarenta créditos que componen el Grado en Matemática Computacional.

En el momento de elegir estancia en prácticas y, consecuentemente, temática sobre la que desarrollar mi PFG, escogí la opción de formar parte del equipo Catness, dentro de la empresa PaynoPain, situada en el Espaitec de la Universitat Jaume I (en adelante, UJI). Tomé esta decisión motivada por el hecho de que la geometría es la rama de las matemáticas que me resulta más atractiva y, a su vez, es muy importante en el mundo de los videojuegos, al que el mencionado equipo dedica su trabajo.

La geometría cobra especial importancia en los videojuegos, no solo a nivel de reproducir en ellos isometrías comunes en el espacio como rotaciones o traslaciones, sino también estudiando cómo tratarlas y programarlas en los motores informáticos enfocados al desarrollo de videojuegos. Tanto es así que, en mi PFG, consideré de vital importancia dedicar esfuerzo a la compresión de la representación de figuras en el espacio de tres dimensiones y de isometrías aplicadas sobre ellas mediante cuaterniones, prestando especial atención a las rotaciones.

Los cuaterniones son números de la forma:

$$q = a + bi + cj + dk$$

donde $a, b, c, d \in \mathbb{R}$ con $i^2 = j^2 = k^2 = ijk = -1$. Fueron *creados* por W. R. Hamilton en el siglo XIX y su conjunto, $\mathbb{H}$, satisface casi todas las propiedades usuales del álgebra excepto la conmutativa del producto, por lo que $(\mathbb{H}, +)$ es un grupo conmutativo y $(\mathbb{H}, \cdot)$ es un grupo no conmutativo. En general, la terna $(\mathbb{H}, +, \cdot)$ constituye un anillo no conmutativo con elemento neutro. Es importante destacar que los cuaterniones pueden asociarse a un espacio vectorial sobre $\mathbb{R}^4$.

A lo largo de este documento explicaré los aspectos más relevantes de mi estancia en prácticas - incluyendo alusiones a los objetivos y a la planificación inicial de las tareas, su consecución o no en los plazos establecidos para las mismas y los resultados finales obtenidos - y presentaré algunos de los aspectos más importantes sobre las isometrías en espacios afines euclídeos y sobre los cuaterniones y sus usos, todo de una forma que espero que resulte a la par interesante y amena.

Capítulo 2

Una matemática computacional en prácticas.

2.1. Catness: mi primera experiencia programando videojuegos.

PaynoPain es una empresa de reciente creación (inició su actividad en el año 2011) dedicada al desarrollo de productos basándose en el aprovechamiento de las nuevas tecnologías. Centran su principal actividad en el diseño de sistemas de seguridad e identificación de usuarios, aunque también existe entre sus equipos de trabajo uno dedicado a los videojuegos. Es en este último en el que yo me integré durante mi estancia en prácticas.

El equipo Catness lo formaban, a mi llegada, dos programadores: David Rodríguez (quien fue mi supervisor) y Marco Domingo. Durante mi colaboración con ellos se unió al equipo un tercer trabajador, Adrian Virlan, como diseñador gráfico encargado del modelado tanto de los personajes de los videojuegos como de sus escenarios.

Actualmente en Catness se está desarrollando un proyecto ambicioso: Hive. Dada la complejidad y el avanzado estado de dicho videojuego, resultaba complicado, en el tiempo disponible en la estancia en prácticas, conseguir mi inmersión y colaboración en dicho proyecto, por lo que David y Marco buscaron alternativas que yo pudiera abordar. Entre las opciones que me ofrecieron, tres de los proyectos presentaban mayor componente geométrico: "*Sculpt!*", "*Our way*" y diseñar un simulador de entornos.

Participar en el desarrollo de "*Sculpt!*" consistía en contribuir a la implementación de un cubo con esquinas redondeadas moldeable mediante clicks para representar el efecto de esculpir un busto. "*Our way*" es una idea de videojuego que pretende recrear batallas en un laberinto infinito cuyos recorridos cambian cada cierto tiempo. Estos dos proyectos parecían interesantes, pero finalmente decidimos, tras reunirme con el equipo Catness a mi llegada el primer día, que mis conocimientos geométricos les resultarían especialmente útiles si los aplicaba al diseño de figuras editables para utilizarlas en el simulador de entornos.

2.2. Objetivos del proyecto a realizar.

Si bien la pretensión final del simulador en el que se decidió que trabajara es recrear completamente una habitación de una vivienda, ofreciendo la posibilidad de editar todo lo relacionado con sus paredes, ventanas y puertas, así como el tamaño, forma y aspecto del mobiliario, completar dicho proyecto requiere una dedicación temporal mucho mayor de la que corresponde a mi estancia en prácticas.

Es por ello que el equipo Catness decidió que lo mejor sería que programara las funciones necesarias para conseguir ofrecer al usuario final varias ediciones de figuras geométricas.

Más ambiciosamente de lo debido, en un principio establecimos que programaría varias isometrías afines y roturas en fragmentos de cubos, esferas y cilindros. No obstante, a la vista de las limitaciones de las herramientas de programación disponibles, se reajustaron los objetivos concluyendo que lo más útil para el equipo sería que, en el momento en el que terminara mi estancia, estuvieran listas las funciones que permitieran el giro, traslación, movimiento de caras y vértices, escalado y extrusión del cubo y, en caso de restar tiempo, generalizar dichas opciones para posibilitar su uso para la edición de otras figuras como cilindros o tetraedros.

2.3. Metodología y definición de tareas.

El equipo Catness utiliza el motor de juego *Unreal Engine* (en adelante, UE) para la programación de sus videojuegos, el cual combina programación en `C++` y Blueprints.

Si bien la programación en `C++` me resultó familiar, pues presenta similitudes con los lenguajes de programación `Java` y `C` que he estudiado durante el grado, Blueprints fue una herramienta novedosa. No obstante, el aprendizaje de su manejo fue rápido, pues UE incorpora Blueprints como una herramienta pensada para quienes se inicien en la creación de videojuegos sin elevados conocimientos de programación y prefieran, consecuentemente, utilizar un entorno gráfico basado en "*click y arrastrar*" para dotar de funcionalidades a su proyecto.

A continuación detallo la definición de las tareas propuestas para su realización durante mi estancia en prácticas:

- Movimiento de vértices, tanto en la dirección de los ejes cartesianos de movimiento como en la dirección de los ejes que representen la rotación de la figura, en caso de haberla. Esto requiere:

    - Detección del cubo que contiene el vértice a editar, distinguiendo su figura del resto del entorno, incluidos otros cubos cercanos y/o en contacto con él. Se estableció que la detección se haría siempre que el cursor se encontrara sobre la figura.

- Detección del vértice a editar, distinguiendo cuál es entre los ocho que presenta el cubo. De nuevo, se estableció que la detección se haría siempre que el cursor se encontrara sobre el vértice. Dado que exigir esto estrictamente implicaría que el cursor estuviera exactamente sobre uno de los píxeles de la pantalla y tanta restricción es poco práctica, se decidió que, cuando el cursor se desplazara cercano a dicho píxel, se iluminaría una esfera translúcida a su alrededor, indicando su detección.

- Selección del vértice a editar. Se estableció que, una vez mostrada la esfera que rodea el entorno del vértice, la selección se haría mediante click izquierdo sobre ella, lo cual la escondería y mostraría en su lugar tres flechas ortogonales con inicio en el vértice y prolongadas sobre las direcciones de los ejes cartesianos.

- Movimiento efectivo del vértice. Una vez seleccionado un vértice y manteniendo pulsado click izquierdo sobre la flecha en cuya dirección desea desplazarse, este se movería a lo largo de los ejes cartesianos. Si se mantiene pulsado click derecho en lugar del izquierdo entonces se movería a lo largo de la dirección seleccionada según la rotación que tenga aplicada la figura.

- Movimiento de las aristas del cubo, tanto en la dirección de los ejes cartesianos de movimiento como en la dirección de los ejes que representen la rotación de la figura, en caso de haberla. Esto requiere la identificación correcta de la arista a mover, su posterior selección y, finalmente, su desplazamiento, abordable utilizando las funciones de movimiento de vértices simultáneamente para los dos vértices situados al inicio y fin de la arista. Dado que se consideró que podría resultar incómodo presentar tantas funcionalidades siempre disponibles, se decidió implementar esta pero dejarla oculta, con posibilidad de asignarla posteriormente a alguna tecla del teclado para poder ejecutarla, pero sin acceso desde el entorno "gráfico" que utiliza el usuario final.

- Movimiento de las caras del cubo en la dirección normal a la cara a editar. Esto requiere:

  - Detección de la cara señalada por el cursor. Mientras que en detección de vértices se mostraba una esfera, se decidió que en este caso gráficamente no se mostraría ningún efecto cuando se detectara la cara. Así, internamente el programa detecta la posición del cursor y la cara debajo de él, pero no varía nada en la figura.

  - Selección de la cara. Tras un click izquierdo sobre la cara ya detectada, se selecciona la misma. Esto provoca que se oculten las esferas visibles de los vértices (en caso de haberlas) y se pase a modo "edición de caras". Este cambio no es perceptible por el usuario, pero internamente bloquea las funciones de detección de vértices, aristas, caras y/o figuras y permite solamente la edición de la cara seleccionada, al tiempo que muestra una flecha que parte del centro de la cara seleccionada y se coloca ortogonal a ella.

  - Movimiento efectivo de la cara. Una vez seleccionada la cara, manteniendo pulsado click izquierdo sobre la flecha y arrastrándola en un sentido del movimiento deseado, la cara se traslada, manteniendo la figura de "cubo" o "prisma rectangular". Hay que tener en cuenta, en este punto, que editar una cara supone modificar cuatro vértices del cubo, lo que afecta a otras cuatro de sus caras que hay que recalcular también.

- Extrusión o, dicho de otro modo, “duplicado” de cubos. Esto consiste en crear un nuevo prisma rectangular con una de sus caras en contacto con la cara sobre la que se hace la extrusión en el cubo de partida. Esto requiere:

    - Detección y selección de la cara. Esto se hace de igual modo que en “movimiento de caras”, activándose tras esto el modo “edición de caras” y mostrándose la flecha ortogonal a la cara a editar.

    - Extrusión efectiva de una cara del cubo. Esto se consigue manteniendo pulsado click derecho sobre la flecha que en el punto anterior he explicado que permitía el movimiento de una cara utilizando el click izquierdo. En cuanto se hace click derecho, se crea un prisma rectangular que tiene una cara pegada a la cara que se está editando en el cubo de partida y, mientras el usuario mantenga pulsado el click derecho y mueva el ratón en un sentido o en otro, desplazará la cara paralela a la que comparte con el otro cubo, es decir, “alargará” o “contraerá” el nuevo prisma rectangular.

- Refactorización del código y corrección de “bugs” o errores. Una vez programado lo anterior, es necesario probar bajo todas las circunstancias imaginables las funciones para corregir todos los errores que se detecten y, cuando esto se logre, hay que refactorizar el código, lo cual consiste en reestructurarlo para compactarlo, consiguiendo funciones lo más atómicas posibles para reutilizarlas y evitar implementar varias veces innecesariamente lo mismo. Además, incluye presentar un código limpio y fácilmente comprensible, lo que conlleva nombres de variables y funciones claros y reducir el código al menor número de líneas posibles.

Cabe notar que, pese a estar hablando de “cubo” porque es la figura de partida, tras la edición de sus caras este puede pasar a ser un prisma rectangular y, en caso de editar sus vértices, cualquier hexaedro.

También es importante mencionar que, para programar todo el código que desarrollé a lo largo de mi estancia en prácticas, el cual puede consultarse en el Anexo A de este documento, partí del código ya implementado por Sébastien Rombauts, disponible en su cuenta de GitHub [1]. De este código aproveché su propuesta de malla procedural “*ProceduralMesh*” y también “*ProceduralMeshComponent*”, su base a partir de la cual desarrollar los componentes de la malla. Las otras dos clases que conforman mi proyecto - programadas por mí - que dejé en Catness tras mis prácticas son “*ProceduralCubeActor*” y “*CustomPlayerController*”, que respectivamente contienen las funciones para la edición del cubo y las que permiten al usuario interactuar con él por medio del ratón.

2.4. Planificación temporal de las tareas.

Dado que el equipo Catness no tenía desarrollado ningún aspecto del simulador de entornos y desconocíamos, tanto ellos como yo, el coste temporal que supondría cada una de las implementaciones porque no sabíamos qué problemas podrían surgir, la planificación inicial fue muy ambiciosa:

[1] URL de la cuenta de GitHub de S. Rombauts: https://github.com/SRombauts/UE4ProceduralMesh

- Primera quincena: asimilación de conceptos básicos necesarios para programar en C++ y familiarización con UE y las particularidades propias de la implementación de videojuegos.

- Segunda quincena: implementación de funciones para habilitar la edición de un cubo (rotación, traslación, escalado, deformación, selección de la apariencia de las caras visibles,...) desde el modo de juego.

- Tercera, cuarta y quinta quincenas: implementación de funciones para habilitar la edición de otros objetos del entorno (rotación, traslación, escalado, deformación, selección de la apariencia de las caras visibles,...) desde el modo de juego.

- Sexta quincena: refactorizar el código y corregir errores de la implementación de la aplicación diseñada.

Si bien es cierto que familiarizarme con la programación en C++ y las particularidades de los videojuegos y aprender a trabajar con UE tan solo me llevó una semana y no dos como se preveía, la implementación interna de las funciones para permitir la edición de las figuras resultó ser lenta y costosa, pues UE permite su traslación, rotación y escalado al programador, pero apenas presenta funciones disponibles para que el usuario final [2] pueda "moldearlas" y fue necesario implementar hasta el más mínimo detalle.

A la vista del ritmo de trabajo viable, la nueva planificación de tareas fue la que sigue:

- Cuatro primeras quincenas: en primer lugar, asimilar conceptos básicos necesarios para programar en C++ y familiarizarme con UE y las particularidades propias de la implementación de videojuegos. Tras esto, implementar las funciones de movimiento de vértices, aristas y caras del cubo, así como su extrusión de caras.

- Quinta quincena: en caso de haber terminado las tareas de las quincenas anteriores y habiendo corregido sus posibles errores, generalizar dichas funciones para permitir su uso en otras figuras poliédricas o cilindros.

- Sexta quincena: refactorizar el código y corregir errores de la implementación de la aplicación diseñada.

[2] UE ofrece un "modo de edición" o "perspectiva de programación" donde el desarollador tiene a su alzance un gran número de funciones preprogramadas y puede diseñar, además, tantas como desee. Una vez completado el diseño - o cuando se desea comprobar los cambios en el mismo -, puede verse el aspecto de lo programado tal como se ofrecería a un usuario final. Esto último se hace cambiando a lo que denominaré, para contextualizar las explicaciones, "modo jugador" o "perspectiva de jugador" y, en él, solamente se ofrecen las funciones que desde el otro modo se hayan implementado y señalado como disponibles para el jugador, por lo que no incluye ninguna de las funciones preprogramadas que tiene UE.

2.5. Recursos del proyecto.

A nivel de recursos materiales, dado el escaso presupuesto del equipo Catness, de reciente creación, y mi disponibilidad de ordenador portátil con suficiente capacidad de procesador y rendimiento gráfico para la actividad a desarrollar, en la empresa únicamente tuvieron que asignarme una mesa de trabajo.

A nivel de software, como nunca antes había hecho nada relacionado con la programación de videojuegos, fue necesario instalar varios componentes en mi ordenador, principalmente *Unreal Engine 4*, *Microsoft Visual Studio* y *Git Hub*. Los dos primeros los utilicé para programar y ejecutar las funciones implementadas y el tercero para crear copias de seguridad periódicas, accesibles para su consulta o para recuperar un código anterior en caso de errores graves en versiones posteriores.

Cabe señalar que *GitHub* permite "compartir" proyectos, por lo que compartí el mío con David Rodríguez y Marco Domingo, lo cual les permitía revisarlo en cualquier momento para estar al día de mis avances durante mi estancia.

2.6. Resultados obtenidos y principales problemas resueltos.

A lo largo de mi estancia en prácticas completé la mayoría de las tareas propuestas en la planificación (en concreto, conseguí que funcionara correctamente todo lo detallado en la sección 2.3. anterior), si bien no hubo tiempo de generalizar las funciones implementadas de edición del cubo a otras figuras poliédricas o cilíndricas. No obstante, antes de marcharme comenté con el equipo Catness mis propuestas para hacer posible esta generalización. Las más importantes fueron:

- Englobar los atributos individuales que representan los vértices en una lista dinámica a la que poder añadir o eliminar cualquiera de ellos en cualquier momento sin tener que reescribir la inicialización de la figura.

- Englobar los vértices de las caras en listas contenidas en una lista mayor, todas ellas dinámicas, para identificar las caras de las figuras rápidamente y hacer más flexibles las modificaciones sobre ellas.

- Adaptar todas las funciones de edición de la figura (movimiento de vértices, aristas y caras, extrusión, etc.) para conseguir un funcionamiento correcto con las listas anteriores y, así, eliminar la dependencia actual del código a una estructura de hexaedro.

En cuanto a problemas resueltos, una vez salvadas las distancias de la programación con UE, destacan:

- UE no trabaja con radianes sino con grados y ángulos de Euler, lo cual desconocía al principio y, hasta que lo detecté, derivó en comportamientos extraños en aquellas funciones que aplicaban

rotaciones o dependían de ellas.

- En una ocasión se produjo *Gimbal Lock* tras la ejecución de una de mis funciones. Sobre este "fenómeno" hablaré en el capítulo tres pero, a grandes rasgos, puede definirse como un "estado" de bloqueo en el que se pierde un grado de libertad en un sistema tridimensional, ocasionando inestabilidades al aplicar rotaciones consecutivas que derivan en comportamientos extraños.

- Al programar para un entorno gráfico con el que el usuario interactúa, muchos detalles pasan desapercibidos en el momento de diseñar las funciones y esto ocasionó errores con frecuencia. Un ejemplo claro que ilustra esto es el hecho de que la primera vez que programé el comportamiento del cubo cuando se hacía click izquierdo sobre un vértice para que aparecieran los ejes sobre los que ejecutar movimiento del mismo, al probarlo en el modo usuario, detecté que si, en lugar de hacer click sobre un vértice, se hacía sobre el cubo o sobre el espacio en el que este se encontraba, aparecían los ejes en el último vértice sobre el que se había desplazado el ratón. D. Rodríguez y M. Domingo me comentaron, durante las primeras semanas, que debía programar pensando siempre en todo lo que haría un usuario utilizando *mal* el programa, consciente o inconscientemente.

Capítulo 3

Matemáticas en los videojuegos.

3.1. Introducción.

Partiendo de los conocimientos en geometría que el grado me ha aportado, iniciaré este capítulo con un repaso de los conceptos conocidos sobre espacios vectoriales y afines euclídeos, donde las distancias pueden medirse por medio de la métrica, así como sobre los morfismos de estructura en estos espacios, es decir, las isometrías afines y lineales.

A continuación, haciendo uso de unas matrices especiales denominadas matrices ortogonales, las cuales verifican que su transpuesta es igual a su inversa, presento una clasificación de las isometrías lineales, incidiendo especialmente en las rotaciones que, como se ve más adelante, permitirán enlazar estos conceptos con el álgebra de los cuaterniones.

A este repaso por las isometrías en espacios afines euclídeos le sigue el estudio de los cuaterniones, partiendo del contexto en el que históricamente se *descubrieron* por primera vez, de la mano de Hamilton. Tras esta breve introducción, estudio el álgebra de los cuaterniones y cómo se utilizan para representar las rotaciones. Por último, expongo una breve reflexión sobre el porqué de utilizar estas cuaternas en la representación de rotaciones frente a otros métodos menos complejos como, por ejemplo, los ángulos de Euler.

Atendiendo a la relación entre lo trabajado durante mi estancia en prácticas y la geometría que presento en este capítulo, cabe señalar que, con ella, se cubren tanto las funciones que programé yo misma en Catness - la mayoría de ellas identificables con isometrías afines -, como el funcionamiento interno del motor gráfico que utilicé durante mis prácticas, en el que muchas de las funciones incluidas en el código fuente se apoyan en el uso de cuaterinones.

3.2. Isometrías en espacios afines euclídeos.

3.2.1. Primeras definiciones.

Empecemos repasando las definiciones de producto escalar, espacio vectorial euclídeo, norma y distancia euclídeas, isometría e isometría lineal, así como una propiedad interesante de estas últimas.

Definición 3.2.1.1. *Sea el espacio vectorial real V, entonces $\langle , \rangle : V \times V \to \mathbb{R}$ es un producto escalar sobre V si:*

i) $\langle \vec{x}, \vec{y} \rangle = \langle \vec{y}, \vec{x} \rangle \quad \forall\, \vec{x}, \vec{y} \in V.$

ii) $\langle \vec{x} + \vec{y}, \vec{z} \rangle = \langle \vec{x}, \vec{z} \rangle + \langle \vec{y}, \vec{z} \rangle \quad \forall\, \vec{x}, \vec{y}, \vec{z} \in V.$

iii) $\langle \lambda\vec{x}, \vec{y} \rangle = \lambda \langle \vec{x}, \vec{y} \rangle \quad \forall\, \vec{x}, \vec{y} \in V, \quad \forall\, \lambda \in \mathbb{R}.$

iv) $\langle \vec{x}, \vec{x} \rangle \geq 0 \quad \forall\, \vec{x}, \vec{y} \in V.$

Se define, bajo estas condiciones, $(V, \langle , \rangle)$ como espacio vectorial euclídeo.

Definición 3.2.1.2. *Se define la norma euclídea o longitud de un vector $\vec{v} \in V$ con V espacio vectorial euclídeo como $||\vec{v}|| = \sqrt{\langle \vec{x}, \vec{x} \rangle} \in \mathbb{R}^+$.*

Definición 3.2.1.3. *Se define la distancia euclídea entre dos vectores $\vec{x}, \vec{y} \in V$ con V espacio vectorial euclídeo como $d(\vec{x}, \vec{y}) = ||\vec{x} - \vec{y}||$.*

Definición 3.2.1.4. *Sean V_1, V_2 dos espacios métricos, entonces se denomina isometría a toda aplicación entre ellos que conserva la distancia, es decir, toda aplicación $\phi : V_1 \to V_2$ / $d_1(\vec{x}, \vec{y}) = d_2(\phi(\vec{x}), \phi(\vec{y}))$ siendo d_1 y d_2 las respectivas distancias en los dos espacios.*

Definición 3.2.1.5. *Sea $f : V \to W$ una aplicación lineal entre espacios vectoriales euclídeos. Diremos que f es una isometría lineal si y solo si $||f(\vec{u})||_W = ||f(\vec{u})||_V \ \ \forall\, \vec{u} \in V$ si y solo si $\langle f(\vec{u}), f(\vec{v}) \rangle = \langle \vec{u}, \vec{v} \rangle$.*

Un primer hecho destacable de las isometrías es que todas ellas conservan la norma, tal como se demuestra en la Proposición que sigue.

Proposición 3.2.1.1. *Sea $f : V \to W$ una isometría lineal, entonces $||f(\vec{u})|| = ||\vec{u}|| \ \forall\, \vec{u} \in V$ si y solo si $\langle f(\vec{u}), f(\vec{v}) \rangle = \langle \vec{u}, \vec{v} \rangle \ \ \forall\, \vec{u}, \vec{v} \in V$.*

Demostración.

Veamos primero que, si $||f(\vec{u})|| = ||\vec{u}|| \; \forall \; \vec{u} \in V$, entonces $\langle f(\vec{u}), f(\vec{v})\rangle = \langle \vec{u}, \vec{v}\rangle \; \forall \; \vec{u}, \vec{v} \in V$.

Si $\vec{u} = \vec{v}$, $\langle f(\vec{u}), f(\vec{u})\rangle = \langle \vec{u}, \vec{u}\rangle$, entonces $||f(\vec{u})||^2 = ||\vec{u}||^2$, luego $||f(\vec{u})|| = ||\vec{u}||$ ya que $||f(\vec{u})|| \geq 0$ y $||\vec{u}|| \geq 0$.

Veamos ahora que $||f(\vec{u})|| = ||\vec{u}|| \; \forall \; \vec{u} \in V$ si $\langle f(\vec{u}), f(\vec{v})\rangle = \langle \vec{u}, \vec{v}\rangle \; \forall \; \vec{u}, \vec{v} \in V$.

En primer lugar, dados $\vec{u}, \vec{v} \in V$, se tiene:

$$\frac{1}{4}\{||\vec{u}+\vec{v}||^2 - ||\vec{u}-\vec{v}||^2\} = \frac{1}{4}\{\langle \vec{u}+\vec{v}, \vec{u}+\vec{v}\rangle - \langle \vec{u}-\vec{v}, \vec{u}-\vec{v}\rangle\} =$$
$$= \frac{1}{4}\{\langle \vec{u}, \vec{u}\rangle + 2\langle \vec{u}, \vec{v}\rangle + \langle \vec{u}, \vec{v}\rangle - (\langle \vec{u}, \vec{u}\rangle - 2\langle \vec{u}, \vec{v}\rangle + \langle \vec{u}, \vec{v}\rangle)\} =$$
$$= \frac{1}{4}\{4\langle \vec{u}, \vec{v}\rangle\} = \langle \vec{u}, \vec{v}\rangle$$

Como esto es cierto $\forall \; \vec{u}, \vec{v} \in V$ y para todo espacio vectorial, teniendo en cuenta que f es lineal, entonces se tiene que:

$$\langle f(\vec{u}), f(\vec{v})\rangle = \frac{1}{4}\{||f(\vec{u})+f(\vec{v})||^2 - ||f(\vec{u})-f(\vec{v})||^2\}\} =$$
$$= \frac{1}{4}\{||f(\vec{u}+\vec{v})||^2 - ||f(\vec{u}-\vec{v})||^2 = \frac{1}{4}\{||\vec{u}+\vec{v}||^2 - ||\vec{u}-\vec{v}||^2\} = \langle \vec{u}, \vec{v}\rangle$$

□

Los objetos geométricos como rectas y planos que no pasan por el origen de coordenadas se estudian mediante la geometría afín, cuyas nociones básicas pasamos a describir.

Definición 3.2.1.6. *Se denomina espacio afín a la terna $(\mathbb{A}, V, \phi)$ formada por un conjunto $\mathbb{A}$, un espacio vectorial V y una aplicación $\phi : \mathbb{A} \times \mathbb{A} \to V$ que cumple:*

i) $\forall \; p \in \mathbb{A}$, $\forall \; \vec{u} \in V$, *existe un único* $q \in \mathbb{A}$ *tal que* $\phi(p, q) = \vec{u}$.

ii) $\phi(p, q) + \phi(q, r) = \phi(p, r) \; \forall \; p, q, r \in \mathbb{A}$.

En particular, cuando el espacio vectorial asociado es el espacio vectorial euclídeo, se tiene el espacio afín euclídeo.

Definición 3.2.1.7. *Sean* $(\mathbb{A}, V, \phi)$, $(\mathbb{A}', V', \phi')$ *dos espacios afines, entonces una aplicación* $f : \mathbb{A} \to \mathbb{A}'$ *es afín si existe una aplicación lineal* $\bar{f} : V \to V'$ *tal que* $\bar{f}(\overrightarrow{pq}) = \overrightarrow{f(p)f(q)}$ $\forall\, p, q \in \mathbb{A}$ *, esto es,* $\forall\, p \in \mathbb{A}$, $\forall\, \vec{u} \in V$ *, se tiene* $f(p + \vec{u}) = f(p) + \bar{f}(\vec{u})$.

Definición 3.2.1.8. *Sea* $\varphi : \mathcal{A} \to \mathcal{A}'$ *una aplicación afín. Entonces* φ *es una isometría afín si y solo si* $d_{\mathcal{A}'}(\varphi(p), \varphi(q)) = d_{\mathcal{A}}(p, q)$ $\forall\, p, q \in \mathcal{A}$.

Ahora bien, ¿qué relación existe entre isometrías lineales e isometrías afines? Lo vemos en la siguiente Proposición.

Proposición 3.2.1.2. $\varphi : \mathcal{A} \to \mathcal{A}'$ *isometría afín si y solo si* $\bar{\varphi} : V \to V'$ *es una isometría lineal.*

Demostración.

Veamos primero que si $\varphi : \mathcal{A} \to \mathcal{A}'$ isometría afín, entonces $\bar{\varphi} : V \to V'$ es una isometría lineal.

Sea $\vec{u} \in V$, ¿ puede afirmarse que $||\varphi(\vec{u})|| = ||\vec{u}||$?

Dado $p \in \mathcal{A}$, sea $q = p + \vec{u}$, entonces $d_{\mathcal{A}'}(\varphi(p), \varphi(q)) = d_{\mathcal{A}}(p, q)$, por tanto se tiene que $||\overline{\varphi(p)\varphi(q)}|| = ||\overline{pq}||$, lo cual implica que $||\varphi(\vec{u})|| = ||\vec{u}||$.

Veamos ahora que $\varphi : \mathcal{A} \to \mathcal{A}'$ isometría afín si $\bar{\varphi} : V \to V'$ es una isometría lineal.

$$d_{\mathcal{A}'}(\varphi(p), \varphi(q)) = ||\overline{\varphi(p)\varphi(q)}|| = ||\bar{\varphi}(\overline{pq})|| = ||\overline{pq}|| = d_{\mathcal{A}}(p, q)$$

Por lo tanto, φ es una isometría afín.

□

A continuación presento dos proposiciones importantes que se necesitan para demostrar el Teorema 3.2.1.1. que las sigue, la primera sobre las isometrías lineales, en la que se demuestra que toda isometría lineal es una aplicación biyectiva cuya inversa es también una isometría lineal, y la segunda sobre isometrías afines, que explica el mismo resultado.

Proposición 3.2.1.3. *Si* $f : V^n \to W^n$ *es una isometría lineal, entonces* f *es biyectiva y, además,* $f^{-1} : W^n \to V^n$ *es una isometría lineal.*

Demostración.

Sea $f : V^n \to W^n$, entonces se tiene que $dim(Im(f)) = n - dim(Ker(F))$.

Si f es inyectiva y, por lo tanto, $Ker(f) = \{\vec{0}\}$, entonces $dim(Im(f)) = n$ y, como $Im(f) \subseteq W^n$, entonces $Im(f) = W$ y, por lo tanto, f es sobreyectiva. Si es a la vez inyectiva y sobreyectiva, f será biyectiva. Comprobemos pues que f es inyectiva.

Si $\vec{v} \in Ker(f)$, entonces $f(\vec{v}) = \vec{0}$, por tanto $||f(\vec{v})|| = ||\vec{0}|| = 0$, y también $||\vec{v}|| = 0$, luego $\vec{v} = 0$, por lo que puede afirmarse que f es inyectiva y, por lo tanto, como también es sobreyectiva, es biyectiva.

Veamos ahora que f^{-1} es una isometría lineal. Para ello, estudiemos si dado $\vec{w} \in W$, puede afirmarse que $||f^{-1}(\vec{w})|| = ||\vec{w}||$.

Por una parte, como f es biyectiva, existe $\vec{v}$ / $f(\vec{v}) = \vec{w}, \vec{v} = f^{-1}(\vec{w})$. Por otra, como f es una isometría, $||f(f^{-1}(\vec{w}))|| = ||f^{-1}(\vec{w})||$.

□

De nuevo, como hemos visto que las isometrías afines guardan relación con las lineales, es natural intentar demostrar si esta propiedad que acabamos de ver sobre las isometrías lineales se cumple también en las isometrías afines.

Proposición 3.2.1.4. *Sea la isometría afín $\varphi : \mathcal{A} \to \mathcal{A}'$, entonces φ es biyectiva y, además, la aplicación $\varphi^{-1} : \mathcal{A}' \to \mathcal{A}$ es una isometría afín.*

Demostración.

Sea φ es una isometría afín. Entonces $\bar{\varphi} : V \to V'$ es una isometría lineal y, por lo tanto $\bar{\varphi}$ es biyectiva, luego necesariamente φ es biyectiva también.

Sea φ una isometría afín. Entonces $\bar{\varphi} : V \to V'$ es una isometría lineal y, por lo tanto, $\bar{\varphi}^{-1} : V' \to V$ es también una isometría lineal.

Además, se tiene que, dados $p', q' \in \mathcal{A}'$, como existe un único $p \in \mathcal{A}$ tal que $\varphi(p) = p'$ y un único $q \in \mathcal{A}$ tal que $\varphi(q) = q'$, entonces:

$$\bar{\varphi}^{-1}(\overline{p'q'}) = \bar{\varphi}^{-1}(\overline{\varphi(p)\varphi(q)}) = \bar{\varphi}^{-1}(\bar{\varphi}(\overline{pq})) = \overline{pq} = \overline{\varphi^{-1}(p')\varphi^{-1}(q')}$$

Luego $\bar{\varphi}^{-1} = \overline{\varphi^{-1}}$ y como, además, $\bar{\varphi}^{-1}$ es una isometría lineal, entonces φ^{-1} es una isometría afín.

□

Teorema 3.2.1.1. *Dado un espacio afín euclídeo $(\mathcal{A}, V, \delta)$, son grupos con la composición los conjuntos definidos como:*

$$O(V) = \{f : V \to V \ / \ f \ \textit{es una isometría lineal}\ \}$$

$$Isom(\mathcal{A}) = \{\varphi : \mathcal{A} \to \mathcal{A} \ / \ \varphi \ \textit{es una isometría afín}\ \}$$

Demostración.

Mediante las dos proposiciones anteriores 3.2.1.3. y 3.2.1.4. se ha demostrado que, dada $f \in O(V)$ y dada $\varphi \in Isom(\mathcal{A})$, ambas poseen elemento inverso en cada uno de los dos casos.

Por otro lado, se tiene que la composición es asociativa en los dos casos y, además, el elemento neutro para $O(V)$ es id_V y para $Isom(\mathcal{A})$ es $id_{\mathcal{A}}$.

Por todo ello, puede afirmarse que ambos conjuntos son grupos con la composición.

□

3.2.2. Isometrías lineales y grupo ortogonal.

En esta sección se presentan algunos resultados interesantes de las isometrías lineales actuando sobre el espacio vectorial euclídeo $(V, \langle , \rangle)$. Empecemos recordando el concepto de matriz ortogonal y de grupo ortogonal.

Definición 3.2.2.1. *Sea una matriz cuadrada $A \in M_{n\times n}(\mathbb{R})$. Diremos que A es una matriz ortogonal si $A^t \cdot A = A \cdot A^t = Id_n$ si y solo si $A^{-1} = A^t$. Además, se define el grupo ortogonal de la forma $O(n) = \{A \in M_{n\times n}(\mathbb{R}) \ / \ A \textit{ es ortogonal}\ \}$.*

Proposición 3.2.2.1. *El determinante de una matriz ortogonal es siempre ± 1.*

Demostración.

Sea la matriz $A \in O(n)$ no singular, i.e. $det(A) \neq 0$, entonces necesariamente $A^t A = Id$ y, por tanto, $det(A^t A) = det(Id) = 1$.

Por propiedades de los determinantes, $det(A^t A) = det(A^t)det(A)$, luego $det(A^t)det(A) = 1$, lo cual implica que $(det(A))^2 = 1$ y, consecuentemente, $det(A) = \pm 1$.

□

Proposición 3.2.2.2. *Sea $O(n)$ el conjunto de todas las matrices ortogonales de dimensión $n \times n$. Entonces se verifica que $O(n)$ tiene estructura de grupo con la multiplicación de matrices.*

Demostración.

Comprobemos que es un grupo.

Sean $A, B \in O(n)$, entonces $AB \in O(n)$, ya que $(AB)^t = B^t A^t = B^{-1} A^{-1} = (AB)^{-1}$.

Además, existe elemento neutro (la identidad), y también inverso, pues por definición de matrices ortogonales se tiene que $A^t = A^{-1}$ y, con ello, si $A \in O(n)$, entonces $A^{-1} \in O(n)$ porque $(A^{-1})^t = (A^t)^{-1} = (A^{-1})^{-1} = A$.

□

Definición 3.2.2.2. *Se define el subgrupo $SO(n) \subseteq O(n)$ de la forma:*

$$SO(n) = \{A \in O(n) \ / \ det(A) = 1\}$$

Se le denomina grupo ortogonal especial, o de las transformaciones ortogonales que preservan la orientación.

Estudiemos un poco más a fondo cómo son los grupos ortogonales especiales $SO(2)$ y $SO(3)$.

Nota 3.2.2.1. *Sea el grupo $SO(2) = \{A \in O(2) \ / \ det(A) = 1\}$, veamos cómo son estas matrices.*

Por un lado, $\forall \ A \in O(2)$, se tiene que $A^{-1} = A^t$, luego $A^t A = Id_2$, es decir, siendo

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

Se cumple que:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} a & c \\ b & d \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{3.1}$$

De aquí se obtiene el sistema de ecuaciones:

$$\begin{cases} a^2 + b^2 = 1 \\ ac + bd = 0 \\ ac + bd = 0 \\ c^2 + d^2 = 1 \end{cases} \tag{3.2}$$

Por un lado, como $a^2 + b^2 = 1$ *, entonces existe un ángulo* $\theta \in [0, 2\pi)$ *tal que:*

$$\begin{cases} a = cos(\theta) \\ b = sen(\theta) \end{cases} \tag{3.3}$$

Por otro lado, como $ac + db = 0$ *, necesariamente* $ac = -bd$ *, luego* $c = -\frac{bd}{a} = -\lambda b$ *,* $d = \lambda a$ *con* $\lambda = \frac{d}{a}$ *. Teniendo en cuenta esto y (3.3), se tiene que si* $A \in O(2)$ *, entonces* $a = cos(\theta)$ *,* $b = sen(\theta)$ *,* $\lambda = \pm 1$ *.*

Entonces, cuando $\lambda = 1$ *, la matriz* A *representa un giro de ángulo* θ *en sentido horario y centro* $(0, 0) \in \mathbb{R}^2$ *y es de la forma:*

$$A = \begin{pmatrix} cos(\theta) & sen(\theta) \\ -sen(\theta) & cos(\theta) \end{pmatrix} \tag{3.4}$$

Y, cuando $\lambda = -1$ *, la matriz* A *representa una simetría ortogonal respecto de la recta que pasa por* $(0, 0) \in \mathbb{R}^2$*, tiene de pendiente* $tg(\frac{\theta}{2})$ *y es de la forma:*

$$A = \begin{pmatrix} cos(\theta) & sen(\theta) \\ sen(\theta) & -cos(\theta) \end{pmatrix} \tag{3.5}$$

Nota 3.2.2.2. *Sea el grupo* $SO(3) = \{A \in O(3) \ / \ det(A) = 1\}$ *, con un desarrollo análogo al de la Nota 3.2.2.1 anterior, pero en este caso con matrices de dimensión* 3×3*, se tiene que las matrices que representan rotaciones de un ángulo* θ *respecto a los tres ejes de coordenadas son de la forma:*

$$R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sen(\theta) \\ 0 & sen(\theta) & cos(\theta) \end{pmatrix}, R_y = \begin{pmatrix} cos(\theta) & 0 & sen(\theta) \\ 0 & 1 & 0 \\ -sen(\theta) & 0 & cos(\theta) \end{pmatrix},$$

$$R_z = \begin{pmatrix} cos(\theta) & -sen(\theta) & 0 \\ sen(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Otra forma de llegar a estas tres matrices es aplicar lo que se verá en la afirmación iii) del Teorema 3.2.3.2., la clave de lo cual yace en identificar cada una de estas tres rotaciones tridimensionales como una rotación bidimensional alrededor de un eje que corresponde a la tercera dimensión. Esto es, R_x se interpreta como una rotación en el plano yz alrededor del eje X, R_y como una rotación en el plano xz alrededor del eje Y, R_z se interpreta como una rotación en el plano xy alrededor del eje Z.

Mediante la siguiente proposición veremos que la descripción de las rotaciones en el espacio tridimensional alrededor de un eje $\vec{w}$ cualquiera, de ángulo θ, puede hacerse mediante una única matriz de rotación $R(\theta, \vec{w})$.

Proposición 3.2.2.3. *Las rotaciones en* $\mathbb{R}^3$ *de ángulo* θ *alrededor de un eje* $\vec{w} = (w_1, w_2, w_3)$ *se generalizan mediante rotación* $R(\theta, \vec{w})$*, cuya representación matricial es:*

$$R(\theta, \hat{w}) = \begin{pmatrix} c + w_1^2(1-c) & w_1w_2(1-c) - sw_3 & w_1w_3(1-c) + sw_2 \\ w_1w_2(1-c) + sw_3 & c + w_2^2(1-c) & w_2w_3(1-c) - sw_1 \\ w_1w_3(1-c) - sw_2 & w_2w_3(1-c) + sw_1 & c + w_3^2(1-c) \end{pmatrix} \tag{3.6}$$

Demostración. Sea el vector unitario que representa el eje de rotación de la forma:

$$\vec{w} = (cos(\alpha)sen(\beta), sen(\alpha)sen(\beta), cos(\beta)) = (w_1, w_2, w_3)$$

Con $\alpha \in [0, 2\pi)$, $\beta \in [0, \pi]$.

Sea el vector columna $\hat{z} = (0, 0, 1)^t$. Entonces es sencillo observar que $\hat{w} = R_z(\alpha)R_y(\beta)\hat{z}$, pues:

$$R_z(\alpha)R_y(\beta)\hat{z} = \begin{pmatrix} cos(\alpha) & -sen(\alpha) & 0 \\ sen(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} cos(\beta) & 0 & sen(\beta) \\ 0 & 1 & 0 \\ -sen(\beta) & 0 & cos(\beta) \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} =$$

$$= \begin{pmatrix} cos(\alpha)cos(\beta) & -sen(\alpha) & cos(\alpha)sen(\beta) \\ sen(\alpha)cos(\beta) & cos(\alpha) & sen(\alpha)sen(\beta) \\ -sen(\beta) & 0 & cos(\beta) \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} =$$

$$= (cos(\alpha)sen(\beta), sen(\alpha)sen(\beta), cos(\beta)) = (w_1, w_2, w_3) = \hat{w}$$

De aquí se obtiene que $\hat{z} = R_y^t(\beta)R_z^t(\alpha)\hat{w}$.

Ahora, para construir la matriz de rotación que gira alrededor de $\hat{w}$ necesitaremos girar alrededor de $\hat{z}$ en un ángulo θ por medio de la matriz $R_z(\theta)$ y ladear $\hat{z}$ en la dirección de $\hat{w}$ (en la que empezó). Veamos esto con detalle.

Primero, girando alrededor de $\hat{z}$ en un ángulo θ por medio de la matriz $R_z(\theta)$ se obtiene:

$$R_z(\theta)R_y^t(\beta)R_z^t(\alpha)\hat{w} = R_z(\theta)\hat{z}$$

A continuación, devolviendo $\hat{z}$ a la dirección de partida de $\hat{w}$ se obtiene:

$$R(\theta, \hat{w}) = R_z(\alpha)R_y(\beta)R_z(\theta)R_y^t(\beta)R_z^t(\alpha)$$

Expandiendo este producto tenemos:

$$R_z(\alpha)R_y(\beta)R_z(\theta) = \begin{pmatrix} cos(\alpha)cos(\beta) & -sen(\alpha) & cos(\alpha)sen(\beta) \\ sen(\alpha)cos(\beta) & cos(\alpha) & sen(\alpha)sen(\beta) \\ -sen(\beta) & 0 & cos(\beta) \end{pmatrix} \cdot \begin{pmatrix} cos(\theta) & -sen(\theta) & 0 \\ sen(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} =$$

$$= \begin{pmatrix} cos(\alpha)cos(\beta)cos(\theta) - sen(\alpha)sen(\theta) & -cos(\alpha)cos(\beta)sen(\theta) - sen(\alpha)cos(\theta) & cos(\alpha)sen(\beta) \\ sen(\alpha)cos(\beta)cos(\theta) + cos(\alpha)sen(\theta) & -sen(\alpha)cos(\beta)sen(\theta) + cos(\alpha)cos(\theta) & sen(\alpha)sen(\beta) \\ -sen(\beta)cos(\theta) & sen(\beta)sen(\theta) & cos(\beta) \end{pmatrix}$$

$$R_z(\alpha)R_y(\beta)R_z(\theta)R_y^t(\beta) =$$

$$= \begin{pmatrix} cos(\alpha)cos(\beta)cos(\theta) - sen(\alpha)sen(\theta) & -cos(\alpha)cos(\beta)sen(\theta) - sen(\alpha)cos(\theta) & cos(\alpha)sen(\beta) \\ sen(\alpha)cos(\beta)cos(\theta) + cos(\alpha)sen(\theta) & -sen(\alpha)cos(\beta)sen(\theta) + cos(\alpha)cos(\theta) & sen(\alpha)sen(\beta) \\ -sen(\beta)cos(\theta) & sen(\beta)sen(\theta) & cos(\beta) \end{pmatrix} \cdot$$

$$\cdot \begin{pmatrix} cos(\beta) & 0 & -sen(\beta) \\ 0 & 1 & 0 \\ sen(\beta) & 0 & cos(\beta) \end{pmatrix} =$$

$$= \begin{pmatrix} cos(\alpha)cos^2(\beta)cos(\theta)- & -cos(\alpha)cos(\beta)sen(\theta)- & -cos(\alpha)cos(\beta)cos(\theta)sen(\beta)+ \\ -sen(\alpha)sen(\theta)cos(\beta)+ & -sen(\alpha)cos(\theta) & +sen(\alpha)sen(\theta)sen(\beta)+ \\ +cos(\alpha)sen^2(\beta) & & +cos(\alpha)sen(\beta)cos(\beta) \\ \\ sen(\alpha)cos^2(\beta)cos(\theta)+ & -sen(\alpha)cos(\beta)sen(\theta)+ & -sen(\alpha)cos(\beta)cos(\theta)sen(\beta)- \\ +cos(\alpha)sen(\theta)cos(\beta)+ & +cos(\alpha)cos(\theta) & -cos(\alpha)sen(\theta)sen(\beta)+ \\ +sen(\alpha)sen^2(\beta) & & +sen(\alpha)sen(\beta)cos(\beta) \\ \\ -sen(\beta)cos(\theta)cos(\beta)+ & sen(\beta)sen(\theta) & sen^2(\beta)cos(\theta) + cos^2(\beta) \\ +cos(\beta)sen(\beta) & & \end{pmatrix}$$

$$R(\theta, \hat{w}) = R_z(\alpha)R_y(\beta)R_z(\theta)R_y^t(\beta)R_z^t(\alpha) =$$

$$\begin{pmatrix} cos(\alpha)cos^2(\beta)cos(\theta)- & -cos(\alpha)cos(\beta)sen(\theta)- & -cos(\alpha)cos(\beta)cos(\theta)sen(\beta)+ \\ -sen(\alpha)sen(\theta)cos(\beta)+ & -sen(\alpha)cos(\theta) & +sen(\alpha)sen(\theta)sen(\beta)+ \\ +cos(\alpha)sen^2(\beta) & & +cos(\alpha)sen(\beta)cos(\beta) \\ \\ sen(\alpha)cos^2(\beta)cos(\theta)+ & -sen(\alpha)cos(\beta)sen(\theta)+ & -sen(\alpha)cos(\beta)cos(\theta)sen(\beta)- \\ +cos(\alpha)sen(\theta)cos(\beta)+ & +cos(\alpha)cos(\theta) & -cos(\alpha)sen(\theta)sen(\beta)+ \\ +sen(\alpha)sen^2(\beta) & & +sen(\alpha)sen(\beta)cos(\beta) \\ \\ -sen(\beta)cos(\theta)cos(\beta)+ & sen(\beta)sen(\theta) & sen^2(\beta)cos(\theta) + cos^2(\beta) \\ +cos(\beta)sen(\beta) & & \end{pmatrix} \cdot$$

$$\cdot \begin{pmatrix} cos(\alpha) & sen(\alpha) & 0 \\ -sen(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix} =$$

$$= \begin{pmatrix} cos^2(\alpha)cos^2(\beta)cos(\theta)- & -cos(\alpha)cos^2(\beta)cos(\theta)sen(\alpha)- & -cos(\alpha)cos(\beta)cos(\theta)sen(\beta)+ \\ -sen(\alpha)sen(\theta)cos(\beta)cos(\alpha)+ & -sen^2(\alpha)sen(\theta)cos(\beta)+ & +sen(\alpha)sen(\theta)sen(\beta)+ \\ +cos^2(\alpha)sen^2(\beta)+ & +cos(\alpha)sen^2(\beta)sen(\alpha)- & +cos(\alpha)sen(\beta)cos(\beta) \\ +cos(\alpha)cos(\beta)sen(\theta)sen(\alpha)+ & -cos^2(\alpha)cos(\beta)sen(\theta)- & \\ +sen^2(\alpha)cos(\theta) & -sen(\alpha)cos(\theta)cos(\alpha) & \\ \\ sen(\alpha)cos^2(\beta)cos(\theta)cos(\alpha)+ & sen^2(\alpha)cos^2(\beta)cos(\theta)+ & -sen(\alpha)cos(\beta)cos(\theta)sen(\beta)- \\ +cos^2(\alpha)sen(\theta)cos(\beta)+ & +cos(\alpha)sen(\theta)cos(\beta)sen(\alpha)+ & -cos(\alpha)sen(\theta)sen(\beta)+ \\ +sen(\alpha)sen^2(\beta)cos(\alpha)+ & +sen^2(\alpha)sen^2(\beta)+ & +sen(\alpha)sen(\beta)cos(\beta) \\ +sen^2(\alpha)cos(\beta)sen(\theta)- & -sen(\alpha)cos(\beta)sen(\theta)cos(\alpha)+ & \\ -cos(\alpha)cos(\theta)sen(\alpha) & +cos^2(\alpha)cos(\theta) & \\ \\ -sen(\beta)cos(\theta)cos(\beta)cos(\alpha)+ & -sen(\beta)cos(\theta)cos(\beta)sen(\alpha)+ & sen^2(\beta)cos(\theta) + cos^2(\beta) \\ +cos(\beta)sen(\beta)cos(\alpha)- & +cos(\beta)sen(\beta)sen(\alpha)+ & \\ -sen(\beta)sen(\theta)sen(\alpha) & +sen(\beta)sen(\theta)cos(\alpha) & \end{pmatrix}$$

Si ahora sustituimos en la matriz $R(\theta, \hat{w})$ obtenida los valores $c = cos(\theta)$, $s = sin(\theta)$, $w_1 = cos(\alpha)sen(\beta)$, $w_2 = sen(\alpha)sen(\beta)$, $w_3 = cos(\beta)$, tenemos:

$$R(\theta, \hat{w}) = R_z(\alpha)R_y(\beta)R_z(\theta)R_y^t(\beta)R_z^t(\alpha) =$$

$$= \begin{pmatrix} cos^2(\alpha)w_3^2c - sen(\alpha)sw_3cos(\alpha) + & cos(\alpha)w_3^2csen(\alpha) - sen^2(\alpha)sw_3 + & -w_1w_3c + w_2s + \\ +w_1^2 + cos(\alpha)w_3ssen(\alpha) + & +w_1w_2 - cos^2(\alpha)w_3s - & +w_1w_3 \\ +sen^2(\alpha)c & -sen(\alpha)ccos(\alpha) & \\ \\ sen^2(\alpha)w_3^2ccos(\alpha) + cos^2(\alpha)sw_3 + & sen^2(\alpha)w_3^2c + cos(\alpha)sw_3sen(\alpha) + & -w_2w_3c - w_1s + \\ +w_1w_2 + sen^2(\alpha)sw_3 - & +w_2^2 - sen(\alpha)w_3scos(\alpha) + & +w_2w_3 \\ -cos(\alpha)csen(\alpha) & +cos^2(\alpha)c & \\ \\ -cw_1w_3 + w_1w_3 - sw_2 & -w_2w_3c + w_2w_3 + sw_1 & sen^2(\beta)c + w_3^2 \end{pmatrix} =$$

$$= \begin{pmatrix} cos^2(\alpha)w_3^2c + w_1^2 + sen^2(\alpha)c & cos(\alpha)w_3^2csen(\alpha) - sw_3 + & w_1w_3(1-c) + sw_2 \\ & +w_1w_2 - sen(\alpha)ccos(\alpha) & \\ \\ sen(\alpha)w_3^2ccos(\alpha) + w_1w_2 + & sen^2(\alpha)w_3^2c + w_2^2 + cos^2(\alpha)c & w_2w_3(1-c) - sw_1 \\ +w_3s - cos(\alpha)csen(\alpha) & & \\ \\ w_1w_3(1-c) - sw_2 & w_2w_3(1-c) + sw_1 & sen^2(\beta)c + w_3^2 \end{pmatrix}$$

Reescribamos ahora cada uno de los términos de esta matriz en función de c, s, w_1, w_2, w_3:

$$cos^2(\alpha)w_3^2c + w_1^2 + sen^2(\alpha)c = w_1^2 + c(w_3^2cos^2(\alpha) + sen^2(\alpha)) =$$
$$= w_1^2 + c(cos^2(\beta)cos^2(\alpha) + 1 - cos^2(\alpha)) = w_1^2 + c(cos^2(\alpha)(1 - sen^2(\beta) - 1) + 1) =$$
$$= w_1^2 + c(1 - cos^2(\alpha)sen^2(\beta)) = w_1^2 + c - cw_1^2 = c + w_1^2(1-c)$$

$$cos(\alpha)w_3^2csen(\alpha) - sw_3 + w_1w_2 - sen(\alpha)ccos(\alpha) = w_1w_2 + sen(\alpha)cos(\alpha)(w_3^2 - 1)c - sw_3 =$$
$$= w_1w_2 + sen(\alpha)cos(\alpha)(1 - sen^2(\beta) - 1)c - sw_3 = w_1w_2 - sen(\alpha)cos(\alpha)sen^2(\beta)c - sw_3 =$$
$$= w_1w_2 - w_1w_2c - sw_3 = w_1w_2(1-c) - sw_3$$

$$sen(\alpha)w_3^2ccos(\alpha) + w_1w_2 + w_3s - cos(\alpha)csen(\alpha) = sen(\alpha)cos(\alpha)(w_3^2 - 1)c + w_1w_2 + w_3s =$$
$$= -sen(\alpha)cos(\alpha)sen^2(\beta)c + w_1w_2 + sw_3 = w_1w_2(1-c) + sw_3$$

$$sen^2(\alpha)w_3^2c + w_2^2 + cos^2(\alpha)c = w_2^2 + c(sen^2(\alpha)w_3^2 + cos^2(\alpha)) =$$
$$= w_2^2 + c(sen^2(\alpha)(1 - sen^2(\beta)) + 1 - sen^2(\alpha)) = w_2^2 + c(-w_2^2 + 1) = c + w_2^2(1-c)$$

$$sen^2(\beta)c + w_3^2 = (1 - cos^2(\beta))c + w_3^2 = c + w_3^2(1-c)$$

Sustituyendo esto en la matriz $R(\theta, \hat{w})$ se obtiene:

$$R(\theta, \hat{w}) = \begin{pmatrix} c + w_1^2(1-c) & w_1w_2(1-c) - sw_3 & w_1w_3(1-c) + sw_2 \\ w_1w_2(1-c) + sw_3 & c + w_2^2(1-c) & w_2w_3(1-c) - sw_1 \\ w_1w_3(1-c) - sw_2 & w_2w_3(1-c) + sw_1 & c + w_3^2(1-c) \end{pmatrix}$$

Ahora es sencillo comprobar que el vector $\hat{w}$ es vector propio de la matriz $R(\theta, \hat{w})$:

$$R(\theta, \hat{w}) \cdot \hat{w} = \begin{pmatrix} c + w_1^2(1-c) & w_1w_2(1-c) - sw_3 & w_1w_3(1-c) + sw_2 \\ w_1w_2(1-c) + sw_3 & c + w_2^2(1-c) & w_2w_3(1-c) - sw_1 \\ w_1w_3(1-c) - sw_2 & w_2w_3(1-c) + sw_1 & c + w_3^2(1-c) \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} =$$

$$= \begin{pmatrix} (c + w_1^2(1-c))w_1 + (w_1w_2(1-c) - sw_3)w_2 + (w_1w_3(1-c) + sw_2)w_3 \\ (w_1w_2(1-c) + sw_3)w_1 + (c + w_2^2(1-c))w_2 + (w_2w_3(1-c) - sw_1)w_3 \\ (w_1w_3(1-c) - sw_2)w_1 + (w_2w_3(1-c) + sw_1)w_2 + (c + w_3^2(1-c))w_3 \end{pmatrix} =$$

$$= \begin{pmatrix} cw_1 + w_1^3(1-c) + w_1w_2^2(1-c) - sw_3w_2 + w_1w_3^2(1-c) + sw_2w_3 \\ w_1^2w_2(1-c) + sw_3w_1 + cw_2 + w_2^3(1-c) + w_2w_3^2(1-c) - sw_1w_3 \\ w_1^2w_3(1-c) - sw_2w_1 + w_2^2w_3(1-c) + sw_1w_2 + cw_3 + w_3^3(1-c) \end{pmatrix} =$$

$$= \begin{pmatrix} w_1(w_1^2 + w_2^2 + w_3^2 + c(1 - w_1^2 - w_2^2 - w_3^2)) \\ w_2(w_1^2 + w_2^2 + w_3^2 + c(1 - w_1^2 - w_2^2 - w_3^2)) \\ w_3(w_1^2 + w_2^2 + w_3^2 + c(1 - w_1^2 - w_2^2 - w_3^2)) \end{pmatrix}$$

Teniendo en cuenta que:

$$w_1^2 + w_2^2 + w_3^2 = cos^2(\alpha)sin^2(\beta) + sin^2(\alpha)sin^2(\beta) + cos^2(\beta) = sin^2(\beta) + cos^2(\beta) = 1$$

Se tiene que:

$$R(\theta, \hat{w}) \cdot \hat{w} = \begin{pmatrix} w_1(1 + c(1-1)) \\ w_2(1 + c(1-1)) \\ w_3(1 + c(1-1)) \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} = \hat{w}$$

□

Pero, ¿cuál es la relación entre las matrices ortogonales y las isometrías lineales? Justo esto es lo que explican los siguientes resultados.

Definición 3.2.2.3. *Sea la base $\{\vec{e}_i\}_{i=1}^n$ del espacio vectorial V, dicha base es una base ortogonal de V si los vectores que la forman son ortogonales entre sí. En particular si, además, son vectores unitarios, se tiene una base ortonormal de V.*

Teorema 3.2.2.1. *Sea* $f : V^n \to V^n$ *una aplicación lineal. Entonces* f *es una isometría lineal si y solo si, si* $\{\vec{e}_i\}_{i=1}^n$ *es una base ortonormal de* V , $\{f(\vec{e}_i)\}_{i=1}^n$ *es una base ortonormal de* V.

Demostración.

Veamos primero que si f es una isometría lineal, entonces dada $\{\vec{e}_i\}_{i=1}^n$ una base ortonormal de V , $\{f(\vec{e}_i)\}_{i=1}^n$ es una base ortonormal de V .

Sea f una aplicación lineal, entonces $f : V \to V$ es un isomorfismo por ser f lineal y biyectiva. Por lo tanto, si $\{\vec{e}_i\}_{i=1}^n$ es una base de V , entonces $\{f(\vec{e}_i)\}_{i=1}^n$ es una base de V . Además, $\langle f(\vec{e}_i), f(\vec{e}_j)\rangle = \langle \vec{e}_i, \vec{e}_j\rangle = \delta_{ij}\ \forall\ i, j \in 1, ..., n$. Luego $\{f(\vec{e}_i)\}_{i=1}^n$ es una base ortonormal.

Veamos ahora la afirmación recíproca.

Sea $\{\vec{e}_i\}_{i=1}^n$ una base ortonormal de V , sabemos que $\{f(\vec{e}_i)\}_{i=1}^n$ es una base ortonormal. Sean también $\vec{u}, \vec{v} \in V$, $\vec{u} = \sum_{i=1}^n u_i\vec{e}_i$, $\vec{v} = \sum_{i=1}^n v_i\vec{e}_j$. Entonces:

$$\langle f(\vec{u}), f(\vec{v})\rangle = \langle f(\sum_{i=1}^n u_i\vec{e}_i), f(\sum_{i=1}^n v_j\vec{e}_j)\rangle = \sum_{i,j=1}^n u_iv_j\langle f(\vec{e}_i), f(\vec{e}_j)\rangle = \sum_{i=1}^n u_iv_j\delta_{ij} = \langle \vec{u}, \vec{v}\rangle$$

Luego f es una isometría lineal.

□

Estudiaremos ahora dos propiedades interesantes sobre las isometrías lineales actuando en el espacio vectorial euclídeo, pero, antes, es necesario recordar la definición del *ángulo entre dos vectores*.

Definición 3.2.2.4. *Sea el espacio afín* $(\mathcal{A}, V, \delta)$ *sobre* $\mathbb{R}$ *y sea la métrica* $g : V \times V \to \mathbb{R}$. *Sean* $\vec{v}, \vec{w} \in V$*, entonces:*

$$|g(\vec{v}, \vec{w})| \leq ||\vec{v}|| \cdot ||\vec{w}||$$

Así, se tiene que:

$$-1 \leq \frac{g(\vec{v}, \vec{w})}{||\vec{v}|| \cdot ||\vec{w}||} \leq 1$$

Por otra parte, la aplicación $f : [0, \pi[\to [-1, 1]$ *tal que* $f(x) = cos(x)$ *con* $x \in [0, \pi[$ *es biyectiva.*

Entonces, dados $\vec{v}, \vec{w} \in V$, existe un único $\alpha \in [0, \pi]$ tal que:

$$cos(\alpha) = \frac{g(\vec{v}, \vec{w})}{||\vec{v}|| \cdot ||\vec{w}||}$$

A α se le llama ángulo entre $\vec{v}$ y $\vec{w}$. Nótese que este ángulo α depende de g.

Proposición 3.2.2.4. *Si $f : V \to V$ es una isometría lineal, entonces:*

i) Conserva normas, distancias y ángulos.

ii) Si $f(\vec{v}) = \lambda\vec{v}$, entonces $\lambda = \pm 1$.

Demostración.

Demostremos i) en primer lugar.

Veamos en primer lugar que se conservan las normas:

$$||f(\vec{u})|| = \sqrt{\langle f(\vec{u}), f(\vec{u})\rangle} = +\sqrt{\langle \vec{u}, \vec{u}\rangle} = ||\vec{u}||$$

Veamos ahora que se conservan las distancias:

$$d_g(\vec{u}, \vec{v}) = ||\vec{v} - \vec{u}|| = +\sqrt{\langle \vec{v} - \vec{u}, \vec{v} - \vec{u}\rangle} = \sqrt{\langle f(\vec{v} - \vec{u}), f(\vec{v} - \vec{u})\rangle} =$$

$$= \sqrt{\langle f(\vec{u}) - f(\vec{v}), f(\vec{u}) - f(\vec{v})\rangle} = ||f(\vec{u}) - f(\vec{v})|| = d_g(f(\vec{u}), f(\vec{v}))$$

Por último, veamos la conservación de ángulos, teniendo en cuenta que se está considerando V espacio afín con V como espacio vectorial asociado.

$$\angle(\vec{u}, \vec{v}) = \alpha \quad \text{tal que} \quad cos(\alpha) = \frac{\langle \vec{u}, \vec{v}\rangle}{||\vec{u}|| \cdot ||\vec{v}||}$$

$$\angle(f(\vec{u}), f(\vec{v})) = \beta \quad \text{tal que} \quad cos(\beta) = \frac{\langle f(\vec{u}), f(\vec{v})\rangle}{||f(\vec{u})|| \cdot ||f(\vec{v})||}$$

Por lo tanto, $cos(\alpha) = cos(\beta)$, con $\alpha, \beta \in (0, \pi]$, luego $\alpha = \beta$.

Demostremos ahora ii).

Sea $\vec{v}$ tal que $f(\vec{v}) = \lambda\vec{v}$, entonces $\langle\vec{v},\vec{v}\rangle = \langle f(\vec{v}), f(\vec{v})\rangle = \langle\lambda\vec{v},\lambda\vec{v}\rangle = \lambda^2\langle\vec{v},\vec{v}\rangle$, por tanto $\lambda^2 = 1$, luego $\lambda = \pm 1$.

□

En el Teorema 3.2.2.1. se ha demostrado que f es una isometría lineal si y solo si la imagen de toda base ortonormal es también una base ortonormal. Veamos ahora qué ocurre con la representación matricial de la imagen por f de una base del espacio vectorial. No obstante, primero incluiré un breve repaso sobre la representación de la métrica con coordenadas.

Definición 3.2.2.5. *Sea el espacio vectorial euclídeo $(V, \langle,\rangle)$ y sea la métrica $\langle,\rangle V \times V \to \mathbb{R}$. Sea, además, la base $\beta = \{\vec{e}_i\}_{i=1}^n$ de V. Entonces:*

i) Si $\vec{x} = \sum_{i=1}^n x_i\vec{e}_i$, $\vec{y} = \sum_{i=1}^n y_i\vec{e}_i$, se tiene:

$$\langle\vec{x},\vec{y}\rangle = \langle\sum_{i=1}^n x_i\vec{e}_i, \sum_{i=1}^n y_i\vec{e}_i\rangle = \sum_{i,j=1}^n x_i y_i\langle\vec{e}_i,\vec{e}_j\rangle =$$

$$= (x_1,...,x_n)\cdot\begin{pmatrix} g_{11} & \cdots & g_{1n} \\ \vdots & & \vdots \\ g_{n1} & \cdots & g_{nn}\end{pmatrix}\cdot\begin{pmatrix} y_1 \\ \vdots \\ y_n\end{pmatrix} = \vec{x}_\beta^t\cdot G_\beta\cdot\vec{y}_\beta$$

Con $g_{ij} = \langle\vec{e}_i,\vec{e}_j\rangle$ $\forall i, \forall j$.

ii) La matriz de la métrica respecto de β es:

$$G_\beta := \begin{pmatrix} g_{11} & \cdots & g_{1n} \\ \vdots & & \vdots \\ g_{n1} & \cdots & g_{nn}\end{pmatrix} = \begin{pmatrix} \langle e_1,e_1\rangle & \cdots & \langle e_1,e_n\rangle \\ \vdots & & \vdots \\ \langle e_n,e_1\rangle & \cdots & \langle e_n,e_n\rangle\end{pmatrix}$$

Nota 3.2.2.3. *La métrica $\langle,\rangle$ no depende de la base β escogida.*

Definición 3.2.2.6. *Sea el espacio afín real $(\mathcal{A}, V, \delta)$ y sea la métrica $\langle,\rangle : V \times V \to \mathbb{R}$. La base $\beta = \{\vec{e}_i\}_{i=1}^n$ es una base ortonormal si y solo si $G_\beta = Id_n$, si y solo si $\langle e_i, e_j\rangle = \delta_{ij}$ $\forall\, i,j \in \{1,...,n\}$. Si $\vec{x},\vec{y} \in V$ son tales que $\vec{x}_\beta = (x_1,...,x_n)$, $\vec{y}_\beta = (y_1,...,y_n)$ y $\beta = \{e_i\}_{i=1}^n$ es una base ortonormal, entonces:*

$$\langle \vec{x}, \vec{y} \rangle = (x_1, ..., x_n) \cdot \begin{pmatrix} 1 & 0 & \cdots & 0 \\ \vdots & & & \vdots \\ 0 & \cdots & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = x_1 y_1 + ... + x_n y_n$$

Nota 3.2.2.4. *Si la base canónica* $\{(1,0,...,0),...,(0,...,0,1)\}$ *es una base ortonormal para la métrica* g *, entonces* $g(\vec{x}.\vec{y}) = x_1 y_1 + ... + x_n y_n$ *es la métrica euclídea (y viceversa).*

Demostración.

Veamos primero que si la base canónica $\{(1,0,...,0),...,(0,...,0,1)\}$ es una base ortonormal para la métrica g, entonces $g(\vec{x}.\vec{y}) = x_1 y_1 + ... + x_n y_n$ es la métrica euclídea.

Supongamos que $[g]_\infty = Id_n$, y sea $\mathcal{C}$ la base canónica.

Entonces, dados $\vec{x}, \vec{y} \in V$, $g(\vec{x}, \vec{y}) = \vec{x}_{\mathcal{C}}^t \cdot Id_n \cdot \vec{y}_{\mathcal{C}}^t = g_u(\vec{x}, \vec{y})$.

Veamos primero que la base canónica $\{(1,0,...,0),...,(0,...,0,1)\}$ es una base ortonormal para la métrica g si $g(\vec{x}.\vec{y}) = x_1 y_1 + ... + x_n y_n$ es la métrica euclídea.

Supongamos que $g = g_u$ la métrica euclídea. Entonces $[g]_{\mathcal{C}} = Id_n$ trivialmente.

□

Nota 3.2.2.5. *Sean las bases* $\beta = \{e_i\}_{i=1}^n$, $\beta' = \{e'_j\}_{i=1}^n$ *de* V *y sea también la matriz* A *del cambio de base de* β' *a* β*, es decir,* $\vec{x}_\beta = A \cdot \vec{x}_{\beta'}$*. Entonces* $G_{\beta'} = A^t \cdot G_\beta \cdot A$*, siendo* $G_{\beta'}$ *y* G_β *las matrices de la métrica en* $G_{\beta'}$ *y* G_β *respectivamente.*

Ahora que ya se ha recordado la representación de una métrica con coordenadas, retomemos las explicaciones teóricas sobre las isometrías lineales y el grupo ortogonal con el siguiente Teorema, que explica la relación entre las isometrías lineales y las matrices ortogonales.

Teorema 3.2.2.2. *Sea* $f : V^n \to V^n$ *una aplicación lineal. Entonces* f *es una isometría lineal si y solo si* $A = [f]_{\{\vec{e}_i\}_{i=1}^n} \in O(n)$ *, i.e. es ortogonal,* $\forall\, \{\vec{e}_i\}_{i=1}^n$ *base ortonormal.*

Demostración.

Sea la base ortonormal $\{\vec{e}_i\}_{i=1}^n$ de V y sean $\vec{x}, \vec{y} \in V$ de la forma $\vec{x} = \sum_{i=1}^n x_i \vec{e}_i$, $\vec{y} = \sum_{i=1}^n y_j \vec{e}_j$.

Por otra parte, sea la aplicación lineal $f(\vec{x}) = \sum_{i=1}^{n} x_i A(\vec{e}_i) = A \cdot \vec{x}$.

Entonces, llamando $B = \{\vec{e}_i\}_{i=1}^{n}$ y $G_B \equiv matriz\ de\ \langle , \rangle\ en\ B$, se tiene:

$$g(f(\vec{x}), f(\vec{y})) \equiv g(A\vec{x}, A\vec{y}) = (A\vec{x})^t \cdot G_B \cdot A\vec{y}$$

Y también:

$$g(\vec{x}, \vec{y}) = \vec{x}^t \cdot G_B \cdot \vec{y}$$

Veamos, en primer lugar, que si f es una isometría lineal, entonces A es ortogonal.

Como B es ortonormal, $G_B = Id$ y, por tanto:

$$g(f(\vec{x}), f(\vec{y})) = G(A\vec{x}, A\vec{y}) = (A\vec{x})^t G_B A\vec{y} = \vec{x}^t A^t Id_n A\vec{y} = \vec{x}^t A^t A\vec{y}$$

Y, además, se tiene que:

$$g(\vec{x}, \vec{y}) = \vec{x}^t G_B \vec{y} = \vec{x}^t \vec{y}$$

Por ser f isometría lineal, $g(f(\vec{x}), f(\vec{y})) = g(\vec{x}, \vec{y})$, entonces se tiene que:

$$\vec{x}^t A^t A\vec{y} = \vec{x}^t \vec{y}\ \ \forall \vec{x}, \vec{y} \in V$$

Esto implica que $A^t A = Id_n$ y, por tanto, puede afirmarse que A es ortogonal.

Veamos ahora que, si $A = [f]_B$ es una matriz ortogonal, siendo B una base ortonormal de V, entonces f es una isometría lineal.

Sea $A = [f]_B$ una matriz ortogonal, B una base ortonormal de V, i.e. $G_B = Id$, entonces, $A^t = A^{-1}$. Por tanto:

$$g(f(\vec{x}), f(\vec{y})) = g(A\vec{x}, A\vec{y}) = (A\vec{x})^t G_B A\vec{y} = \vec{x}^t A^t A\vec{y} = \vec{x}^t \vec{y} = \vec{x}^t G_B \vec{y} = g(\vec{x}, \vec{y})$$

□

Debido a esta relación entre isometrías y matrices ortogonales, las isometrías lineales también se denominan transformaciones ortogonales. Sin embargo hay que destacar que la matriz asociada es ortogonal si y solo si está referida a una base ortonormal.

Nota 3.2.2.6. *Sea $(V, \langle , \rangle)$ un espacio vectorial euclídeo sobre $\mathbb{R}$. Si $O(n, \mathbb{R}) = \{A \in M_{n\times n}(\mathbb{R}) \,/\, A^t = A^{-1}\}$, entonces $O(V) = \{f : V \to V \,/\, f$ es una isometría lineal $\} \simeq O(n, \mathbb{R})$.*

3.2.3. Isometrías en $\mathbb{R}^2$ y $\mathbb{R}^3$: resultados generales y clasificación.

Una característica interesante de las isometrías afines es que, para cualquiera de ellas en $\mathbb{R}^n$, existirá una única traslación y un único isomorfismo ortogonal que verificarán ciertas propiedades como, por ejemplo, que de su composición resultará la propia isometría afín.

Definición 3.2.3.1. *Sea K un cuerpo. Sea $E = E(V)$ un espacio afín definido sobre K. Se dice que $\mathcal{F} \subset E$ es una variedad lineal si existen un subespacio vectorial S de V y $p \in E$ de manera que $\mathcal{F} = p + S = \{p + \vec{v} : \vec{v} \in S\}$.*

Proposición 3.2.3.1. *Sea $\varphi : \mathbb{R}^n \to \mathbb{R}^n$ una aplicación afín. Sea $\mathcal{F}_\varphi$ la variedad lineal tal que $\mathcal{F}_\varphi = \{p \in \mathbb{R}^n \,/\, \varphi(p) = p\} = p_0 + \mathcal{F}_{\bar{\varphi}}$ con $p_0 \in \mathcal{F}_\varphi$. Sea el subconjunto $\mathcal{F}_{\bar{\varphi}} = \{v \in V \,/\, \bar{\varphi}(v) = v\} \subset V$. Entonces se tiene que:*

i) $\mathcal{F}_{\bar{\varphi}} = \{\vec{x} \in \mathbb{R}^n \,/\, \bar{\varphi}(\vec{x}) = \vec{x}\} \hookrightarrow \mathbb{R}^n$.

ii) Si existe $p \in \mathbb{R}^n \,/\, \varphi(p) = p$, entonces $\mathcal{F}_\varphi = p + \mathcal{F}_{\bar{\varphi}}$.

iii) Si $\mathcal{F}_{\bar{\varphi}} = \{\vec{0}\}$ y $\mathcal{F}_\varphi \neq \emptyset$, entonces existe un único p punto fijo de φ.

Teorema 3.2.3.1. *Sea $\varphi : \mathbb{R}^n \to \mathbb{R}^n$ una isometría afín. Entonces existe un único isomorfismo ortogonal, $\psi : \mathbb{R}^n \to \mathbb{R}^n$, y una única traslación, $t_{\vec{v}} : \mathbb{R}^n \to \mathbb{R}^n$, tales que:*

i) $\varphi = t_{\vec{v}} \circ \psi$

ii) $\mathcal{F}_\psi = \{\vec{x} \in \mathbb{R}^n \,/\, \psi(\vec{x}) = \vec{x}\} \neq \emptyset$.

iii) $\mathcal{F}_\psi = p + F$ con $p \in \mathcal{F}_\psi$, $\mathcal{F} = \mathcal{F}_{\bar{\varphi}} = Ker\{\bar{\varphi} - Id_{\mathbb{R}^n}\} = Ker\{\psi - Id_{\mathbb{R}^n}\}$.

Notar que $Ker(\psi - Id_{\mathbb{R}^n}) \equiv$ subespacio director de la variedad lineal de los puntos fijos de φ; también que $\psi = \bar{\varphi}$ es la aplicación lineal asociada y que si φ tiene puntos fijos, entonces $v = 0$ y $\varphi = \bar{\varphi} = \psi$.

Demostración.

Demostremos i) en primer lugar.

Supongamos que $\varphi : \mathbb{R}^n \to \mathbb{R}^n \ / \ \varphi(\vec{0}) = \vec{0}$ es una isometría afín.

Entonces se tiene que:

$$\bar{\varphi}(\vec{x}) = \bar{\varphi}(\vec{x} - \vec{0}) = \bar{\varphi}(\overrightarrow{\vec{0}\vec{x}}) = \overrightarrow{\varphi(\vec{0})\varphi(\vec{x})} = \overrightarrow{\vec{0}\varphi(\vec{x})} = \varphi(\vec{x}) - \vec{0} = \varphi(\vec{x}) \tag{3.7}$$

Acabamos de ver, pues, que $\varphi = \bar{\varphi}$, su aplicación lineal asociada. Por tanto, φ es lineal y, además, $\varphi = \bar{\varphi} = t_{\vec{0}} \circ \bar{\varphi}$.

Por otro lado, supongamos que $\varphi : \mathbb{R}^n \to \mathbb{R}^n \ / \ \varphi(\vec{0}) \neq \vec{0}$ es una isometría afín y sea la aplicación $\psi : \mathbb{R}^n \to \mathbb{R}^n \ / \ \psi(\vec{x}) := \varphi(\vec{x}) - \varphi(\vec{0})$. Entonces ψ es afín con la aplicación lineal asociada $\bar{\varphi} : \mathbb{R}^n \to \mathbb{R}^n$, es decir, $\bar{\psi} = \bar{\varphi}$, porque, como $\bar{\varphi}$ es la lineal asociada a φ, entonces $\bar{\varphi}(\overrightarrow{\vec{o}\vec{x}}) = \overrightarrow{\varphi(\vec{0})\varphi(\vec{x})}$, luego:

$$\bar{\varphi}(\overrightarrow{\vec{o}\vec{x}}) = \overrightarrow{\varphi(\vec{0})\varphi(\vec{x})} = \varphi(\vec{x}) - \varphi(\vec{0}) = \psi(\vec{x}) = \psi(\vec{x}) - \psi(\vec{0}) = \overrightarrow{\psi(\vec{0})\psi(\vec{x})} \tag{3.8}$$

También es cierto que, como φ es una isometría, entonces $\bar{\varphi}$ es una isometría lineal (isomorfismo ortogonal) y, por lo tanto, ψ es una isometría afín.

Por lo que se ha visto en (3.7), como $\psi(\vec{0}) = \vec{0}$, entonces $\psi = \bar{\psi} = \bar{\varphi}$ es un isomorfismo ortogonal.

Así, puede concluirse que, por la definición de ψ, $\psi(\vec{x}) = \varphi(\vec{x}) - \varphi(\vec{0})$, tenemos:

$$\varphi(\vec{x}) = \psi(\vec{x}) + \varphi(\vec{0}) = t_{\varphi(\vec{0})} \circ \psi(\vec{x}) \tag{3.9}$$

Además, si x_0 es un punto dijo de φ, entonces $x_0 = \varphi(x_0) = \psi(x_0) + \varphi(0)$.

Cabe señalar que $t_{\vec{v}} = t_{\varphi(\vec{0})}$ es única y $\psi(\vec{x}) = \varphi(\vec{x}) - \varphi(\vec{0})$ también. Veamos esto.

Si existe $\psi_2 \ / \ \varphi(\vec{x}) = t_{\varphi(\vec{0})} \circ \psi_2(\vec{x}) = \psi_2(\vec{x}) + \varphi(\vec{x})$ y $\varphi(\vec{x}) = \psi(\vec{x}) + \varphi(\vec{0})$, entonces $\psi_2 = \psi$ necesariamente.

Si existe $\vec{v} \ / \ \varphi(\vec{x}) = t_{\vec{v}} \circ \psi(\vec{x}) = \psi(\vec{x}) + \vec{0}$ y $\varphi(\vec{x}) = \psi(\vec{x}) + \varphi(\vec{0})$, entonces $\psi(\vec{0}) + \vec{v} = \psi(\vec{0}) + \varphi(\vec{0})$, por tanto $\vec{v} = \varphi(\vec{0})$.

Hemos demostrado que, tanto si $\varphi(\vec{0}) = \vec{0}$ como si $\varphi(\vec{0}) \neq \vec{0}$, existen un único $\vec{v} = \varphi(\vec{0})$ y un único isomorfismo ortogonal ψ tales que $\varphi = t_{\vec{v}} \circ \psi$. Además, $\bar{\varphi} = \psi$.

Demostremos ahora ii).

Como $\psi(\vec{0}) = \varphi(\vec{0}) - \varphi(\vec{0}) = \vec{0}$, entonces:

$$\mathcal{F}_\psi = \{\vec{x} \in \mathbb{R}^n \ / \ \psi(\vec{x}) = \vec{x}\} \neq \emptyset \tag{3.10}$$

Por último, demostremos iii).

Sea la variedad lineal $\mathcal{F}_\psi$ y sea $\vec{p} \in \mathcal{F}_\psi$. Veamos que $\mathcal{F}_\psi = \vec{p} + Ker(\psi - Id_{\mathbb{R}^n})$.

Veamos primero que $\mathcal{F}_\psi \subseteq \vec{p} + Ker(\psi - Id_{\mathbb{R}^n})$.

Esto puede verse teniendo en cuenta que, dado $p \in \mathcal{F}_\psi$ y $\vec{x} \in \mathcal{F}_\psi$, entonces $\psi(\vec{x} - \vec{p}) = \psi(\vec{x}) - \psi(\vec{p}) = \vec{x} - \vec{p}$, luego $\vec{x} = \vec{p} + \vec{x} - \vec{p} \in p + Ker(\psi - Id)$.

Veamos ahora que $\mathcal{F}_\psi \supseteq \vec{p} + Ker(\psi - Id_{\mathbb{R}^n})$, con $p \in \mathcal{F}_\psi$.

Sea $\vec{x} \in \vec{p} + Ker(\psi - Id_{\mathbb{R}^n})$, entonces $\vec{x} = \vec{p} + \vec{y}$ con $\vec{y} \ / \ \psi(\vec{y}) = \vec{y}$. Así, teniendo en cuenta que ψ es lineal, se tiene que:

$$\psi(\vec{x}) = \psi(\vec{p} + \vec{y}) = \psi(\vec{p}) + \psi(\vec{y}) = \vec{p} + \vec{y} = \vec{x}$$

Luego $\vec{x} \in \mathcal{F}_\psi$.

Además, como $\bar{\psi} = \bar{\varphi} = \psi$, entonces $Ker(\psi - Id_{\mathbb{R}^n}) = Ker(\bar{\varphi} - Id_{\mathbb{R}^n})$.

□

El Teorema 3.2.3.2. que sigue, clasifica las isometrías lineales. En virtud del Teorema 3.2.3.1., esta clasificación se traslada a las isometrías afines salvo traslación.

Teorema 3.2.3.2. *Sea $T : \mathbb{R}^3 \to \mathbb{R}^3$ una isometría afín y sea $\bar{T} : \mathbb{R}^3 \to \mathbb{R}^3$ la isometría lineal asociada; $\bar{T}$ es un isomorfismo ortogonal. Entonces:*

i) Si $dim(Ker(\bar{T} - Id_{\mathbb{R}^3})) = 3$, entonces la isometría $\bar{T}$ es la identidad, es decir, $\bar{T} = Id_{\mathbb{R}^3} = T$.

ii) Si $dim(Ker(\bar{T} - Id_{\mathbb{R}^3})) = 2$, entonces la isometría $\bar{T}$ es simétrica ortogonal lineal respecto de $\Pi = Ker(\bar{T} - Id_{\mathbb{R}^3})$, lo cual implica que T es una simetría ortogonal.

iii) Si $dim(Ker(\bar{T} - Id_{\mathbb{R}^3})) = 1$, entonces la isometría $\bar{T}$ es una rotación (giro) alrededor del eje $Ker(\bar{T} - Id_{\mathbb{R}^3})$.

iv) Si $dim(Ker(\bar{T} - Id_{\mathbb{R}^3})) = 0$, entonces la isometría $\bar{T}$ es una simetría rotacional; una rotación (giro) alrededor de $Ker(\bar{T} - Id_{\mathbb{R}^3})$ compuesta con una simetría ortogonal respecto de $Ker(\bar{T} - Id_{\mathbb{R}^3})^{\perp}$.

Demostración.

i) es obvio. Veamos ii)

Sea $\Pi = Ker(\bar{T} - Id_{\mathbb{R}^3}) = \langle\{\vec{v}_1, \vec{v}_2\}\rangle$ una base ortonormal. Sea $\vec{v}_3$ unitario, $\vec{v}_3 \in \Pi^{\perp}$. Entonces $\{\vec{v}_1, \vec{v}_2, \vec{v}_3\}$ una base ortonormal de $\mathbb{R}^3$.

Como $\bar{T}$ es una isometría ortogonal, entonces $\{T(\vec{v}_i)\}_{i=1}^3$ es una base ortonormal de $\mathbb{R}^3$. Así, se tiene que, como $\bar{T}(\vec{v}_1) = \vec{v}_1$, $\bar{T}(\vec{v}_2) = \vec{v}_2$:

$$\bar{T}(\vec{v}_3) = \langle \bar{T}(\vec{v}_3), \vec{v}_1\rangle \vec{v}_1 + \langle \bar{T}(\vec{v}_3), \vec{v}_2\rangle \vec{v}_2 + \langle \bar{T}(\vec{v}_3), \vec{v}_3\rangle \vec{v}_3 =$$

$$= \langle \bar{T}(\vec{v}_3), \bar{T}(\vec{v}_1)\rangle \vec{v}_1 + \langle \bar{T}(\vec{v}_3), \bar{T}(\vec{v}_2)\rangle \vec{v}_2 + \langle \bar{T}(\vec{v}_3), \vec{v}_3\rangle \vec{v}_3 =$$

$$= \langle \vec{v}_3, \vec{v}_1\rangle \vec{v}_1 + \langle \vec{v}_3, \vec{v}_2\rangle \vec{v}_2 + \langle \bar{T}(\vec{v}_3), \vec{v}_3\rangle \vec{v}_3 = 0 + 0 + \langle \bar{T}(\vec{v}_3), \vec{v}_3\rangle \vec{v}_3 = \lambda \vec{v}_3$$

Como $\bar{T}$ es ortogonal, y $\bar{T}(\vec{v}_3) = \lambda \vec{v}_3$, entonces $\lambda = \pm 1$. Pero $T(\vec{v}_3) \neq \vec{v}_3$ porque si $\bar{T}(\vec{v}_3) = \vec{v}_3$, entonces $\vec{v}_3 \in Ker(\bar{T} - Id_{\mathbb{R}^3})$, y, por tanto, $\vec{v}_3 = \vec{0}$, pero esto es una contradicción, por lo tanto necesariamente $T(\vec{v}_3) = -\vec{v}_3$.

Entonces se tiene que:

$$[\bar{T}]_{\{\vec{v}_1, \vec{v}_2, \vec{v}_3\}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{3.11}$$

Si $\vec{x} \in \mathbb{R}^3$, siendo $\vec{x} = \sum_{i=1}^n x_i \vec{v}_i$, entonces:

$$\bar{T}(\vec{x}) = \bar{T}(\sum_{i=1}^{3} x_i\vec{v}_i) = x_1\vec{v}_1 + x_2\vec{v}_2 + x_3\vec{v}_3 - 2x_3\vec{v}_3 = \vec{x} - 2\frac{\langle\sum_{i=1}^{n} x_i\vec{v}_i, \vec{v}_3\rangle}{||\vec{v}_3||^2}\vec{v}_3 \tag{3.12}$$

Se puede demostrar que:

$$\bar{T}(\vec{x}) = \vec{x} - 2\frac{\langle\sum_{i=1}^{n} x_i\vec{v}_i, \vec{v}_3\rangle}{||\vec{v}_3||^2}\vec{v}_3 = S_{\Pi}(\vec{x})$$

Demostremos ahora iii)

Sea $Ker(\bar{T} - Id_{\mathbb{R}^3}) = \langle\{\vec{v}_1\}\rangle$ con $\vec{v}_1$ unitario y sea $\Pi = \langle\{\vec{v}_1\}\rangle^{\perp}$ y $\{\vec{v}_2, \vec{v}_3\}$ una base ortonormal de Π.

Entonces la base ortonormal $\{\vec{v}_1, \vec{v}_2, \vec{v}_3\}$ de $\mathbb{R}^3$ es $\{\bar{T}(\vec{v}_1), \bar{T}(\vec{v}_2), \bar{T}(\vec{v}_3)\} = \{\vec{v}_1, \bar{T}(\vec{v}_2), \bar{T}(\vec{v}_3)\}$. Como es una base ortonormal de $\mathbb{R}^3$ entonces $\bar{T}(\vec{v}_2)\perp\vec{v}_1$ y $\bar{T}(\vec{v}_3)\perp\vec{v}_1$, por lo tanto $\bar{T}(\vec{v}_2), \bar{T}(\vec{v}_3) \in \langle\vec{v}_1\rangle^{\perp} = \Pi$. Así, $\bar{T}|_{\Pi} : \Pi \to \Pi$ es una transformación ortogonal, porque $\{T(v_i)\}_{i=2}^{3}$ es una base ortonormal.

Si $\vec{v} \in \Pi = \langle\vec{v}_1\rangle^{\perp}$ tal que $\bar{T}(\vec{v}) = \vec{v}$ y, por tanto, $\vec{v} \in \langle\vec{v}_1\rangle$, entonces $\vec{v} = \vec{0}$.

Aplicando el Teorema 3.2.3.2. de esta sección, puede afirmarse que $\bar{T}|_{\Pi}$ es una transformación ortogonal sin vectores fijos (excepto el origen) y, por lo tanto, un giro, ya que, si no, sería una simetría axial y tendría vectores fijos no nulos. Este giro es, concretamente:

$$[\bar{T}]_{\Pi} = \begin{bmatrix} cos(\theta) & -sen(\theta) \\ sen(\theta) & cos(\theta) \end{bmatrix} \tag{3.13}$$

Con lo que la matriz de T en la base $\{\vec{v}_1, \vec{v}_2, \vec{v}_3\}$ viene dada por:

$$[\bar{T}] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sen(\theta) \\ 0 & sen(\theta) & cos(\theta) \end{bmatrix} \tag{3.14}$$

En esta segunda matriz las columnas representan, respectivamente, $\bar{T}(\vec{v}_1), \bar{T}(\vec{v}_2)$ y $\bar{T}(\vec{v}_3)$.

Por último, demostremos iv)

Sea $dim(Ker(\bar{T} - Id_{\mathbb{R}^3})) = 0$. Entonces $\bar{T}(\vec{v}) = \vec{v}$ si y solo si $\vec{v} = \vec{0}$, luego $\lambda = 1$ no es un valor propio de $\bar{T}$. Sin embargo, $\bar{T}$ posee valores propios; estudiemos esto.

λ será un valor propio de $\bar{T}$ si y solo si $det[\bar{T} - \lambda Id_3] = 0$. Veamos esto con detalle.

Por un lado, como $rg(\bar{T} - \lambda Id_3) \leq 2$, entonces:

$$Im(\bar{T} - \lambda Id_3) + dim(Ker(\bar{T} - \lambda Id_3)) = 3 \tag{3.15}$$

Por otro lado se tiene que:

$$Im(\bar{T} - \lambda Id_3) + dim(Ker(\bar{T} - \lambda Id_3)) \leq 2 + dim(Ker(\bar{T} - \lambda Id_3)) \tag{3.16}$$

Por lo tanto, teniendo en cuenta (3.15) y (3.16), puede afirmarse que:

$$dim(Ker(\bar{T} - \lambda Id_3)) \geq 1 \tag{3.17}$$

Por lo tanto, existe $\vec{v} \neq \vec{0}$ tal que $\bar{T}(\vec{v}) - \lambda\vec{v} = 0$, luego λ es valor propio de $\bar{T}$.

Por otra parte, $det[\bar{T} - \lambda Id_3] = 0$ si y solo si existe $\lambda \ / \ a\lambda^3 + b\lambda^2 + c\lambda + d = 0$ para ciertos valores a, b, c, d y esto es cierto porque el polinomio es continuo y, además:

$$\begin{cases} lim_{\lambda\to\infty}\ a\lambda^3 + b\lambda^2 + c\lambda + d = \infty \\ lim_{\lambda\to-\infty}\ a\lambda^3 + b\lambda^2 + c\lambda + d = -\infty \end{cases} \tag{3.18}$$

Como $\bar{T}$ es una isometría ortogonal, las raíces valores propios de T pueden ser $\lambda = +1$ o $\lambda = -1$, pero $\lambda = +1$ no lo es, y el valor propio es $\lambda = -1$.

Sea $\vec{v}_1$ unitario tal que $\bar{T}(\vec{v}_1) = -\vec{v}_1$. Sea $\Pi = \langle\vec{v}_1\rangle^{\perp}$ y sean $\vec{v}_2, \vec{v}_3$ vectores de la base ortonormal de Π, entonces $\{\vec{v}_1, \vec{v}_2, \vec{v}_3\}$ es una base ortonormal de $\mathbb{R}^3$.

Como, por un lado, se tiene que $\langle\bar{T}(\vec{v}_2), \vec{v}_1\rangle = \langle\bar{T}(\vec{v}_2), -\bar{T}(\vec{v}_1)\rangle = -\langle\vec{v}_2, \vec{v}_1\rangle = 0$ y, por otro lado, que $\langle\bar{T}(\vec{v}_3), \vec{v}_1\rangle = \langle\bar{T}(\vec{v}_3), -\bar{T}(\vec{v}_1)\rangle = -\langle\vec{v}_3, \vec{v}_1\rangle = 0$, entonces $\bar{T}(\vec{v}_2), \bar{T}(\vec{v}_3) \in \Pi$.

Así, $\bar{T}|_{\Pi} : \Pi \to \Pi$ es una aplicación lineal, ortogonal y sin puntos fijos: si $\vec{w} \neq \vec{0}$ con $w \in \Pi$ verificase $\bar{T}(\vec{w}) = \vec{w}$, entonces $\vec{w} \in Ker(\bar{T} - Id_{\mathbb{R}^3}) = \{\vec{0}\}$, lo cual es contradictorio. Entonces $T|_{\Pi}$ es un giro y

existe $\theta \in [0, 2\pi)$ tal que:

$$[\bar{T}|_{\Pi}] = \begin{bmatrix} cos(\theta) & -sen(\theta) \\ sen(\theta) & cos(\theta) \end{bmatrix} \tag{3.19}$$

Luego:

$$[\bar{T}] = \begin{bmatrix} -1 & 0 & 0 \\ 0 & cos(\theta) & -sen(\theta) \\ 0 & sen(\theta) & cos(\theta) \end{bmatrix} \tag{3.20}$$

□

Veamos, para terminar esta sección, la relación entre las matrices del grupo ortogonal y las rotaciones.

Corolario 3.2.3.1. *Sea $A \in SO(3)$ una matriz de dimensión 3×3. Entonces, existen una matriz $P \in SO(3)$ y un ángulo $\theta \in [0, 2\pi)$ tales que $P^{-1}AP = R_\theta$, donde R_θ es una matriz cuya transformación representa una rotación de ángulo θ.*

Demostración.

Veamos primero que $det(A - I_3) = 0$:

$$det(A - I_3) = det(A - AA^T) = det(A(I_3 - A)^T) = (det(A))det((I_3 - A)^T) = -det(A - I_3)$$

Ahora, como $A - I_3$ es singular, entonces existe un vector columna distinto de cero, w, tal que $Aw = w$, y es justamente el vector propio que determinará el eje de rotación. Veámoslo con detalle.

Como $det(A - I_3) = 0$, entonces $rg(A - I_3) \leq 2$ y, por lo tanto, $dim(Im(A - I_3)) \leq 2$.

Pero sabemos que $3 = dim(Ker(A - I_3)) + dim(Im(A - I_3)) \leq dim(Ker(A - I_3) + 2$, luego $1 \leq dim(Ker(A - I_3))$ y, consecuentemente, existe $\vec{w} \in Ker(A - I_3)$, $\vec{w} \neq \vec{0}$ tal que $(A - I_3)\vec{w} = \vec{0}$, y, por lo tanto, existe $\vec{w} \in Ker(A - I_3)$, $\vec{w} \neq \vec{0}$ tal que $A\vec{w} = \vec{w}$.

Sea $\vec{w_3} = \frac{\vec{w}}{||\vec{w}||}$ y sea $\Pi = \langle \vec{w_3} \rangle^{\perp}$. Sea también la base ortonormal $\{\vec{w}_1, \vec{w}_2\}$ de Π.

Como $A\vec{w}_1, A\vec{w}_2 \in \Pi = \langle \vec{w_3} \rangle$, entonces $A|_{\Pi} : \Pi \to \Pi$ es una transformación ortogonal porque $\{A|_{\pi}(w_i)\}$ es una base ortonormal de Π. Así, como $A|_{\Pi} \in O(2)$, se tiene que $det(A|_{\Pi}) = \pm 1$, pero

como además $A \in SO(3)$, entonces $det(A) = 1$. En consecuencia, como $det(A) = (-1)^6 det(A|_\pi)$, necesariamente $det(A|_\pi) = 1$, y así $A|_\Pi \in SO(2)$, por lo que $A|_\Pi$ es un giro cuya matriz es:

$$\begin{pmatrix} cos(\theta) & -sen(\theta) \\ sen(\theta) & cos(\theta) \end{pmatrix}$$

□

Justo este resultado es el que se obtuvo en la afirmación iii) del Teorema 3.2.3.2.

Ahora que hemos recordado los conceptos más importantes sobre los espacios vectoriales y afines euclídeos y las isometrías sobre ellos, es momento de presentar a los cuaterniones.

3.3. Los cuaterniones y su importancia en gráficos por ordenador.

3.3.1. El *nacimiento* de los cuaterniones y su creciente popularidad.

> *"Mañana será el decimoquinto cumpleaños de los cuaterniones. Surgieron a la vida, o a la luz, ya crecidos, el 16 de octubre de 1843, cuando me encontraba caminando con la Sra. Hamilton hacia Dublín, y llegamos al Puente de Broughman. Es decir, entonces y ahí, cerré el circuito galvánico del pensamiento y las chispas que cayeron fueron las ecuaciones fundamentales entre i, j, k; exactamente como las he usado desde entonces. Saqué, en ese momento, una libreta de bolsillo, que todavía existe, e hice una anotación, sobre la cual, en ese mismo preciso momento, sentí que posiblemente sería valioso el extender mi labor por al menos los diez (o podían ser quince) años por venir. Es justo decir que esto sucedía porque sentí, en ese momento, que un problema había sido resuelto, un deseo intelectual aliviado, deseo que me había perseguido por lo menos los quince años anteriores. No pude resistir el impulso de coger mi navaja y grabar en una piedra del Puente Brougham la fórmula fundamental con los símbolos i, j, k:*
>
> $$i^2 = j^2 = k^2 = ijk = -1$$
>
> *que contenían la solución del Problema, que desde entonces sobrevive como inscripción."*

Es esto lo que quince años después del 16 de octubre de 1843, Hamilton escribía en una carta a uno de sus hijos. Su interés por estudiar lo que acabaría motivando la creación de los cuaterniones surgió cuando leyó, en 1828, por recomendación de Graves, el recién publicado "*Tratado sobre la representación geométrica de las raíces cuadradas de cantidades negativas*", del matemático John Warren. En este trabajo se describía la representación de los números complejos de Argand. Hamilton se sintió atraído por la idea de que, si se consideraban los números complejos como vectores del plano, entonces las operaciones aritméticas elementales adquirían una interpretación geométrica natural, de modo que la suma

compleja resultaba equivalente a la suma vectorial, y la multiplicación compleja equivalente al producto de un vector por un escalar más la rotación de dicho vector. Fue entonces cuando en Hamilton surgió la idea de intentar generalizar los números complejos con el fin de representar rotaciones y movimientos de vectores en el espacio tridimensional. Si esta generalización funcionaba, sin lugar a dudas los números complejos resultarían una herramienta potentísima para la formulación de las leyes básicas de la física con el fin de describir el movimiento de cuerpos rígidos en el espacio.

Desconocía el "*atecnológico*" Hamilton del siglo diecinueve la importancia que en la actualidad tendría su "*creación*", utilizada en los siglos posteriores para describir dinámicas en tres dimensiones. Por ejemplo, destaca especialmente el uso de los cuaterniones en el software de vuelo del Space Shuttle [1], que los utilizaba para el control de navegación y vuelo, en software actual con utilidades similares, o en los gráficos por computadora.

Entre las ventajas que motivan el uso de los cuaterniones destacan las relativas al código en el que se utilizan, pues consiguen hacerlo más compacto y aumentar la velocidad de cómputo eliminando singularidades en los cálculos. Como se explicará en la sección 3.3.4., comparados con los ángulos de Euler, ofrecen una representación de la orientación de un objeto tridimensional no singular, más compacta y más rápida que las matrices a nivel de cálculos computacionales.

Cabe señalar que los cuaterniones también cobraron importancia por el uso que se les dio en la representación de fractales, tanto para las técnicas de modelaje de computación gráfica como para la formación de algoritmos para abstraer y codificar los detalles de un modelo, evitando así almacenar explícitamente muchas primitivas de bajo nivel para lograr el modelo. Un claro ejemplo del uso de los cuaterniones en este ámbito es el que permitía el almacenamiento de las imágenes de la enciclopedia Encarta [2] en un espacio reducido.

Por su parte, los motores de programación de videojuegos como Unity o Unreal Engine (el que utilicé durante mis prácticas), hacen uso de los cuaterniones también para las representaciones gráficas.

Si bien la anterior sección de este capítulo estuvo dedicada al estudio de las isometrías en los espacios afines euclídeos, en esta sección profundizaremos en el estudio de los cuaterniones.

Muchas de las aplicaciones de los números complejos a la geometría pueden generalizarse utilizando cuaterniones, en los que la parte imaginaria la forma un vector de $\mathbb{R}^3$. Dado que el campo de aplicación de los cuaterniones es muy amplio, nos centraremos en el estudio de su relación con las isometrías en espacios afines euclídeos y, concretamente, en su utilidad a la hora de representar rotaciones en el espacio de tres dimensiones.

[1] Transbordador espacial gestionado por la Administración Aeronáutica y Espacial Nacional de Estados Unidos (NASA) cuyo primer vuelo de prueba orbital se produjo en 1981.

[2] Microsoft Encarta fue una enciclopedia multimedia digital publicada por Microsoft Corporation desde 1993 hasta 2009.

En la sección 3.3.2. que sigue se presentan las definiciones tanto de los cuaterniones como de algunas de sus operaciones, así como algunas de sus propiedades básicas. A continuación, presento otra sección donde se estudiarán las rotaciones representadas con cuaterniones; con los cuaterniones ya suficientemente trabajados, podrá entenderse el *puente* entre ellos y las isometrías en espacios afines que se han presentado en la sección 3.2. Finalmente, presento una reflexión sobre porqué utilizar los cuaterniones pese a su patente complejidad frente a otras representaciones de rotaciones como los ángulos de Euler.

3.3.2. El álgebra de los cuaterniones.

Los cuaterniones pueden interpretarse como vectores de cuatro componentes y, por tanto, como elementos que pertenecen a un espacio de cuatro dimensiones. La presente sección está dedicada al estudio del álgebra de los cuaterniones partiendo de la definición de cuaternión y explicando a partir de ella algunas de sus propiedades y operaciones.

Definición 3.3.2.1. *Se definen los cuaterniones como elementos de la forma:*

$$q = a + bi + cj + dk\,, \quad \textit{con } a, b, c, d \in \mathbb{R}, \quad i, j, k \ \textit{ los elementos de la base canónica de } \mathbb{R}^3.$$

El conjunto de cuaterniones se representa con la letra $\mathbb{H}$, en honor a su creador Sir William Rowan Hamilton.

Además, como cada cuaternión q vendrá definido por los coeficientes a, b, c, d que forman el vector (a, b, c, d), puede identificarse $\mathbb{H}$ con $\mathbb{R}^4$ considerado como espacio vectorial, con la suma y la ley externa definidas de forma natural a partir de las de $\mathbb{R}^4$. Dado $a \in \mathbb{H}$, también se representa $q = a + \vec{v}$ con $\vec{v} = (b, c, d) = bi + cj + dk$.

Es natural observar que puede dividirse el cuaternión en dos partes, una real y una imaginaria, de la forma $\mathfrak{Re}(q) = a$, $\mathfrak{Im}(q) = bi + cj + dk$, que respectivamente pueden considerarse como un número real y un vector de $\mathbb{R}^3$. Así, cualquier cuaternión $q \in \mathbb{H}$ puede escribirse como $q = a + v$, con $\mathfrak{Re}(q) = a \in \mathbb{R}$, $\mathfrak{Im}(q) = v \in \mathbb{R}^3$.

Veamos mediante la siguiente definición cómo es el producto de cuaterniones.

Definición 3.3.2.2. *Sean $q_1, q_2 \in \mathbb{H}$, $q_1 = a_1 + b_1 i + c_1 j + d_1 k$, $q_2 = a_2 + b_2 i + c_2 j + d_2 k$. Se define el producto de cuaterniones como:*

$$\begin{aligned} q_1 \cdot q_2 = {} & (a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2) + (a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2)i + \\ & + (a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2)j + (a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2)k \end{aligned} \tag{3.21}$$

¿De dónde sale esta expresión de apariencia tan *artificial*? Recordemos, para justificarla, la cita que daba comienzo a la sección 3.3., en la que Hamilton explicaba a uno de sus hijos cómo fue aquel momento en el que, años atrás, *descubrió* la expresión $i^2 = j^2 = k^2 = ijk = -1$. Esta expresión se convertiría

en la fórmula fundamental de los cuaterniones y permite el desarrollo del álgebra de cuaterniones que estamos introduciendo.

Nota 3.3.2.1. *De la expresión $i^2 = j^2 = k^2 = ijk = -1$ pueden obtenerse las siguientes Reglas:*

$$\begin{cases} ij = ij(-k^2) = -ijk \cdot k = k \\ ik = -j \\ ji = -k \\ jk = (-i^2)jk = -i(ijk) = i \\ ki = -(ji)i = -ji^2 = j \\ kj = (ij)j = ij^2 = -i \end{cases} \tag{3.22}$$

De las Reglas anteriores se deduce que la definición del producto, como se ve en la siguiente Proposición.

Proposición 3.3.2.1. *Utilizando las Reglas que se derivan de la expresión $i^2 = j^2 = k^2 = ijk = -1$, detalladas en (3.22), se deriva que el producto de cuaterniones se expresa como:*

$$\begin{aligned} q_1 \cdot q_2 = (a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2) + (a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2)i + \\ +(a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2)j + (a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2)k \end{aligned} \tag{3.23}$$

Demostración.

Desarrollando parcialmente el producto q_1q_2 y usando las identidades obtenidas en (3.22) se tiene:

$$q_1q_2 = (a_1 + b_1i + c_1j + d_1k)(a_2 + b_2i + c_2j + d_2k) =$$

$$\begin{aligned} = a_1a_2 + a_1b_2i + a_1c_2j + a_1d_2k + b_1a_2i + b_1b_2i^2 + b_1c_2ij + b_1d_2ik + \\ +c_1a_2j + c_1b_2ij + c_1c_2j^2 + c_1d_2jk + d_1a_2k + d_1b_2ki + d_1c_2kj + d_1d_2k^2 = \end{aligned}$$

$$\begin{aligned} = (a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2) + (a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2)i + \\ +(a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2)j + (a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2)k \end{aligned}$$

□

Definición 3.3.2.3. *Si $q \in \mathbb{H}$ verifica que $\mathfrak{Re}(q) = 0$, entonces se le denomina cuaternión imaginario.*

Lema 3.3.2.1. *Sean $q_1 = a_1 + v_1 \in \mathbb{H}$, $q_2 = a_2 + v_2 \in \mathbb{H}$, entonces:*

$$q_1q_2 = a_1a_2 - \langle v_1v_2 \rangle + a_1v_2 + a_2v_1 + v_1 \times v_2 \tag{3.24}$$

De donde se deduce que $\mathfrak{Re}(q_1 \cdot q_2) = a_1a_2 - \langle v_1v_2 \rangle$, $\mathfrak{Im}(q_1 \cdot q_2) = a_1v_2 + a_2v_1 + v_1 \times v_2$.

En particular, si tanto q_1 como q_2 son cuaterniones imaginarios, i.e. $q_1 = v_1$ y $q_2 = v_2$, entonces:

$$q_1 \cdot q_2 = -v_1 v_2 + v_1 \times v_2 \tag{3.25}$$

Demostración.

Sean $q_1 = a_1 + v_1 = a_1 + (b_1, c_1, d_1) \in \mathbb{H}$, $q_2 = a_2 + v_2 = a_2 + (b_2, c_2, d_2) \in \mathbb{H}$.

El producto de cuaterniones se expresa:

$$q_1 q_2 = (a_1 + b_1 i + c_1 j + d_1 k)(a_2 + b_2 i + c_2 j + d_2 k) =$$

$$= a_1 a_2 + a_1 b_2 i + a_1 c_2 j + a_1 d_2 k + b_1 a_2 i + b_1 b_2 i^2 + + b_1 c_2 ij + b_1 d_2 ik +$$
$$+ c_1 a_2 j + c_1 b_2 ij + c_1 c_2 j^2 + c_1 d_2 jk + d_1 a_2 k + d_1 b_2 ki + d_1 c_2 kj + d_1 d_2 k^2 =$$

$$= a_1 a_2 - b_1 b_2 - d_1 d_2 - c_1 c_2 + (a_1 b_2 + b_1 a_2) i + (a_1 c_2 + c_1 a_2) j +$$
$$+ (a_1 d_2 + d_1 a_2) k + b_1 c_2 k - b_1 d_2 j - c_1 b_2 k + c_1 d_2 i + d_1 b_2 j - d_1 c_2 i =$$

$$= a_1 a_2 - \langle v_1, v_2 \rangle + a_1 (b_2 i + c_2 j + d_2 k) + a_2 (b_1 i + a_1 j + d_1 k) +$$
$$+ (c_1 d_2 - d_1 c_2) i + (d_1 b_2 - b_1 d_2) j + (b_1 c_2 - c_1 b_2) k =$$

$$= a_1 a_2 - \langle (b_1, c_1, d_1), (b_2, c_2, d_2) \rangle + a_1 v_2 + a_2 v_1 + det \begin{bmatrix} i & j & k \\ b_1 & c_1 & d_1 \\ b_2 & c_2 & d_2 \end{bmatrix} =$$

$$= a_1 a_2 - \langle v_1, v_2 \rangle + a_1 v_2 + a_2 v_1 + v_1 \times v_2$$

Donde $\mathfrak{Re}(q_1 q_2) = a_1 a_2 - \langle v_1, v_2 \rangle$, $\mathfrak{Im}(q_1 q_2) = a_1 v_2 + a_2 v_1 + v_1 \times v_2$.

□

Corolario 3.3.2.1. *La multiplicación de cuaterniones es asociativa, es decir:*

$$(q_1 q_2) q_3 = q_1 (q_2 q_3) \ \ \forall q_1, q_2, q_3 \in \mathbb{H}$$

Demostración.

Sean $q_1, q_2, q_3 \in \mathbb{H}$, $q_1 = a_1 + v_1$, $q_2 = a_2 + v_2$,$q_3 = a_3 + v_3$. Entonces los productos $(q_1 q_2) q_3$ y $q_1 (q_2 q_3)$ son, respectivamente:

$$(q_1q_2)q_3 = (a_1a_2 - \langle v_1, v_2\rangle + a_1v_2 + a_2v_1 + v_1 \times v_2)q_3 =$$

$$= (a_1a_2 - \langle v_1, v_2\rangle)a_3 - \langle a_1v_2 + a_2v_1 + v_1 \times v_2, v_3\rangle + (a_1a_2 - \langle v_1, v_2\rangle)v_3 +$$
$$+a_3(a_1v_2 + a_2v_1 + v_1 \times v_2) + (a_1v_2 + a_2v_1 + v_1 \times v_2) \times v_3 =$$

$$= a_1a_2a_3 - a_3\langle v_1, v_2\rangle - a_1\langle v_2, v_3\rangle - a_2\langle v_1, v_3\rangle + \langle v_1 \times v_2, v_3\rangle + a_1a_2v_3 - \langle v_1, v_2\rangle v_3 +$$
$$+a_3a_1v_2 + a_3a_2v_1 + a_3(v_1 \times v_2) + a_1v_2 \times v_3 + a_2v_1 \times v_3 + (v_1 \times v_2) \times v_3$$

$$q_1(q_2q_3) = q_1(a_2a_3 - \langle v_2, v_3\rangle + a_2v_3 + a_3v_2 + v_2 \times v_3) =$$

$$= a_1(a_2a_3 - \langle v_2, v_3\rangle) - \langle v_1, a_2v_3 + a_3v_2 + v_2 \times v_3\rangle + a_1(a_2v_3 + a_3v_2 + v_2 \times v_3) +$$
$$+(a_2a_3 - \langle v_2, v_3\rangle)v_1 + v_1 \times (a_2v_3 + a_3v_2 + v_2 \times v_3) =$$

$$= a_1a_2a_3 - a_1\langle v_2, v_3\rangle - a_2\langle v_1, v_3\rangle - a_3\langle v_1, v_2\rangle + \langle v_1, v_2 \times v_3\rangle + a_1a_2v_3 + a_1a_3v_2 +$$
$$+a_1(v_2 \times v_3) + a_2a_3v_1 - \langle v_2, v_3\rangle v_1 - a_2v_3 \times v_1 - a_3v_2 \times v_1 - (v_2 \times v_3) \times v_1$$

Ambos productos son iguales, lo que demuestra que el producto de cuaterniones verifica la propiedad asociativa.

□

Proposición 3.3.2.2. *La multiplicación de cuaterniones no es conmutativa.*

Demostración.

Veamos esto con un contraejemplo.

Sean $q_1 = 2 + (1, 0, 1)$, $q_2 = 3 + (1, 2, 3)$, $q_1, q_2, \in \mathbb{H}$. Los respectivos productos q_1q_2 y q_2q_1 son:

$$q_1q_2 = 2(1, 2, 3) + 3(1, 0, 1) + (1, 0, 1) \times (1, 2, 3) = (5, 4, 9) + (-2, -2, 2) = (3, 2, 11)$$

$$q_2q_1 = 3(1, 0, 1) + 2(1, 2, 3) + (1, 2, 3) \times (1, 0, 1) = (5, 4, 9) + (2, 2, -2) = (7, 6, 7)$$

Luego $q_1q_2 \neq q_2q_1$ y, por lo tanto, queda demostrado que la multiplicación de cuaterniones no es conmutativa.

□

Una vez estudiado el producto de cuaterniones, es momento de definir la norma y los conceptos de cuaternión unitario y conjugado, los cuales dejarán entrever similitudes entre $\mathbb{H}$ y $\mathbb{C}$.

Definición 3.3.2.4. *Sea el cuaternión $q = a + bi + cj + dk = a + v \in \mathbb{H}$. Se define:*

i) El módulo o norma del cuaternión q como $|q|^2 = a^2 + b^2 + c^2 + d^2 = a^2 + ||v||^2$ que, por abuso de notación, con frecuencia se identificarán $||v|| = |v|$.

ii) Si $q \in \mathbb{H}$ verifica que $|q| = 1$, entonces se le denomina cuaternión unitario. Se denota el conjunto de cuaterniones unitarios como $\mathcal{U} = \{q \in \mathbb{H} \,/\, |q| = 1\} \subset \mathbb{H}$.

iii) El conjugado $\bar{q}$ de un cuaternión q se define como $\bar{q} = a - bi - cj - dk = a - v$, es decir, se consigue invirtiendo el signo de la parte imaginaria.

iv) Se representa el espacio de los cuaterniones imaginarios de la forma: $\mathfrak{Im}\ \mathbb{H} = \{q \in \mathbb{H} \,/\, \bar{q} = -q\}$, que es justamente $\mathbb{R}^3$ considerado como espacio vectorial. Cabe señalar, además, que si q es un cuaternión imaginario, i.e. $q = v \in \mathbb{H}$, entonces $q^2 = v^2 = -v \cdot v = -|v|^2$, que es un número real.

Detengámonos un momento a estudiar algunas de las propiedades de la norma y los conjugados de los cuaterniones.

Lema 3.3.2.2. *Sean los cuaterniones $q_1, q_2, q_3 \in \mathbb{H}$. Entonces:*

i) $q\bar{q} = |q|^2$.

ii) $\overline{q_1 q_2} = \overline{q_2} \cdot \overline{q_1}$.

iii) $|q_1 q_2| = |q_1| \cdot |q_2|$.

Demostración.

Veamos en primer lugar la demostración de i).

Sea $q = a+v \in \mathbb{H}$, entonces $\bar{q} = a-v$, luego $q\bar{q} = a^2 - v(-v) + a(-v) + av + v \times (-v) = a^2 + v^2 = |q|^2$.

Demostremos ahora ii).

Sean $q_1 = a_1 + v_1 \in \mathbb{H}$, $q_2 = a_2 + v_2 \in \mathbb{H}$, entonces $\overline{q_1} = a_1 - v_1$, $\overline{q_2} = a_2 - v_2$. Sabemos que

$q_1q_2 = a_1a_2 - v_1v_2 + a_1v_2 + a_2v_1 + v_1 \times v_2$, por lo tanto, $\overline{q_1q_2} = a_1a_2 - v_1v_2 - a_1v_2 - a_2v_1 - v_1 \times v_2$.

Por otro lado:

$$\overline{q_2} \cdot \overline{q_1} = a_2a_1 - (-v_2)(-v_1) + a_2(-v_1) + a_1(-v_2) + (-v_2) \times (-v_1) = \\ = a_1a_2 - v_1v_2 - a_2v_1 - a_1v_2 - (v_2 \times v_1)$$

Como por propiedades del producto vectorial es cierto que $v_1 \times v_2 = -(v_2 \times v_1)$, acabamos de comprobar que se verifica que $\overline{q_1q_2} = \overline{q_2} \cdot \overline{q_1}$.

Por último, demostremos iii).

Por i) podemos afirmar que $|q_1q_2|^2 = q_1q_2\overline{q_1q_2}$.

Por ii) podemos añadir que $|q_1q_2|^2 = q_1q_2\overline{q_1q_2} = q_1q_2\overline{q_2} \cdot \overline{q_1}$.

Aplicando i) de nuevo tenemos que $|q_1q_2|^2 = q_1q_2\overline{q_1q_2} = q_1q_2\overline{q_2} \cdot \overline{q_1} = q_1|q_2|^2\bar{q_1} = |q_1|^2|q_2|^2$.

□

Nota 3.3.2.2. *Es interesante destacar los hechos que siguen.*

i) Cabe añadir que de la afirmación i) del Lema anterior se desprende una propiedad interesante: $q^{-1} = \frac{1}{|q|}\bar{q}$ *con* $q \neq 0$*, que conlleva que* $\mathbb{H}$ *es un anillo de división o cuerpo no conmutativo.*

ii) También es importante comentar que si v *es un cuaternión unitario imaginario fijo, i.e.* $v \in \mathfrak{Im}\,\mathbb{H} \cap \mathcal{U}$*, entonces se puede identificar el subespacio bidimensional* $\langle 1, v\rangle = \{a + bv \,/\, a, b, \in \mathbb{R}\} \subseteq \mathbb{H}$ *con* $\mathbb{C}$ *por medio de la transformación siguiente:*

$$\varphi_{\vec{v}} : \mathbb{C} \to \langle 1, \vec{v}\rangle \subseteq \mathbb{H} \\ a + ib \to a + b\vec{v}$$

Así, las multiplicaciones son consistentes en ambos lados, pues:

$$(a_1 + b_1v)(a_2 + b_2v) = a_1a_2 + (a_1b_2 + b_1a_2)v + b_1b_2v^2 = (a_1a_2 - b_1b_2) + (a_1b_2 + b_1a_2)v$$

$$(a_1 + b_1i)(a_2 + b_2i) = a_1a_2 + (a_1b_2 + b_1a_2)i - b_1b_2 = (a_1a_2 - b_1b_2) + (a_1b_2 + b_1a_2)v$$

iii) Es interesante resaltar que el conjunto de cuaterniones imaginarios está parametrizado por la esfera unitaria $S^2(1)$, ya que, dado $\vec{v} \in Im(\mathbb{H})$, $|\vec{v}|^2 = 1$ si y solo si $\vec{v} \in S^2(1)$. Como $\forall \vec{v} \in Im(\mathbb{H}) \cap S^2(1)$, entonces $\varphi_{\vec{v}} : \mathbb{C} \to \langle 1, \vec{v} \rangle$, y así tenemos que hay una esfera de planos complejos dentro de $\mathbb{H}$. Esta forma en que $\mathbb{H}$ extiende $\mathbb{C}$ cuando se aplica a los cuaterniones, es lo que se denomina geometría hipercompleja.

En la sección que se presenta a continuación se explican algunas conclusiones obtenidas al trabajar con cuaterniones y se presentan las rotaciones expresadas mediante el uso de cuaterniones.

3.3.3. Los cuaterniones en las rotaciones.

A lo largo de esta sección presentaré cómo los cuaterniones juegan un papel vital en la representación de las rotaciones en los espacios tridimensionales.

Para poder relacionarlas correctamente, es necesario conocer primero el grupo de cuaterniones unitarios. Si bien el cuaternión 0 no tiene inverso para el producto y por lo tanto el conjunto $\mathbb{H}$ no es grupo, sí lo es el conjunto $\mathbb{H}\backslash\{0\}$ y, como se demuestra en la Proposición que sigue, también el grupo de cuaterniones unitarios.

Proposición 3.3.3.1. *El conjunto de cuaterniones unitarios $\mathcal{U} = \{a \in \mathbb{H} \ / \ |q| = 1\}$ es un grupo.*

Demostración.

Por un lado, en la sección anterior se ha demostrado que los cuaterniones verifican la propiedad asociativa, aunque no la conmutativa para el producto.

En el Lema 3 de la sección anterior se ha demostrado que dados $q_1, q_2 \in \mathcal{U}$, entonces $|q_1 q_2| = |q_1||q_2| = 1$, luego $\mathcal{U}$ es un conjunto cerrado bajo el producto. Además, dado $q \in \mathcal{U}$ entonces $q^{-1} = \bar{q} \in \mathcal{U}$, por lo que también será cerrado bajo la inversión.

□

Nota 3.3.3.1. *Dado que los cuaterniones unitarios son de la forma $q = a + bi + cj + dk, \ \ con \ \ a^2 + b^2 + c^2 + d^2 = 1$, el conjunto de cuaterniones unitarios puede denotarse también, para facilitar su interpretación, como $\mathcal{U} = \{(a, b, c, d) \in \mathbb{R}^4 \ / \ a^2 + b^2 + c^2 + d^2 = 1\}$, que coincide con la hiperesfera $S^3(1)$, formada por todos los vectores de cuatro dimensiones y con distancia unitaria al origen.*

Proposición 3.3.3.2. *Sea $q \in \mathcal{U}$. Si $w \in \mathfrak{Im}\ \mathbb{H}$, entonces $qw\bar{q}$ es un cuaternión imaginario con la misma norma que w.*

Demostración.

Sea $w \in \mathfrak{Im}\ \mathbb{H}$, entonces $\bar{w} = -w$ y $w^2 = -|w|^2$. Sea también $p = qw\bar{q}$.

Por un lado, por la propiedad de la conjugación, se tiene que $\bar{p} = \bar{\bar{q}}\bar{w}\bar{q} = q(-q)\bar{q} = -p$, por lo que $p = qw\bar{q}$ es también un cuaternión imaginario.

Por otro lado, $|p|^2 = p\bar{p} = (qw\bar{q})(-qw\bar{q}) = q(-w^2)\bar{q} = q|w|^2\bar{q} = |w|^2$. □

Esta Proposición motiva la siguiente definición.

Definición 3.3.3.1. *Dado $q = a + \vec{v} \in \mathcal{U}$, se define la aplicación $R[q] : Im(\mathbb{H}) \to Im(\mathbb{H})$ como:*

$$R[q]w = qw\bar{q} = (a + v)w(a - v) = a^2w + a(vw - wv) - vwv \tag{3.26}$$

Con $w \in \mathfrak{Im}\ \mathbb{H}$.

Notar que, en virtud de la identificación $Im(\mathbb{H}) \equiv \mathbb{R}^3$, $R[q] : \mathbb{R}^3 \to \mathbb{R}^3$.

Teorema 3.3.3.1. *Sea q un cuaternión unitario. Entonces $R[q]$ representa una rotación en sentido antihorario de ángulo $2arccos(\mathfrak{Re}q)$, comprendido entre 0 y 2π, con los ejes apuntando en la dirección de $\mathfrak{Im}\ q$. Debe entenderse por sentido antihorario lo equivalente a la Regla del "sacacorchos" de la mano derecha que se utiliza comúnmente en física y que ilustra la figura 3.1 que sigue.*

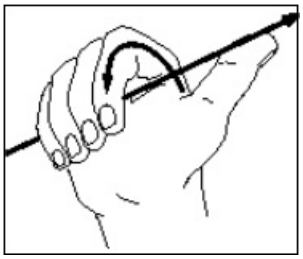

Figura 3.1: Regla de la mano derecha ilustrando el sentido antihorario.

Demostración.

Sean $v = \mathfrak{Im}\ q$, $q = a + \vec{v} \in \mathcal{U}$. Entonces existe un único ángulo $\theta \in [0, 2\pi]$ tal que $a = cos(\frac{\theta}{2})$ y $|v| = sin(\frac{\theta}{2}) \geq 0$. Veamos esto con detalle.

Sea $q \in \mathcal{U}$, i.e. $|q| = 1$, luego $|q|^2 = a^2 + |\vec{v}|^2 = 1$. Entonces existe un único $\alpha \in [0, 2\pi)$ tal que $a = cos(\alpha)$ y $|\vec{v}| = sen(\alpha)$. Sea $\theta = 2\alpha$, entonces $\alpha = Arccos(\alpha)$, por tanto $\theta = 2\alpha = 2Arccos(\alpha) = 2Arccos(\mathfrak{Re}\ q)$.

Ahora, si $v = 0$, entonces es evidente que $q = \pm 1$ y o bien $\theta = 0$ o bien $\theta = 2\pi$, lo que representa la rotación identidad y verifica el Teorema.

Cuando $|v| \neq 0$, entonces, según se ha definido la aplicación, tenemos que:

$$R[q]v = a^2 v + a(vv - vv) - vvv = a^2 v + |v|^2 v = (a^2 + |v|^2)v = v$$

Sea el cuaternión unitario imaginario $w_1 \in \mathfrak{Im}\ \mathbb{H} = \mathbb{R}^3$ perpendicular a v. Entonces $w_1 v = 0$ y $vw_1 = -w_1 v = v \times w_1$. Si se define $w_3 = \frac{1}{|v|} v$, $w_2 = w_3 w_1$, entonces $\{w_1, w_2, w_3\}$ es una base ortonormal de $\mathbb{R}^3$ con $w_1 \times w_2 = w_3$. Además, utilizando la transformación $R[q]v$ definida, se tiene que:

$$R[q]w_3 = R[q]\left(\frac{1}{|\vec{v}|}\vec{v}\right) = \frac{1}{|\vec{v}|} R[q]\vec{v} = \frac{\vec{v}}{|\vec{v}|} = w_3$$

$$R[q]w_1 = w_1 cos^2(\frac{\theta}{2}) + 2vw_1 cos(\frac{\theta}{2}) - w_1 sin^2(\frac{\theta}{2}) = w_1 cos(\theta) + w_2 sin(\theta)$$

Y, del mismo modo, también:

$$R[q]w_2 = w_2 cos^2(\frac{\theta}{2}) + 2vw_2 cos(\frac{\theta}{2}) - w_2 sin^2(\frac{\theta}{2}) = -w_1 sin(\theta) + w_2 cos(\theta)$$

Con esto se define el sistema de ecuaciones que sigue:

$$R[q]w = \begin{cases} R[q]w_1 = w_1 cos(\theta) + w_2 sin(\theta) \\ R[q]w_2 = -w_1 sin(\theta) + w_2 cos(\theta) \end{cases} \tag{3.27}$$

¿A qué resultado acabamos de llegar? La rotación $R[q]$ definida mediante el sistema de ecuaciones (3.27) es equivalente a la matriz A definida en (3.4), la cual representa un giro de ángulo θ en sentido horario. Así pues, la transformación lineal $R[q]$ tiene una matriz relativa a la base ortonormal $\{w_1, w_2, w_3\}$ de la forma:

$$[R[q]]_{\{w_1, w_2, w_3\}} = \begin{pmatrix} cos(\theta) & -sen(\theta) & 0 \\ sen(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Y, por tanto es un giro, luego acabamos de demostrar que se verifica el enunciado del Teorema.

□

Es natural preguntarse, llegado este punto, si la matriz de rotación alrededor de un eje $\vec{w}$ cualquiera tiene su equivalente en los cuaterniones, es decir, si utilizando los cuaterniones puede conseguirse una matriz que describa una rotación alrededor de un eje arbitrario. Para poder entender el papel que juegan los cuaterniones en la representación de rotaciones, partiremos de la siguiente Regla, que asumiremos sin demostración, la cual proviene de la interpretación física de *velocidad angular*.

Regla 3.3.3.1. *Cualquier rotación puede representarse mediante un vector $\vec{x} \in \mathbb{R}^3$ con $|\vec{x}| \leq \pi$, siendo el eje de rotación la dirección dada por dicho vector y su norma los radianes que representan el ángulo de giro, en sentido antihorario.*

Definamos, además, la bola cerrada $B^3(\pi) = \{\vec{x} \in \mathbb{R}^3 \ / \ |\vec{x}| \leq \pi\}$, equivalente a la unión de la esfera $S^2(\pi)$, de radio π, y todos los puntos situados en su interior.

¿Dónde encajan aquí los cuaterniones? Para comprenderlo, representaremos la rotación definida por su eje-vector unitario $\vec{x}$ y ángulo θ de forma analítica. Veamos, mediante la siguiente Proposición, cómo se expresa la rotación $R(\theta, \vec{x})$ con el cuaternión $q = a + \vec{v}$ donde $a = cos(\frac{\theta}{2})$, $\vec{v} = sen(\frac{\theta}{2})\vec{x}$.

Proposición 3.3.3.3. *Sea el cuaternión unitario $q = (q_0, q_1, q_2, q_3)$. Entonces la rotación $R[q]$ puede representarse respecto de la base canónica mediante la matriz:*

$$R(q) = \begin{pmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2q_1q_2 - 2q_0q_3 & 2q_1q_3 + 2q_0q_2 \\ 2q_1q_2 + 2q_0q_3 & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2q_2q_3 - 2q_0q_1 \\ 2q_1q_3 - 2q_0q_2 & 2q_2q_3 + 2q_0q_1 & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{pmatrix} \tag{3.28}$$

Siendo $q_0 = a = cos(\frac{\theta}{2})$, $q_1 = w_1b = w_1sin(\frac{\theta}{2})$, $q_2 = w_2b = w_2sin(\frac{\theta}{2})$, $q_3 = w_3b = w_3sin(\frac{\theta}{2})$, donde el vector $\vec{w} = (w_1, w_2, w_3)$ representa el eje de rotación.

Demostracion.

Sean $q_0 = a = cos(\frac{\theta}{2})$, $q_1 = w_1b = w_1sin(\frac{\theta}{2})$, $q_2 = w_2b = w_2sin(\frac{\theta}{2})$, $q_3 = w_3b = w_3sin(\frac{\theta}{2})$ las componentes del cuaternión $q = (q_0, q_1, q_2, q_3)$.

Lo primero que necesitamos es representar la matriz de rotaciones según el ángulo mitad. Llamaremos $a = cos(\frac{\theta}{2})$, $b = sin(\frac{\theta}{2})$.

Sabemos, por lo estudiado en la sección 3.2., que, siendo $c = cos(\theta)$, $s = sin(\theta)$, $\hat{w} = (w_1, w_2, w_3)$, se tiene la matriz de rotación de la forma:

$$R(\theta,\hat{w}) = \begin{pmatrix} c+w_1^2(1-c) & w_1w_2(1-c)-sw_3 & w_1w_3(1-c)+sw_2 \\ w_1w_2(1-c)+sw_3 & c+w_2^2(1-c) & w_2w_3(1-c)-sw_1 \\ w_1w_3(1-c)-sw_2 & w_2w_3(1-c)+sw_1 & c+w_3^2(1-c) \end{pmatrix}$$

Como nos interesa la representación mediante el ángulo mitad, utilizaremos las equivalencias que siguen.

Por un lado, como:

$$1-c = 1-cos(\theta) = 2sin^2(\frac{\theta}{2}) = 2b^2 \tag{3.29}$$

Entonces

$$c = 1-2b^2 \tag{3.30}$$

Por otro lado, como:

$$a^2+b^2 = cos^2(\frac{\theta}{2}) + sin^2(\frac{\theta}{2}) = 1 \tag{3.31}$$

Es cierto que:

$$c = 1-2b^2 = a^2+b^2-2b^2 = a^2-b^2 \tag{3.32}$$

Además, como $s = sin(\theta)$ y se verifica que $c^2+s^2 = cos^2(\theta)+sin^2(\theta) = 1$, entonces:

$$s^2 = 1-c^2 = 1-(1-2b^2) \tag{3.33}$$

Luego:

$$s = \sqrt{1-(1+4b^4-4b^2)} = \sqrt{-4b^4+4b^2} = 2b\sqrt{1-b^2} = 2b\sqrt{a^2} = 2ab \tag{3.34}$$

Luego, utilizando las equivalencias (3.30), (3.32), (3.34), la matriz $R(\theta,\hat{w})$ queda de la forma:

$$\begin{pmatrix} a^2-b^2+2b^2w_1^2 & 2b^2w_1w_2-2abw_3 & 2b^2w_1w_3+2abw_2 \\ 2b^2w_1w_2+2abw_3 & a^2-b^2+2b^2w_2^2 & 2b^2-w_2w_3-2abw_1 \\ 2b^2w_1w_3-2abw_2 & 2b^2w_2w_3+2abw_1 & a^2-b^2+2b^2w_3^2 \end{pmatrix} \tag{3.35}$$

Si, además, tenemos en cuenta que $w_1^2 + w_2^2 + w_3^2 = 1$, entonces se tiene que:

$$a^2 - b^2 + 2b^2w_1^2 = a^2 - (w_1^2 + w_2^2 + w_3^2)b^2 + 2b^2w_1^2 = a^2 + b^2(w_1^2 - w_2^2 - w_3^2) \tag{3.36}$$

$$a^2 - b^2 + 2b^2w_2^2 = a^2 - (w_1^2 + w_2^2 + w_3^2)b^2 + 2b^2w_2^2 = a^2 + b^2(-w_1^2 + w_2^2 - w_3^2) \tag{3.37}$$

$$a^2 - b^2 + 2b^2w_3^2 = a^2 - (w_1^2 + w_2^2 + w_3^2)b^2 + 2b^2w_3^2 = a^2 + b^2(-w_1^2 - w_2^2 + w_3^2) \tag{3.38}$$

Sustituyendo (3.36), (3.37), (3.38) en la matriz (3.35) obtenemos:

$$\begin{pmatrix} a^2 + b^2(w_1^2 - w_2^2 - w_3^2) & 2b^2w_1w_2 - 2abw_3 & 2b^2w_1w_3 + 2abw_2 \\ 2b^2w_1w_2 + 2abw_3 & a^2 + b^2(-w_1^2 + w_2^2 - w_3^2) & 2b^2 - w_2w_3 - 2abw_1 \\ 2b^2w_1w_3 - 2abw_2 & 2b^2w_2w_3 + 2abw_1 & a^2 + b^2(-w_1^2 - w_2^2 + w_3^2) \end{pmatrix} \tag{3.39}$$

Si llamamos $q_0 = a = cos(\frac{\theta}{2})$, $q_1 = bw_1 = w_1 sin(\frac{\theta}{2})$, $q_2 = bw_2 = w_2 sin(\frac{\theta}{2})$, $q_3 = bw_3 = w_3 sin(\frac{\theta}{2})$, la matriz anterior es equivalente a:

$$R[q] = \begin{pmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2q_1q_2 - 2q_0q_3 & 2q_1q_3 + 2q_0q_2 \\ 2q_1q_2 + 2q_0q_3 & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2q_2q_3 - 2q_0q_1 \\ 2q_1q_3 - 2q_0q_2 & 2q_2q_3 + 2q_0q_1 & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{pmatrix} \tag{3.40}$$

Por lo tanto, acabamos de demostrar que la rotación $R(\theta, \vec{w})$ tiene su equivalente representación $R[q]$ utilizando los cuaterniones.

□

Es conveniente remarcar que q y $-q$ determinan la misma transformación, debido a las propiedades de los ángulos mitad. Veamos esto.

Corolario 3.3.3.1. *Sea el cuaternión $q \in \mathbb{H}$ tal que $q = (a, b, c, d)$, entonces $R[q] = R[-q]$.*

Demostración.

Basándonos en la matriz de rotación (3.40) definida utilizando cuaterniones en la Proposición 3.3.3.3., si la rotación se hace sobre $-q = (-a, -b, -c, -d)$, entonces se tiene:

$$R[-q] = \begin{pmatrix} (-a)^2 + (-b)^2 - (-c)^2 - (-d)^2 & 2(-b)(-c) - 2(-a)(-d) & 2(-a)(-c) + 2(-b)(-d) \\ 2(-b)(-c) + 2(-a)(-d) & (-a)^2 - (-b)^2 + (-c)^2 - (-d)^2 & 2(-c)(-d) - 2(-a)(-b) \\ 2(-b)(-d) - 2(-a)(-c) & 2(-a)(-b) + 2(-c)(-d) & (-a)^2 - (-b)^2 - (-c)^2 + (-d)^2 \end{pmatrix} =$$

$$= \begin{pmatrix} a^2+b^2-c^2-d^2 & 2bc-2ad & 2ac+2bd \\ 2bc+2ad & a^2-b^2+c^2-d^2 & 2cd-2ab \\ 2bd-2ac & 2ab+2cd & a^2-b^2-c^2+d^2 \end{pmatrix} = R[q]$$

□

Para terminar la sección, cabe añadir que resultaría natural plantearse si matriz puede estar completamente determinada por la parte imaginaria $\mathfrak{Im}\ q = (b, c, d)$, definiendo $a = \sqrt{1-b^2-c^2-d^2}$. No obstante, esta idea debe descartarse porque $\mathfrak{Re}q$ puede ser negativa y cambiar el signo de q para ajustar la definición de a en función de b, c, d a este hecho conlleva invertir el signo de v y del ángulo de rotación.

3.3.4. Ángulos de Euler, giroscopios y cuaterniones. Pese a su complejidad, ¿conviene utilizar los cuaterniones?

El modo tradicional de especificar una rotación es descomponiéndola en la composición de tres rotaciones, cada una alrededor de uno de los ejes cartesianos, y se utiliza en la actualidad en algunos sistemas de giroscopios o, por ejemplo, en las maquinarias de los parques de atracciones.

Para presentar los ángulos de Euler de modo que resulten comprensibles, empecemos imaginando tres ejes perpendiculares en el espacio de tres dimensiones, correspondiéndose con el esquema clásico conocido de los ejes X, Y, Z.

Si, teniendo esto anterior en mente, colocamos ahora un avión en el origen y nos imaginamos su trayectoria desde el despegue hasta que alcanza la altura durante la que desarrollará un vuelo horizontal prolongado hacia su destino, es obvio que durante este periodo realizará movimientos que se pueden representar como combinaciones de giros entorno a cada uno de los tres ejes.

Llamemos ahora a cada uno de los ángulos que representan las rotaciones alrededor de los ejes X, Y, Z, respectivamente ψ, θ, ϕ. La tripleta formada por estos ángulos, (ψ, θ, ϕ) será lo que denominaremos ángulos de Euler, equivalente a la notación $(pitch, yaw, roll)$, como se observa en la figura 3.2. Estos ángulos se utilizan con bastante frecuencia para representar rotaciones con vectores tridimensionales; de hecho, si bien a nivel interno Unreal Engine - el motor gráfico con el que programé durante mi estancia en prácticas - utiliza cuaterniones, en la interfaz de programador permite establecer las rotaciones de los elementos mediante ángulos de Euler.

Cada tripleta de vectores ortonormales definiendo los ejes en un movimiento determinado se denominará marco, F_i, y una sucesión de movimientos - representados como giros mediante ángulos de Euler en las tripletas - se obtendrá componiendo distintos marcos.

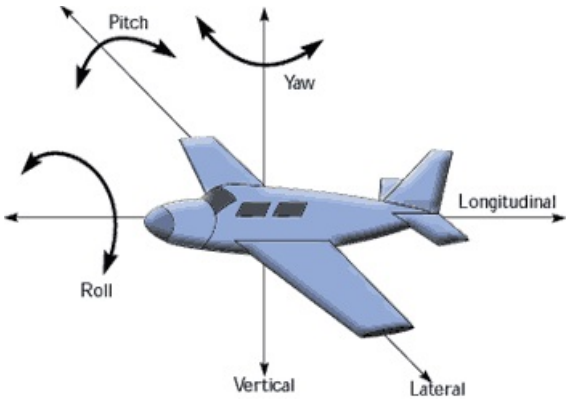

Figura 3.2: Esquema de las rotaciones $(pitch, yaw, roll)$ que definen los ángulos de Euler en un avión.

El Teorema de Euler afirma que cualquier elemento de $SO(3)$ puede descomponerse como composición de tres rotaciones alrededor de ejes ortogonales, en cualquier orden. Atendiendo a lo estudiado sobre cuaterniones unitarios, presentaremos ahora un Teorema que recoge esta idea.

Antes, no obstante, es necesario destacar que se considerará, en el desarrollo del Teorema, el círculo $\mathcal{U}_v = \{a + bv \ / \ a^2 + b^2 = 1\}$ formado por la intersección del subespacio $\langle 1, v\rangle$ con $\mathcal{U}$.

Teorema 3.3.4.1. *Sea $q \in \mathcal{U}$. Entonces existen:*

i) $q_1 \in \mathcal{U}_i, q_2 \in \mathcal{U}_j, q_3 \in \mathcal{U}_k$ tales que $q = q_3q_2q_1$.

ii) $p_1 \in \mathcal{U}_i, p_2 \in \mathcal{U}_j, p_3 \in \mathcal{U}_k$ tales que $q = p_3p_2p_1$

Omitiremos la demostración de este Teorema, pero demostraremos el Lema que sigue.

Lema 3.3.4.1. *Dado $v \in \mathfrak{Im}\mathbb{H}$, existen $a, b, c \in \mathbb{R}$ tales que $\mathfrak{Im}[(a + i)(b + j)(c + k)] = v$.*

Demostración.

Sea $v = ri + sj + tk$. Sabemos (por hipótesis) que:

$$\mathfrak{Im}[(a + i)(b + j)(c + k)] = \mathfrak{Im}[(ab + aj + ib + k)(c + k)] =$$

$$= \mathfrak{Im}[abc + abk + acj + ajk + icb + ikb + ck + k^2] =$$

$$= \mathfrak{Im}[abc + abk + acj + ai + icb - jb + ck + 1] =$$

$$= \mathfrak{m}[abc + (a + cb)i + (ac - b)j + (ab + c)k - 1] = v$$

Por lo tanto debe verificarse que:

$$\begin{cases} r = (a + cb) \\ s = (ac - b) \\ t = (ab + c) \end{cases}$$

Luego:

$$\begin{cases} a(1 + c^2) = cs + r \\ c(1 + a^2) = as + t \end{cases}$$

De aquí se deduce que: $a^5 - ra^4 + 2a^3 + (st - 2r)a^2 + (t^2 - s^2 + 1)a - r - st = 0$.

Está claro que esta ecuación de quinto grado tiene al menos una raíz real, que es a. Con esto, el sistema inicial se reduce a dos ecuaciones lineales que se pueden resolver para b y c.

□

Se ha hablado a lo largo de este documento de las bondades de los cuaterniones, pero su uso parece más complejo que los ángulos de Euler. ¿Por qué no se utilizan en su lugar entonces? El motivo está justificado; por un lado los ángulos de Euler dependen de la elección de la secuencia de ejes que definen la rotación, pero lo que más les aleja de situarse como una herramienta mejor para representar las rotaciones es su propensión a sufrir lo que se denomina *Gimbal Lock*.

¿Cuándo ocurre esto? Si nos interesan hechos reales como respuesta, puede revisarse lo que ocurrió con el viaje del Apollo 11 a la Luna [3], que al entrar en Gimbal Lock perdió toda estabilidad y empezó a dar bandazos sin sentido, hasta que mediante el control manual pudo salir de dicho estado. Si vamos un poco más allá, no es difícil detectar que cuando un ángulo de Euler se aproxime a $\frac{\pi}{2}$, su coseno lo hará a cero, y esto derivará en graves problemas si los cómputos los está llevando a cabo un ordenador, incapaz de dividir entre 0.

En los casos en los que se pierde uno de los grados de libertad por entrar en Gimbal Lock, los ángulos que definen la rotación respecto a los otros dos ejes dejan de ser independientes y desencadenan el caos en el sistema sobre el que están aplicando las rotaciones.

La ventaja de utilizar rotaciones basadas en cuaterniones imaginarios es que la aplicación $B^3(\pi) \rightarrow SO(3)$ definida por $\rho(x) = exp(A[x]) = R[q]$ mantiene siempre las derivadas parciales linealmente independientes, ρ_x, ρ_y, ρ_z, evitando así perder un grado de libertad cayendo en Gimbal Lock. Esto no es así con la aplicación $\sigma : R \rightarrow SO(3)$ siendo $\sigma(\psi, \theta, \phi)$ el marco con las rotaciones, $R = [0, 2\pi] \times [-\pi/2, \pi/2] \times [-\pi, \pi]$ el dominio usual sobre el que se definen los ángulos de Euler.

[3] https://www.hq.nasa.gov/alsj/gimbals.html

Capítulo 4

Conclusiones.

Poco después de comenzar mi estancia en prácticas con el equipo Catness, detecté comportamientos extraños en la ejecución de las funciones programadas, especialmente en aquellas que incluían giros.

Una de las primeras cosas que hube de tener en cuenta fue que los ángulos no se expresaban en radianes sino en grados, pero pese a *retocar* mi código teniendo en cuenta esto, seguía sin obtener los resultados esperados. Fue entonces cuando detecté que Unreal Engine 4 utiliza los ángulos de Euler [1] para representar las coordenadas de los puntos desde el modo *programador*, y no su representación "tradicional" respecto al origen y el sistema de coordenadas cartesiano.

Con los ángulos de Euler, lo que representa cada una de las coordenadas es una rotación. En particular, en tres dimensiones las representaciones son de la forma `(pitch, yaw, roll)`, las cuales corresponden, respectivamente, a rotaciones respecto de los ejes X, Y, Z. En otros sistemas donde se utilizan los ángulos de Euler, como los de los aeroplanos, los tres ángulos corresponden a la dirección ("heading" o yaw), elevación (pitch) y ángulo de alabeo (roll).

Esto, sumado al hecho de que, en una ocasión, la ejecución de una de mis funciones provocó *Gimbal Lock* y la consecuente pérdida de la posibilidad de rotación sobre uno de los ejes, despertó mi interés por estudiar la representación de las rotaciones en el espacio con métodos distintos a los que se me habían explicado a lo largo del grado. Tras explicárselo a mi tutor, coincidimos en que esta temática podría resultar interesante y nos pusimos a *investigar* sobre el tema, para descubrir cómo se tratan realmente las rotaciones en tres dimensiones más allá de las matrices de rotación explicadas en las asignaturas de geometría del grado.

Al iniciar el proceso de búsqueda de bibliografía relacionada con esta temática, pronto aparecieron los cuaterniones, que se presentaban como la herramienta más "estable" para la representación de isometrías en el espacio, especialmente de las rotaciones. Además, en la mayoría de fuentes consultadas se expli-

[1] Detalles sobre el uso de los ángulos de Euler en Unreal Engine 4 pueden encontrarse en: https://docs.unrealengine.com/latest/INT/API/Runtime/Core/Math/FRotator/index.html

caba que los motores gráficos - como por ejemplo Unity o Unreal Engine -, utilizan internamente en sus códigos fuente los cuaterniones para representaciones tridimensionales [2] o, al menos, los ofrecen al programador como herramienta para las rotaciones.

Dada la importancia que parecían presentar los cuaterniones, mi tutor y yo decidimos que resultaba interesante incluir en la sección matemática de mi PFG referencias a ellos. Así, decidimos que lo mejor era que, en primer lugar, recordara los conceptos sobre isometrías lineales y afines estudiados en el grado, lo cual queda reflejado en la sección 3.2. Tras esto, nos dedicamos a la comprensión de los cuaterniones y conseguimos encontrar su conexión con las isometrías que acababa de repasar. En la sección 3.3., tras presentar el álgebra de los cuaterniones, queda reflejado que es posible caracterizar las rotaciones en el espacio tridimensional mediante cuaterniones, apoyándonos para ello en la exponenciación de matrices ya conocida y, lo que es aún más importante, se ha evidenciado que hay una clara correspondencia entre las rotaciones representadas matricialmente en la Nota 2 de la sección 3.2. y la matriz $R[q]$ de rotación expresada en función de los cuaterniones en la sección 3.3.

Además, en la sección 3.3.6. se ha visto que, pese a que los cuaterniones presentan mayor dificultad de comprensión que los ángulos de Euler, consiguen representaciones más estables - ya que las funciones que los manipulan no presentan singularidades -, que evitan incurrir en *Gimbal Lock* y perder así grados de libertad en las rotaciones, lo que justifica sobradamente su uso.

Ahora que han concluido tanto mi estancia en prácticas como mi estudio sobre los cuaterniones, puedo decir que ambos me han resultado muy interesantes.

La estancia en prácticas, por su parte, me dio la oportunidad de conocer un componente de los videojuegos antes desconocido para mí: su programación. Pese a que me hubiera gustado que el tiempo compartido con Catness hubiera sido mayor y así haber podido completar las funciones programadas, generalizarlas y dotarlas de mayor funcionalidad, me siento satisfecha con lo conseguido a lo largo de las doscientas noventa horas, no solo a nivel de objetivos conseguidos, sino también en lo que a aprendizaje personal se refiere. El ambiente de trabajo fue muy agradable y me permitió aprender que la programación de videojuegos tiene una dificultad añadida respecto al tipo de programación que ya conocía hasta el momento: el usuario final. Aprendí que al diseñar un videojuego no solamente ha de presentarse un "paquete" de funciones útiles de fácil aprendizaje, sino también tan robustas como sea posible, para evitar así los *bugs* que tanto detestan quienes juegan a videojuegos y que, además, pueden explotarse con fines de dudosa moralidad en contra del videojuego final presentado. Además, a nivel de programación, las prácticas sirvieron para mejorar mi capacidad de diseño de algoritmos, afianzar conceptos sobre programación orientada a objetos y aprender a programar en `C++` utilizando Unreal Engine como motor gráfico, lo cual me enseñó a combinar la programación de código en `C++` con las posibilidades de programación que presentan los *Blueprints* de su interfaz gráfica.

Respecto a la vertiente matemática de mi PFG, he de decir que me ha resultado muy interesante el estudio de los cuaterniones y, a la vez, motivador, puesto que la temática la planteé yo al descubrir en mi

[2] En el Anexo B se presenta un extracto de código fuente real de Unreal Engine, correspondiente a la clase `FTransform`, en el que puede comprobarse que efectivamente se utilizan los cuaterniones en los motores gráficos para las rotaciones.

estancia en prácticas los ángulos de Euler y enfrentarme a un *Gimbal Lock*. Debo añadir que la ayuda de mi tutor ha sido esencial en ciertos momentos de "frustración" y es pues necesario agradecerle que consiguiera mantenerme motivada.

Espero que este documento haya resultado interesante e ilustrativo al lector, tanto como para mí lo fueron las experiencias que marcaron mi estancia en prácticas, de las que surge el capítulo dos, y el tiempo dedicado a la lectura e investigación sobre los cuaterniones, del que surge el capítulo tres.

Bibliografía

[1] HANSON, ANDREW J., *Visualizing quaternions*, Elsevier, San Francisco, 2006.

[2] GRAY, ALFRED, ABBENA, ELSA & SALAMON, SIMON, *Modern differential geometry of curves and surfaces with Mathematica*, Third edition, Chapman & Hall/CRC, Florida, 2006.

[3] VAN VERTH, JAMES M. & BISHOP, LARS M., *Essential Mathematics for games and interactive applications*, Second edition, Elsevier, Burlington, 2008.

[4] SÁNCHEZ MUÑOZ, JOSÉ MANUEL, *Historias de Matemáticas: Hamilton y el Descubrimiento de los Cuaterniones*, Revista de investigación *Pensamiento Matemático*, octubre de 2011.

[5] *Documentación en línea de Unreal Engine 4*, https://docs.unrealengine.com/ [Consulta: Marzo - Mayo, 2015]

[6] *Código programado por S. Rombauts para una malla procedural en Unreal Engine*, https://github.com/SRombauts/UE4ProceduralMesh [Consulta: Marzo 2015]

Anexo A

Código implementado durante la estancia en prácticas.

El código ejecutable en Unreal Engine se programa en `C++`. En este lenguaje cada una de las clases necesita siempre dos archivos, cuyas extensiones son `.h` y `.cpp`, que respectivamente contienen un *listado* de las cabeceras y propiedades (atributos) de la clase a implementar, y el código fuente de la clase (la implementación en sí de lo detallado en el archivo `.h`).

A continuación presento los códigos A.1, A.2, A.3, A.4, que respectivamente corresponden, en ese orden, a los archivos `.h` y `.cpp` de mis clases implementadas, que son las que definen el aspecto del cubo y sus funciones disponibles: `ProceduralCubeActor.h` y `ProceduralCubeActor.cpp` y las relativas a las reacciones del entorno ante las acciones del controlador: `CustomPlayerController.h` y `CustomPlayerController.cpp`, que básicamente perciben los clicks y desplazamientos del ratón y determinan su efecto.

Código A.1: ProceduralCubeActor.h

```
#pragma once

#include "NumericLimits.h"
#include "GameFramework/Actor.h"
#include "ProceduralMeshComponent.h"
#include "CustomPlayerController.h"
#include "ProceduralCubeActor.generated.h"

UCLASS()
class PROCEDURALMESH_API AProceduralCubeActor : public AActor
{
        GENERATED_UCLASS_BODY()

public:

        // ---------- PROPERTIES ---------- \\

        UPROPERTY(BlueprintReadWrite, Category = Materials)
        TSubobjectPtr<UProceduralMeshComponent> mesh;
```

```
20
21          UPROPERTY(BlueprintReadWrite, Category = Materials)
22          ACustomPlayerController* CustomPController;
23
24          UPROPERTY(BlueprintReadWrite, Category = Materials)
25          bool KeepOnHover;
26
27          UPROPERTY(BlueprintReadWrite, Category = Materials)
28          bool VertexMovementState;
29
30          UPROPERTY(BlueprintReadWrite, Category = Materials)
31          bool CubeFacesEditionState;
32
33          UPROPERTY(BlueprintReadWrite, Category = Materials)
34          FVector p0;
35          UPROPERTY(BlueprintReadWrite, Category = Materials)
36          FVector p1;
37          UPROPERTY(BlueprintReadWrite, Category = Materials)
38          FVector p2;
39          UPROPERTY(BlueprintReadWrite, Category = Materials)
40          FVector p3;
41          UPROPERTY(BlueprintReadWrite, Category = Materials)
42          FVector p4;
43          UPROPERTY(BlueprintReadWrite, Category = Materials)
44          FVector p5;
45          UPROPERTY(BlueprintReadWrite, Category = Materials)
46          FVector p6;
47          UPROPERTY(BlueprintReadWrite, Category = Materials)
48          FVector p7;
49
50          UPROPERTY(BlueprintReadWrite, Category = Materials)
51          FProceduralMeshVertex v0;
52          UPROPERTY(BlueprintReadWrite, Category = Materials)
53          FProceduralMeshVertex v1;
54          UPROPERTY(BlueprintReadWrite, Category = Materials)
55          FProceduralMeshVertex v2;
56          UPROPERTY(BlueprintReadWrite, Category = Materials)
57          FProceduralMeshVertex v3;
58          UPROPERTY(BlueprintReadWrite, Category = Materials)
59          FProceduralMeshVertex v4;
60          UPROPERTY(BlueprintReadWrite, Category = Materials)
61          FProceduralMeshVertex v5;
62          UPROPERTY(BlueprintReadWrite, Category = Materials)
63          FProceduralMeshVertex v6;
64          UPROPERTY(BlueprintReadWrite, Category = Materials)
65          FProceduralMeshVertex v7;
66
67          UPROPERTY(BlueprintReadWrite, Category = Materials)
68          UStaticMeshComponent* V0Sphere;
69          UPROPERTY(BlueprintReadWrite, Category = Materials)
70          UStaticMeshComponent* V0Sphere_Arrow0;
71          UPROPERTY(BlueprintReadWrite, Category = Materials)
72          UStaticMeshComponent* V0Sphere_Arrow1;
73          UPROPERTY(BlueprintReadWrite, Category = Materials)
74          UStaticMeshComponent* V0Sphere_Arrow2;
```

```
75          UPROPERTY(BlueprintReadWrite, Category = Materials)
76          UStaticMeshComponent* V1Sphere;
77          UPROPERTY(BlueprintReadWrite, Category = Materials)
78          UStaticMeshComponent* V1Sphere_Arrow0;
79          UPROPERTY(BlueprintReadWrite, Category = Materials)
80          UStaticMeshComponent* V1Sphere_Arrow1;
81          UPROPERTY(BlueprintReadWrite, Category = Materials)
82          UStaticMeshComponent* V1Sphere_Arrow2;
83          UPROPERTY(BlueprintReadWrite, Category = Materials)
84          UStaticMeshComponent* V2Sphere;
85          UPROPERTY(BlueprintReadWrite, Category = Materials)
86          UStaticMeshComponent* V2Sphere_Arrow0;
87          UPROPERTY(BlueprintReadWrite, Category = Materials)
88          UStaticMeshComponent* V2Sphere_Arrow1;
89          UPROPERTY(BlueprintReadWrite, Category = Materials)
90          UStaticMeshComponent* V2Sphere_Arrow2;
91          UPROPERTY(BlueprintReadWrite, Category = Materials)
92          UStaticMeshComponent* V3Sphere;
93          UPROPERTY(BlueprintReadWrite, Category = Materials)
94          UStaticMeshComponent* V3Sphere_Arrow0;
95          UPROPERTY(BlueprintReadWrite, Category = Materials)
96          UStaticMeshComponent* V3Sphere_Arrow1;
97          UPROPERTY(BlueprintReadWrite, Category = Materials)
98          UStaticMeshComponent* V3Sphere_Arrow2;
99          UPROPERTY(BlueprintReadWrite, Category = Materials)
100         UStaticMeshComponent* V4Sphere;
101         UPROPERTY(BlueprintReadWrite, Category = Materials)
102         UStaticMeshComponent* V4Sphere_Arrow0;
103         UPROPERTY(BlueprintReadWrite, Category = Materials)
104         UStaticMeshComponent* V4Sphere_Arrow1;
105         UPROPERTY(BlueprintReadWrite, Category = Materials)
106         UStaticMeshComponent* V4Sphere_Arrow2;
107         UPROPERTY(BlueprintReadWrite, Category = Materials)
108         UStaticMeshComponent* V5Sphere;
109         UPROPERTY(BlueprintReadWrite, Category = Materials)
110         UStaticMeshComponent* V5Sphere_Arrow0;
111         UPROPERTY(BlueprintReadWrite, Category = Materials)
112         UStaticMeshComponent* V5Sphere_Arrow1;
113         UPROPERTY(BlueprintReadWrite, Category = Materials)
114         UStaticMeshComponent* V5Sphere_Arrow2;
115         UPROPERTY(BlueprintReadWrite, Category = Materials)
116         UStaticMeshComponent* V6Sphere;
117         UPROPERTY(BlueprintReadWrite, Category = Materials)
118         UStaticMeshComponent* V6Sphere_Arrow0;
119         UPROPERTY(BlueprintReadWrite, Category = Materials)
120         UStaticMeshComponent* V6Sphere_Arrow1;
121         UPROPERTY(BlueprintReadWrite, Category = Materials)
122         UStaticMeshComponent* V6Sphere_Arrow2;
123         UPROPERTY(BlueprintReadWrite, Category = Materials)
124         UStaticMeshComponent* V7Sphere;
125         UPROPERTY(BlueprintReadWrite, Category = Materials)
126         UStaticMeshComponent* V7Sphere_Arrow0;
127         UPROPERTY(BlueprintReadWrite, Category = Materials)
128         UStaticMeshComponent* V7Sphere_Arrow1;
129         UPROPERTY(BlueprintReadWrite, Category = Materials)
```

```
130         UStaticMeshComponent* V7Sphere_Arrow2;
131
132         UPROPERTY(BlueprintReadWrite, Category = Materials)
133         UStaticMeshComponent* FrontFaceArrow;
134         UPROPERTY(BlueprintReadWrite, Category = Materials)
135         UStaticMeshComponent* BackFaceArrow;
136         UPROPERTY(BlueprintReadWrite, Category = Materials)
137         UStaticMeshComponent* LeftFaceArrow;
138         UPROPERTY(BlueprintReadWrite, Category = Materials)
139         UStaticMeshComponent* RightFaceArrow;
140         UPROPERTY(BlueprintReadWrite, Category = Materials)
141         UStaticMeshComponent* TopFaceArrow;
142         UPROPERTY(BlueprintReadWrite, Category = Materials)
143         UStaticMeshComponent* BottomFaceArrow;
144
145         UPROPERTY(BlueprintReadWrite, Category = Materials)
146         TArray<AProceduralCubeActor*> ExtrudedCubes;
147
148         // ---------- GENERATION FUNCTIONS ---------- \\
149
150         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
151         void GenerateCube(FVector StarterP0Location, float XSize, float
                YSize, float ZSize, FColor VtxsColor, APlayerController*
                GivenPController, bool IsExtruding);
152
153         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
154         void ModifyStaticMeshComponent(UStaticMeshComponent *
                GivenComponent, UStaticMesh * GivenStaticMesh, float GivenScale,
                 TArray<FName> GivenTag);
155
156         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
157         void GenerateCubePs(FVector P0Coords, float XSize, float YSize,
                float ZSize);
158
159         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
160         void GenerateCubeVs();
161
162         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
163         void UpdateVsSpheresLocations();
164
165         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
166         void UpdateFacesArrowsLocationsAndRotations();
167
168         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
169         void UpdateVertexArrowsLocationsAndRotations();
170
171         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
```

```
172         void SetCubeVColors(FColor VertexColor);
173
174         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
175         void GenerateCubeFaces(TArray<FProceduralMeshTriangle> &
                OutTriangles);
176
177         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
178         void GenerateCubeFace(FProceduralMeshVertex GivenV0,
                FProceduralMeshVertex GivenV1, FProceduralMeshVertex GivenV2,
                FProceduralMeshVertex GivenV3, FProceduralMeshTriangle& t1,
                FProceduralMeshTriangle& t2);
179
180         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
181         int32 ExtrusionFromGivenFaceVertexes(AProceduralCubeActor* NewCube,
                 TArray<FProceduralMeshVertex> FaceVertexes);
182
183         // ---------- VERTEX FUNCTIONS ---------- \\
184
185         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
186         FVector FindAndMoveVertex(FVector MovementDirection,
                FProceduralMeshVertex VertexToMove, TArray<
                FProceduralMeshTriangle>& CurrentTriangles);
187
188         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
189         void UpdateCubeVertexLocation(FProceduralMeshVertex VertexToUpdate)
                ;
190
191         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
192         int32 IdentifyFaceFromVertexes(FProceduralMeshVertex FVertex0,
                FProceduralMeshVertex FVertex1, FProceduralMeshVertex FVertex2,
                FProceduralMeshVertex FVertex3);
193
194         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
195         void MoveVertexAlongWorldAxis(FVector AxisOfMovement,
                UStaticMeshComponent* ClickedSphere, FProceduralMeshVertex
                VToMove, float MovementSign);
196
197         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
198         void MoveVertexAlongRotatedAxis(FVector AxisOfMovement,
                UStaticMeshComponent* ClickedSphere, FProceduralMeshVertex
                VToMove, float MovementSign);
199
200         // ---------- FACES and ARROWS FUNCTIONS ---------- \\
201
202         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
203              TArray<FProceduralMeshVertex> MoveFace(float MovementSign,
```

```
                    TArray<FProceduralMeshVertex> VertexesArray,
                    UStaticMeshComponent* FaceArrow);
204
205         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
206         FVector CalculateFaceMiddlePoint(TArray<FProceduralMeshVertex>
                FaceVertexes);
207
208         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
209         TArray<FProceduralMeshVertex> FindFaceVertexesFromArrowLocation(
                FVector ArrowLocation);
210
211         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
212         FRotator GetOrtogonalFaceDirectionFromFaceVertex(FVector
                GivenLocation, TArray<FProceduralMeshVertex> VertexesArray);
213
214         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
215         AProceduralCubeActor* ExtrudeFaceOfCube(UStaticMeshComponent*
                ClickedArrow);
216
217
218         // ---------- AUX FUNCTIONS ---------- \\
219
220         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
221         FVector UnitVector(FVector GivenVector);
222
223         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
224         float EuclideanDistance(FVector P, FVector Q);
225
226         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
227         float VectorModule(FVector GivenVector);
228
229         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
230         FVector CrossProd(FVector U, FVector V);
231
232         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
233         bool VectorContainsElement(TArray<int32> GivenVector, int32
                GivenElement);
234
235         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh")
236         FRotator ConvertToPitchRollYawRotator(FVector VToConvert);
237
238         // ---------- CUSTOM EVENTS ---------- \\
239
240         UFUNCTION(BlueprintNativeEvent, Category = SomeCategory)
241         void ArrowOnClickEvent(UStaticMeshComponent* ClickedArrow, bool
```

```
            KeepMoving, bool KeepExtrMov, float LMBMovDir, float RMBMovDir);
242
243         UFUNCTION()
244         void PlayerOnHover();
245
246         UFUNCTION()
247         void PlayerOnExitHover();
248
249         UFUNCTION()
250         void KeepingOnHover();
251
252         // ---------- HIDE FUNCTIONS ---------- \\
253
254         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
255         void HideVertexSpheres();
256
257         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
258         void HideFacesArrows();
259
260         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
261         void ShowFacesArrows();
262
263         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
264         void HideSpheresArrows();
265
266         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
267         void HideAllComponents();
268
269         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
270         void InitVertexMovementState(UStaticMeshComponent* SelectedSphere);
271
272         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
273         void QuitVertexMovementState();
274
275         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
276         void UnsetVertexMovementState();
277
278         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
279         void InitCubeFacesEditionState();
280
281         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
282         void QuitCubeFacesEditionState();
283
284         UFUNCTION(BlueprintCallable, Category = "Components|ProceduralMesh"
                )
```

```
285         void UnsetCubeFacesEditionState();
286
287         // ---------- OVERRIDE FUNCTIONS ---------- \\
288
289         void Tick(float deltaSeconds) override;
290 };
```

Código A.2: ProceduralCubeActor.cpp

```
1  #include "ProceduralMesh.h"
2  #include "ProceduralCubeActor.h"
3
4  AProceduralCubeActor::AProceduralCubeActor(const class
       FPostConstructInitializeProperties& PCIP)
5          : Super(PCIP)
6  {
7          this->SetActorTickEnabled(true);
8
9          KeepOnHover = false;
10         VertexMovementState = false;
11         CubeFacesEditionState = false;
12
13         // Define cube mesh and set it as cube root component
14         mesh = PCIP.CreateDefaultSubobject<UProceduralMeshComponent>(this,
               TEXT("ProceduralCube"));
15         RootComponent = mesh;
16
17         // Apply a simple material directly using the VertexColor as its
               BaseColor input
18         static ConstructorHelpers::FObjectFinder<UMaterialInterface>
               Material(TEXT("Material'/Game/Materials/BaseColor.BaseColor'"));
19
20         // Apply a real material with textures, using UVs
21         mesh->SetMaterial(0, Material.Object);
22
23         // Define SphereMeshComponents and ArrowMeshComponents
24         static ConstructorHelpers::FObjectFinder<UStaticMesh>
               SphereStaticMesh(TEXT("/Game/Sphere_Brush_StaticMesh.
               Sphere_Brush_StaticMesh"));
25         static ConstructorHelpers::FObjectFinder<UStaticMesh>
               ArrowStaticMesh(TEXT("/Game/SM_Arrow3.SM_Arrow3"));
26         float SpheresScale = 0.5; float FaceArrowsScale = 0.2; float
               VertexArrowScale = 0.3;
27         TArray<FName> VertexSphereTag, FaceArrowTag, VertexArrowTag;
28         VertexSphereTag.Add("VertexSphere"); FaceArrowTag.Add("FaceArrow");
               VertexArrowTag.Add("VertexArrow");
29
30         // Set vertexes spheres
31         V0Sphere = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(this,
               TEXT("V0Sphere")); ModifyStaticMeshComponent(V0Sphere,
               SphereStaticMesh.Object, SpheresScale, VertexSphereTag);
32         V1Sphere = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(this,
               TEXT("V1Sphere")); ModifyStaticMeshComponent(V1Sphere,
               SphereStaticMesh.Object, SpheresScale, VertexSphereTag);
```

```
33          V2Sphere = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(this,
                TEXT("V2Sphere")); ModifyStaticMeshComponent(V2Sphere,
                SphereStaticMesh.Object, SpheresScale, VertexSphereTag);
34          V3Sphere = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(this,
                TEXT("V3Sphere")); ModifyStaticMeshComponent(V3Sphere,
                SphereStaticMesh.Object, SpheresScale, VertexSphereTag);
35          V4Sphere = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(this,
                TEXT("V4Sphere")); ModifyStaticMeshComponent(V4Sphere,
                SphereStaticMesh.Object, SpheresScale, VertexSphereTag);
36          V5Sphere = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(this,
                TEXT("V5Sphere")); ModifyStaticMeshComponent(V5Sphere,
                SphereStaticMesh.Object, SpheresScale, VertexSphereTag);
37          V6Sphere = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(this,
                TEXT("V6Sphere")); ModifyStaticMeshComponent(V6Sphere,
                SphereStaticMesh.Object, SpheresScale, VertexSphereTag);
38          V7Sphere = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(this,
                TEXT("V7Sphere")); ModifyStaticMeshComponent(V7Sphere,
                SphereStaticMesh.Object, SpheresScale, VertexSphereTag);
39
40          // Faces arrows
41          FrontFaceArrow = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(
                this, TEXT("FrontFaceArrow"));       ModifyStaticMeshComponent(
                FrontFaceArrow, ArrowStaticMesh.Object, FaceArrowsScale,
                FaceArrowTag);
42          BackFaceArrow = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(
                this, TEXT("BackFaceArrow"));         ModifyStaticMeshComponent(
                BackFaceArrow, ArrowStaticMesh.Object, FaceArrowsScale,
                FaceArrowTag);
43          LeftFaceArrow = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(
                this, TEXT("LeftFaceArrow"));         ModifyStaticMeshComponent(
                LeftFaceArrow, ArrowStaticMesh.Object, FaceArrowsScale,
                FaceArrowTag);
44          RightFaceArrow = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(
                this, TEXT("RightFaceArrow"));       ModifyStaticMeshComponent(
                RightFaceArrow, ArrowStaticMesh.Object, FaceArrowsScale,
                FaceArrowTag);
45          TopFaceArrow = PCIP.CreateDefaultSubobject<UStaticMeshComponent>(
                this, TEXT("TopFaceArrow"));          ModifyStaticMeshComponent
                (TopFaceArrow, ArrowStaticMesh.Object, FaceArrowsScale,
                FaceArrowTag);
46          BottomFaceArrow = PCIP.CreateDefaultSubobject<UStaticMeshComponent
                >(this, TEXT("BottomFaceArrow"));     ModifyStaticMeshComponent(
                BottomFaceArrow, ArrowStaticMesh.Object, FaceArrowsScale,
                FaceArrowTag);
47
48          // Vertex arrows
49          V0Sphere_Arrow0 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
                >(this, TEXT("V0Sphere_Arrow0")); ModifyStaticMeshComponent(
                V0Sphere_Arrow0, ArrowStaticMesh.Object, VertexArrowScale,
                VertexArrowTag);
50          V0Sphere_Arrow1 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
                >(this, TEXT("V0Sphere_Arrow1")); ModifyStaticMeshComponent(
                V0Sphere_Arrow1, ArrowStaticMesh.Object, VertexArrowScale,
                VertexArrowTag);
51          V0Sphere_Arrow2 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
```

```
>(this, TEXT("V0Sphere_Arrow2")); ModifyStaticMeshComponent(
    V0Sphere_Arrow2, ArrowStaticMesh.Object, VertexArrowScale,
    VertexArrowTag);
V1Sphere_Arrow0 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
    >(this, TEXT("V1Sphere_Arrow0")); ModifyStaticMeshComponent(
    V1Sphere_Arrow0, ArrowStaticMesh.Object, VertexArrowScale,
    VertexArrowTag);
V1Sphere_Arrow1 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
    >(this, TEXT("V1Sphere_Arrow1")); ModifyStaticMeshComponent(
    V1Sphere_Arrow1, ArrowStaticMesh.Object, VertexArrowScale,
    VertexArrowTag);
V1Sphere_Arrow2 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
    >(this, TEXT("V1Sphere_Arrow2")); ModifyStaticMeshComponent(
    V1Sphere_Arrow2, ArrowStaticMesh.Object, VertexArrowScale,
    VertexArrowTag);
V2Sphere_Arrow0 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
    >(this, TEXT("V2Sphere_Arrow0")); ModifyStaticMeshComponent(
    V2Sphere_Arrow0, ArrowStaticMesh.Object, VertexArrowScale,
    VertexArrowTag);
V2Sphere_Arrow1 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
    >(this, TEXT("V2Sphere_Arrow1")); ModifyStaticMeshComponent(
    V2Sphere_Arrow1, ArrowStaticMesh.Object, VertexArrowScale,
    VertexArrowTag);
V2Sphere_Arrow2 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
    >(this, TEXT("V2Sphere_Arrow2")); ModifyStaticMeshComponent(
    V2Sphere_Arrow2, ArrowStaticMesh.Object, VertexArrowScale,
    VertexArrowTag);
V3Sphere_Arrow0 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
    >(this, TEXT("V3Sphere_Arrow0")); ModifyStaticMeshComponent(
    V3Sphere_Arrow0, ArrowStaticMesh.Object, VertexArrowScale,
    VertexArrowTag);
V3Sphere_Arrow1 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
    >(this, TEXT("V3Sphere_Arrow1")); ModifyStaticMeshComponent(
    V3Sphere_Arrow1, ArrowStaticMesh.Object, VertexArrowScale,
    VertexArrowTag);
V3Sphere_Arrow2 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
    >(this, TEXT("V3Sphere_Arrow2")); ModifyStaticMeshComponent(
    V3Sphere_Arrow2, ArrowStaticMesh.Object, VertexArrowScale,
    VertexArrowTag);
V4Sphere_Arrow0 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
    >(this, TEXT("V4Sphere_Arrow0")); ModifyStaticMeshComponent(
    V4Sphere_Arrow0, ArrowStaticMesh.Object, VertexArrowScale,
    VertexArrowTag);
V4Sphere_Arrow1 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
    >(this, TEXT("V4Sphere_Arrow1")); ModifyStaticMeshComponent(
    V4Sphere_Arrow1, ArrowStaticMesh.Object, VertexArrowScale,
    VertexArrowTag);
V4Sphere_Arrow2 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
    >(this, TEXT("V4Sphere_Arrow2")); ModifyStaticMeshComponent(
    V4Sphere_Arrow2, ArrowStaticMesh.Object, VertexArrowScale,
    VertexArrowTag);
V5Sphere_Arrow0 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
    >(this, TEXT("V5Sphere_Arrow0")); ModifyStaticMeshComponent(
    V5Sphere_Arrow0, ArrowStaticMesh.Object, VertexArrowScale,
    VertexArrowTag);
```

```
65          V5Sphere_Arrow1 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
                >(this, TEXT("V5Sphere_Arrow1")); ModifyStaticMeshComponent(
                V5Sphere_Arrow1, ArrowStaticMesh.Object, VertexArrowScale,
                VertexArrowTag);
66          V5Sphere_Arrow2 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
                >(this, TEXT("V5Sphere_Arrow2")); ModifyStaticMeshComponent(
                V5Sphere_Arrow2, ArrowStaticMesh.Object, VertexArrowScale,
                VertexArrowTag);
67          V6Sphere_Arrow0 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
                >(this, TEXT("V6Sphere_Arrow0")); ModifyStaticMeshComponent(
                V6Sphere_Arrow0, ArrowStaticMesh.Object, VertexArrowScale,
                VertexArrowTag);
68          V6Sphere_Arrow1 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
                >(this, TEXT("V6Sphere_Arrow1")); ModifyStaticMeshComponent(
                V6Sphere_Arrow1, ArrowStaticMesh.Object, VertexArrowScale,
                VertexArrowTag);
69          V6Sphere_Arrow2 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
                >(this, TEXT("V6Sphere_Arrow2")); ModifyStaticMeshComponent(
                V6Sphere_Arrow2, ArrowStaticMesh.Object, VertexArrowScale,
                VertexArrowTag);
70          V7Sphere_Arrow0 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
                >(this, TEXT("V7Sphere_Arrow0")); ModifyStaticMeshComponent(
                V7Sphere_Arrow0, ArrowStaticMesh.Object, VertexArrowScale,
                VertexArrowTag);
71          V7Sphere_Arrow1 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
                >(this, TEXT("V7Sphere_Arrow1")); ModifyStaticMeshComponent(
                V7Sphere_Arrow1, ArrowStaticMesh.Object, VertexArrowScale,
                VertexArrowTag);
72          V7Sphere_Arrow2 = PCIP.CreateDefaultSubobject<UStaticMeshComponent
                >(this, TEXT("V7Sphere_Arrow2")); ModifyStaticMeshComponent(
                V7Sphere_Arrow2, ArrowStaticMesh.Object, VertexArrowScale,
                VertexArrowTag);
73
74          HideAllComponents();
75  }
76
77  void AProceduralCubeActor::ModifyStaticMeshComponent(UStaticMeshComponent*
        GivenComponent, UStaticMesh* GivenStaticMesh, float GivenScale, TArray<
        FName> GivenTag)
78  {
79          GivenComponent->StaticMesh = GivenStaticMesh;
80          GivenComponent->SetWorldScale3D(FVector(GivenScale, GivenScale,
                GivenScale));
81          GivenComponent->AttachTo(mesh);
82          GivenComponent->ComponentTags = GivenTag;
83  }
84
85  void AProceduralCubeActor::GenerateCube(FVector StarterP0Location, float
        XSize, float YSize, float ZSize, FColor VtxsColor, APlayerController*
        GivenPController, bool IsExtruding)
86  {
87          // Enable cursor on hover
88          this->OnBeginCursorOver.AddDynamic(this, &AProceduralCubeActor::
                PlayerOnHover);
89          this->OnEndCursorOver.AddDynamic(this, &AProceduralCubeActor::
```

```
            PlayerOnExitHover);

        // Set Player Controller
        CustomPController = Cast<ACustomPlayerController>(GivenPController)
            ;

        // P variables
        if (!IsExtruding) { GenerateCubePs(StarterP0Location, XSize, YSize,
             ZSize); }

        // Vertexes
        GenerateCubeVs();
        SetCubeVColors(VtxsColor);
        UpdateVsSpheresLocations();
        UpdateVertexArrowsLocationsAndRotations();

        // Faces and Mesh triangles
        TArray<FProceduralMeshTriangle> OutTriangles;
        GenerateCubeFaces(OutTriangles);
        UpdateFacesArrowsLocationsAndRotations();

        // Update cube image at World
        mesh->SetProceduralMeshTriangles(OutTriangles);
}

void AProceduralCubeActor::GenerateCubePs(FVector P0Coords, float XSize,
    float YSize, float ZSize)
{
        p0 = P0Coords + FVector(0.f, 0.f, 0.f);                 p1 =
            P0Coords + FVector(0.f, 0.f, ZSize);
        p2 = P0Coords + FVector(XSize, 0.f, ZSize);             p3 =
            P0Coords + FVector(XSize, 0.f, 0.f);
        p4 = P0Coords + FVector(XSize, YSize, 0.f);             p5 =
            P0Coords + FVector(XSize, YSize, ZSize);
        p6 = P0Coords + FVector(0.f, YSize, ZSize);             p7 =
            P0Coords + FVector(0.f, YSize, 0.f);
}

void AProceduralCubeActor::GenerateCubeVs()
{
        v0.Position = p0; v0.Id = 0;    v1.Position = p1; v1.Id = 1;
        v2.Position = p2; v2.Id = 2;    v3.Position = p3; v3.Id = 3;
        v4.Position = p4; v4.Id = 4;    v5.Position = p5; v5.Id = 5;
        v6.Position = p6; v6.Id = 6;    v7.Position = p7; v7.Id = 7;
}

void AProceduralCubeActor::UpdateVsSpheresLocations()
{
        V0Sphere->SetRelativeLocation(p0);      V1Sphere->
            SetRelativeLocation(p1);
        V2Sphere->SetRelativeLocation(p2);      V3Sphere->
            SetRelativeLocation(p3);
        V4Sphere->SetRelativeLocation(p4);      V5Sphere->
            SetRelativeLocation(p5);
        V6Sphere->SetRelativeLocation(p6);      V7Sphere->
```

```
            SetRelativeLocation(p7);
}

void AProceduralCubeActor::UpdateFacesArrowsLocationsAndRotations()
{
        // Set cube faces vertexes arrays
        TArray<FProceduralMeshVertex> FrontFace, BackFace, LeftFace,
            RightFace, TopFace, BottomFace;
        FrontFace.Add(v0);              FrontFace.Add(v1);
            FrontFace.Add(v2);              FrontFace.Add(v3);
        BackFace.Add(v4);               BackFace.Add(v5);
            BackFace.Add(v6);               BackFace.Add(v7);
        LeftFace.Add(v7);               LeftFace.Add(v6);
            LeftFace.Add(v1);               LeftFace.Add(v0);
        RightFace.Add(v3);              RightFace.Add(v2);
            RightFace.Add(v5);              RightFace.Add(v4);
        TopFace.Add(v1);                TopFace.Add(v6);
            TopFace.Add(v5);                TopFace.Add(v2);
        BottomFace.Add(v3);             BottomFace.Add(v4);
            BottomFace.Add(v7);             BottomFace.Add(v0);

        // Update Faces Arrows locations and rotations
        FVector FrontFaceArrowLocation = CalculateFaceMiddlePoint(FrontFace
            );
        FrontFaceArrow->SetRelativeLocationAndRotation(
            FrontFaceArrowLocation, FRotator(90, 90, 90));

        FVector BackFaceArrowLocation = CalculateFaceMiddlePoint(BackFace);
        BackFaceArrow->SetRelativeLocationAndRotation(BackFaceArrowLocation
            , FRotator(-90, -90, -90));

        FVector LeftFaceArrowLocation = CalculateFaceMiddlePoint(LeftFace);
        LeftFaceArrow->SetRelativeLocationAndRotation(LeftFaceArrowLocation
            , FRotator(0, -90, 0));

        FVector RightFaceArrowLocation = CalculateFaceMiddlePoint(RightFace
            );
        RightFaceArrow->SetRelativeLocationAndRotation(
            RightFaceArrowLocation, FRotator(0, 90, 0));

        FVector TopFaceArrowLocation = CalculateFaceMiddlePoint(TopFace);
        TopFaceArrow->SetRelativeLocationAndRotation(TopFaceArrowLocation,
            FRotator(0, 0, 90));

        FVector BottomFaceArrowLocation = CalculateFaceMiddlePoint(
            BottomFace);
        BottomFaceArrow->SetRelativeLocationAndRotation(
            BottomFaceArrowLocation, FRotator(0, 0, -90));
}

void AProceduralCubeActor::UpdateVertexArrowsLocationsAndRotations()
{
        // Set 'natural' directions
        FRotator XDirection = FRotator::MakeFromEuler(FVector(90, 0, 0));
        FRotator YDirection = FRotator::MakeFromEuler(FVector(0, 90, 0));
```

```
172         FRotator ZDirection = FRotator::MakeFromEuler(FVector(0, 0, 90));
173 
174         // Update Vertex Arrows locations
175         V0Sphere_Arrow0->SetRelativeLocation(p0);
               V0Sphere_Arrow1->SetRelativeLocation(p0);
               V0Sphere_Arrow2->SetRelativeLocation(p0);
176         V0Sphere_Arrow0->SetWorldRotation(XDirection);  V0Sphere_Arrow1->
               SetWorldRotation(YDirection);  V0Sphere_Arrow2->SetWorldRotation
               (ZDirection);
177 
178         V1Sphere_Arrow0->SetRelativeLocation(p1);
               V1Sphere_Arrow1->SetRelativeLocation(p1);
               V1Sphere_Arrow2->SetRelativeLocation(p1);
179         V1Sphere_Arrow0->SetWorldRotation(XDirection);  V1Sphere_Arrow1->
               SetWorldRotation(YDirection);  V1Sphere_Arrow2->SetWorldRotation
               (ZDirection);
180 
181         V2Sphere_Arrow0->SetRelativeLocation(p2);
               V2Sphere_Arrow1->SetRelativeLocation(p2);
               V2Sphere_Arrow2->SetRelativeLocation(p2);
182         V2Sphere_Arrow0->SetWorldRotation(XDirection);  V2Sphere_Arrow1->
               SetWorldRotation(YDirection);  V2Sphere_Arrow2->SetWorldRotation
               (ZDirection);
183 
184         V3Sphere_Arrow0->SetRelativeLocation(p3);
               V3Sphere_Arrow1->SetRelativeLocation(p3);
               V3Sphere_Arrow2->SetRelativeLocation(p3);
185         V3Sphere_Arrow0->SetWorldRotation(XDirection);  V3Sphere_Arrow1->
               SetWorldRotation(YDirection);  V3Sphere_Arrow2->SetWorldRotation
               (ZDirection);
186 
187         V4Sphere_Arrow0->SetRelativeLocation(p4);
               V4Sphere_Arrow1->SetRelativeLocation(p4);
               V4Sphere_Arrow2->SetRelativeLocation(p4);
188         V4Sphere_Arrow0->SetWorldRotation(XDirection);  V4Sphere_Arrow1->
               SetWorldRotation(YDirection);  V4Sphere_Arrow2->SetWorldRotation
               (ZDirection);
189 
190         V5Sphere_Arrow0->SetRelativeLocation(p5);
               V5Sphere_Arrow1->SetRelativeLocation(p5);
               V5Sphere_Arrow2->SetRelativeLocation(p5);
191         V5Sphere_Arrow0->SetWorldRotation(XDirection);  V5Sphere_Arrow1->
               SetWorldRotation(YDirection);  V5Sphere_Arrow2->SetWorldRotation
               (ZDirection);
192 
193         V6Sphere_Arrow0->SetRelativeLocation(p6);
               V6Sphere_Arrow1->SetRelativeLocation(p6);
               V6Sphere_Arrow2->SetRelativeLocation(p6);
194         V6Sphere_Arrow0->SetWorldRotation(XDirection);  V6Sphere_Arrow1->
               SetWorldRotation(YDirection);  V6Sphere_Arrow2->SetWorldRotation
               (ZDirection);
195 
196         V7Sphere_Arrow0->SetRelativeLocation(p7);
               V7Sphere_Arrow1->SetRelativeLocation(p7);
               V7Sphere_Arrow2->SetRelativeLocation(p7);
```

```
197         V7Sphere_Arrow0->SetWorldRotation(XDirection);  V7Sphere_Arrow1->
                SetWorldRotation(YDirection);  V7Sphere_Arrow2->SetWorldRotation
                (ZDirection);
198 }
199
200 void AProceduralCubeActor::SetCubeVColors(FColor VertexColor)
201 {
202         v0.Color = v1.Color = v2.Color = v3.Color = v4.Color = v5.Color =
                v6.Color = v7.Color = VertexColor;
203 }
204
205 void AProceduralCubeActor::GenerateCubeFaces(TArray<FProceduralMeshTriangle
        > & OutTriangles)
206 {
207         FProceduralMeshTriangle t1, t2;
208
209         // front face
210         GenerateCubeFace(v0, v1, v2, v3, t1, t2);       OutTriangles.Add(t1
                );   OutTriangles.Add(t2);
211
212         // back face
213         GenerateCubeFace(v4, v5, v6, v7, t1, t2);       OutTriangles.Add(t1
                );   OutTriangles.Add(t2);
214
215         // left face
216         GenerateCubeFace(v7, v6, v1, v0, t1, t2);       OutTriangles.Add(t1
                );   OutTriangles.Add(t2);
217
218         // right face
219         GenerateCubeFace(v3, v2, v5, v4, t1, t2);       OutTriangles.Add(t1
                );   OutTriangles.Add(t2);
220
221         // top face
222         GenerateCubeFace(v1, v6, v5, v2, t1, t2);       OutTriangles.Add(t1
                );   OutTriangles.Add(t2);
223
224         // bottom face
225         GenerateCubeFace(v3, v4, v7, v0, t1, t2);       OutTriangles.Add(t1
                );   OutTriangles.Add(t2);
226 }
227
228 void AProceduralCubeActor::GenerateCubeFace(FProceduralMeshVertex GivenV0,
        FProceduralMeshVertex GivenV1, FProceduralMeshVertex GivenV2,
        FProceduralMeshVertex GivenV3, FProceduralMeshTriangle& t1,
        FProceduralMeshTriangle& t2)
229 {
230         // Set vertexes components U, V
231         GivenV0.U = 0.f;        GivenV1.U = 0.f;        GivenV2.U = 1.f;
                    GivenV3.U = 1.f;
232         GivenV0.V = 0.f;        GivenV1.V = 1.f;        GivenV2.V = 1.f;
                    GivenV3.V = 0.f;
233
234         // Face triangle t1
235         t1.Vertex0 = GivenV0;   t1.Vertex1 = GivenV1;   t1.Vertex2 =
                GivenV2;
```

```
236
237         // Face triangle t2
238         t2.Vertex0 = GivenV0;   t2.Vertex1 = GivenV2;   t2.Vertex2 =
                GivenV3;
239 }
240
241 int32 AProceduralCubeActor::ExtrusionFromGivenFaceVertexes(
        AProceduralCubeActor* NewCube, TArray<FProceduralMeshVertex>
        FaceVertexes)
242 {
243         // Set new cube location and rotation (the same that its parent
                cube has)
244         NewCube->SetActorRotation(this->GetActorRotation());
245         NewCube->SetActorLocation(this->GetActorLocation());
246
247         // The extruded cube will have its parent cube edges sizes except
                one of them which will be smaller (the one in the orthogonal
                face direction) in order to extrude a smaller cube
248         FVector ParentEdgesSizes = FVector(abs(p3.X - p0.X), abs(p7.Y - p0.
                Y), abs(p1.Z - p0.Z));
249         float NewCubeSize = 10;
250         FVector CrossProdVector = UnitVector(CrossProd(FaceVertexes[1].
                Position - FaceVertexes[0].Position, FaceVertexes[2].Position -
                FaceVertexes[0].Position));
251         FVector NewCubeEdgesSizes = ParentEdgesSizes - ParentEdgesSizes *
                FVector(abs(CrossProdVector[0]), abs(CrossProdVector[1]), abs(
                CrossProdVector[2])) + NewCubeSize * FVector(abs(CrossProdVector
                [0]), abs(CrossProdVector[1]), abs(CrossProdVector[2]));
252
253         // Integer representing which arrow will be moved after extruding (
                from 0 to 5)
254         int32 ArrowIdToMoveAfterExtruding = IdentifyFaceFromVertexes(
                FaceVertexes[0], FaceVertexes[1], FaceVertexes[2], FaceVertexes
                [3]);
255
256         // Calculate new cube P0 coordinates
257         FVector NewCubeP0Coords, NewCubeP1Coords, NewCubeP2Coords,
                NewCubeP3Coords, NewCubeP4Coords, NewCubeP5Coords,
                NewCubeP6Coords, NewCubeP7Coords;
258
259         // Calculate new cube p's coordinates
260         switch (ArrowIdToMoveAfterExtruding)
261         {
262                 // Front face
263                 case 0:
264                         NewCube->p0 = FVector(p0.X, p0.Y -
                                NewCubeEdgesSizes[1], p0.Z);
265                         NewCube->p1 = FVector(p1.X, p1.Y -
                                NewCubeEdgesSizes[1], p1.Z);
266                         NewCube->p2 = FVector(p2.X, p2.Y -
                                NewCubeEdgesSizes[1], p2.Z);
267                         NewCube->p3 = FVector(p3.X, p3.Y -
                                NewCubeEdgesSizes[1], p3.Z);
268                         NewCube->p4 = p3;
269                         NewCube->p5 = p2;
```

```
270                         NewCube->p6 = p1;
271                         NewCube->p7 = p0;
272                         break;
273 
274                 // Back face
275                 case 1:
276                         NewCube->p0 = p7;
277                         NewCube->p1 = p6;
278                         NewCube->p2 = p5;
279                         NewCube->p3 = p4;
280                         NewCube->p4 = FVector(p4.X, p4.Y +
                               NewCubeEdgesSizes[1], p4.Z);
281                         NewCube->p5 = FVector(p5.X, p5.Y +
                               NewCubeEdgesSizes[1], p5.Z);
282                         NewCube->p6 = FVector(p6.X, p6.Y +
                               NewCubeEdgesSizes[1], p6.Z);
283                         NewCube->p7 = FVector(p7.X, p7.Y +
                               NewCubeEdgesSizes[1], p7.Z);
284                         break;
285 
286                 // Left face
287                 case 2:
288                         NewCube->p0 = FVector(p0.X - NewCubeEdgesSizes[0],
                               p0.Y, p0.Z);
289                         NewCube->p1 = FVector(p1.X - NewCubeEdgesSizes[0],
                               p1.Y, p1.Z);
290                         NewCube->p2 = p1;
291                         NewCube->p3 = p0;
292                         NewCube->p4 = p7;
293                         NewCube->p5 = p6;
294                         NewCube->p6 = FVector(p6.X - NewCubeEdgesSizes[0],
                               p6.Y, p6.Z);
295                         NewCube->p7 = FVector(p7.X - NewCubeEdgesSizes[0],
                               p7.Y, p7.Z);
296                         break;
297 
298                 // Right face
299                 case 3:
300                         NewCube->p0 = p3;
301                         NewCube->p1 = p2;
302                         NewCube->p2 = FVector(p2.X + NewCubeEdgesSizes[0],
                               p2.Y, p2.Z);
303                         NewCube->p3 = FVector(p3.X + NewCubeEdgesSizes[0],
                               p3.Y, p3.Z);;
304                         NewCube->p4 = FVector(p4.X + NewCubeEdgesSizes[0],
                               p4.Y, p4.Z);
305                         NewCube->p5 = FVector(p5.X + NewCubeEdgesSizes[0],
                               p5.Y, p5.Z);;
306                         NewCube->p6 = p5;
307                         NewCube->p7 = p4;
308                         break;
309 
310                 // Top face
311                 case 4:
312                         NewCube->p0 = p1;
```

```
313                             NewCube->p1 = FVector(p1.X, p1.Y, p1.Z +
                                    NewCubeEdgesSizes[2]);
314                             NewCube->p2 = FVector(p2.X, p2.Y, p2.Z +
                                    NewCubeEdgesSizes[2]);
315                             NewCube->p3 = p2;
316                             NewCube->p4 = p5;
317                             NewCube->p5 = FVector(p5.X, p5.Y, p5.Z +
                                    NewCubeEdgesSizes[2]);
318                             NewCube->p6 = FVector(p6.X, p6.Y, p6.Z +
                                    NewCubeEdgesSizes[2]);
319                             NewCube->p7 = p6;
320                             break;
321
322                     // Bottom face
323                     case 5:
324                             NewCube->p0 = FVector(p0.X, p0.Y, p0.Z -
                                    NewCubeEdgesSizes[2]);
325                             NewCube->p1 = p0;
326                             NewCube->p2 = p3;
327                             NewCube->p3 = FVector(p3.X, p3.Y, p3.Z -
                                    NewCubeEdgesSizes[2]);
328                             NewCube->p4 = FVector(p4.X, p4.Y, p4.Z -
                                    NewCubeEdgesSizes[2]);
329                             NewCube->p5 = p4;
330                             NewCube->p6 = p7;
331                             NewCube->p7 = FVector(p7.X, p7.Y, p7.Z -
                                    NewCubeEdgesSizes[2]);
332                             break;
333
334                     default:
335                             break;
336             }
337
338             // Generate new cube
339             NewCube->GenerateCube(NewCubeP0Coords, NewCubeEdgesSizes[0],
                    NewCubeEdgesSizes[1], NewCubeEdgesSizes[2], FColor::Red,
                    CustomPController, true);
340
341             // Add new cube to parent's cube extruded cubes list
342             ExtrudedCubes.Add(NewCube);
343
344             NewCube->ShowFacesArrows();
345
346             // Return integer representing the arrow to move after creating the
                     new cube
347             return ArrowIdToMoveAfterExtruding;
348     }
349
350     FVector AProceduralCubeActor::FindAndMoveVertex(FVector MovementDirection,
        FProceduralMeshVertex VertexToMove, TArray<FProceduralMeshTriangle>&
        CurrentTriangles)
351     {
352             FVector NewVertexPosition = VertexToMove.Position;
353
354             bool found = false;
```

```
355         for (int i = 0; i < CurrentTriangles.Num(); i++)
356         {
357                 if (CurrentTriangles[i].Vertex0.Id == VertexToMove.Id)
358                 {
359                         if (!found) { found = true; NewVertexPosition =
                                CurrentTriangles[i].Vertex0.Position +
                                MovementDirection; }
360                         CurrentTriangles[i].Vertex0.Position =
                                NewVertexPosition;
361                 }
362                 else if (CurrentTriangles[i].Vertex1.Id == VertexToMove.Id)
363                 {
364                         if (!found) { found = true; NewVertexPosition =
                                CurrentTriangles[i].Vertex1.Position +
                                MovementDirection; }
365                         CurrentTriangles[i].Vertex1.Position =
                                NewVertexPosition;
366                 }
367                 else if (CurrentTriangles[i].Vertex2.Id == VertexToMove.Id)
368                 {
369                         if (!found) { found = true; NewVertexPosition =
                                CurrentTriangles[i].Vertex2.Position +
                                MovementDirection; }
370                         CurrentTriangles[i].Vertex2.Position =
                                NewVertexPosition;
371                 }
372         }
373         VertexToMove.Position = NewVertexPosition;
374         UpdateCubeVertexLocation(VertexToMove);
375         UpdateVertexArrowsLocationsAndRotations();
376         UpdateFacesArrowsLocationsAndRotations();
377         return NewVertexPosition;
378 }
379
380 void AProceduralCubeActor::UpdateCubeVertexLocation(FProceduralMeshVertex
       VertexToUpdate)
381 {
382         if (v0.Id == VertexToUpdate.Id)          { v0.Position =
                VertexToUpdate.Position; p0 = VertexToUpdate.Position; }
383         else if (v1.Id == VertexToUpdate.Id) { v1.Position = VertexToUpdate
                .Position; p1 = VertexToUpdate.Position; }
384         else if (v2.Id == VertexToUpdate.Id) { v2.Position = VertexToUpdate
                .Position; p2 = VertexToUpdate.Position; }
385         else if (v3.Id == VertexToUpdate.Id) { v3.Position = VertexToUpdate
                .Position; p3 = VertexToUpdate.Position; }
386         else if (v4.Id == VertexToUpdate.Id) { v4.Position = VertexToUpdate
                .Position; p4 = VertexToUpdate.Position; }
387         else if (v5.Id == VertexToUpdate.Id) { v5.Position = VertexToUpdate
                .Position; p5 = VertexToUpdate.Position; }
388         else if (v6.Id == VertexToUpdate.Id) { v6.Position = VertexToUpdate
                .Position; p6 = VertexToUpdate.Position; }
389         else if (v7.Id == VertexToUpdate.Id) { v7.Position = VertexToUpdate
                .Position; p7 = VertexToUpdate.Position; }
390
391         UpdateVsSpheresLocations();
```

```
392         UpdateVertexArrowsLocationsAndRotations();
393 }
394
395 int32 AProceduralCubeActor::IdentifyFaceFromVertexes(FProceduralMeshVertex
    FVertex0, FProceduralMeshVertex FVertex1, FProceduralMeshVertex FVertex2
    , FProceduralMeshVertex FVertex3)
396 {
397         // Set cube faces vertexes arrays
398         TArray<int32> FrontFace;        FrontFace.Add(v0.Id);   FrontFace.
            Add(v1.Id);   FrontFace.Add(v2.Id);   FrontFace.Add(v3.Id);
399         TArray<int32> BackFace;         BackFace.Add(v4.Id);    BackFace.
            Add(v5.Id);    BackFace.Add(v6.Id);    BackFace.Add(v7.Id);
400         TArray<int32> LeftFace;         LeftFace.Add(v7.Id);    LeftFace.
            Add(v6.Id);    LeftFace.Add(v1.Id);    LeftFace.Add(v0.Id);
401         TArray<int32> RightFace;        RightFace.Add(v3.Id);   RightFace.
            Add(v2.Id);   RightFace.Add(v5.Id);   RightFace.Add(v4.Id);
402         TArray<int32> TopFace;          TopFace.Add(v1.Id);
            TopFace.Add(v6.Id);             TopFace.Add(v5.Id);
            TopFace.Add(v2.Id);
403         TArray<int32> BottomFace;       BottomFace.Add(v3.Id);  BottomFace.
            Add(v4.Id);  BottomFace.Add(v7.Id);  BottomFace.Add(v0.Id);
404
405         // If given vertexes are FRONT FACE vertexes, 0 will be returned
406         if (VectorContainsElement(FrontFace, FVertex0.Id) &&
            VectorContainsElement(FrontFace, FVertex1.Id) &&
            VectorContainsElement(FrontFace, FVertex2.Id) &&
            VectorContainsElement(FrontFace, FVertex3.Id)) { return 0; }
407
408         // If given vertexes are BACK FACE vertexes, 1 will be returned
409         else if (VectorContainsElement(BackFace, FVertex0.Id) &&
            VectorContainsElement(BackFace, FVertex1.Id) &&
            VectorContainsElement(BackFace, FVertex2.Id) &&
            VectorContainsElement(BackFace, FVertex3.Id)) { return 1; }
410
411         // If given vertexes are LEFT FACE vertexes, 2 will be returned
412         else if (VectorContainsElement(LeftFace, FVertex0.Id) &&
            VectorContainsElement(LeftFace, FVertex1.Id) &&
            VectorContainsElement(LeftFace, FVertex2.Id) &&
            VectorContainsElement(LeftFace, FVertex3.Id)) { return 2; }
413
414         // If given vertexes are RIGHT FACE vertexes, 3 will be returned
415         else if (VectorContainsElement(RightFace, FVertex0.Id) &&
            VectorContainsElement(RightFace, FVertex1.Id) &&
            VectorContainsElement(RightFace, FVertex2.Id) &&
            VectorContainsElement(RightFace, FVertex3.Id)) { return 3; }
416
417         // If given vertexes are TOP FACE vertexes, 4 will be returned
418         else if (VectorContainsElement(TopFace, FVertex0.Id) &&
            VectorContainsElement(TopFace, FVertex1.Id) &&
            VectorContainsElement(TopFace, FVertex2.Id) &&
            VectorContainsElement(TopFace, FVertex3.Id)) { return 4; }
419
420         // If given vertexes are BOTTOM FACE vertexes, 5 will be returned
421         else if (VectorContainsElement(BottomFace, FVertex0.Id) &&
            VectorContainsElement(BottomFace, FVertex1.Id) &&
```

```
            VectorContainsElement(BottomFace, FVertex2.Id) &&
            VectorContainsElement(BottomFace, FVertex3.Id)) { return 5; }
422
423         // If given vertexes doesn't match any face vertexes, -1 will be
                returned
424         else { return -1; }
425 }
426
427 void AProceduralCubeActor::MoveVertexAlongWorldAxis(FVector AxisOfMovement,
        UStaticMeshComponent* ClickedSphere, FProceduralMeshVertex VToMove,
    float MovementSign)
428 {
429         if (MovementSign != 0 && (AxisOfMovement.Equals(FVector(1, 0, 0))
                || AxisOfMovement.Equals(FVector(0, 1, 0)) || AxisOfMovement.
                Equals(FVector(0, 0, 1))))
430         {
431                 if (MovementSign < 0) { MovementSign = -1; }
432                 else { MovementSign = 1; }
433
434                 ClickedSphere->SetWorldLocation(ClickedSphere->
                        GetComponentLocation() + AxisOfMovement * MovementSign);
435
436                 TArray<FProceduralMeshTriangle> CurrentTriangles = mesh->
                        GetProceduralMeshTriangles();
437
438                 for (int i = 0; i < CurrentTriangles.Num(); i++)
439                 {
440                         if (CurrentTriangles[i].Vertex0.Id == VToMove.Id)
                                                 {       CurrentTriangles[i].
                                Vertex0.Position = ClickedSphere->
                                RelativeLocation; }
441                         else if (CurrentTriangles[i].Vertex1.Id == VToMove.
                                Id)  {       CurrentTriangles[i].Vertex1.
                                Position = ClickedSphere->RelativeLocation; }
442                         else if (CurrentTriangles[i].Vertex2.Id == VToMove.
                                Id)  {       CurrentTriangles[i].Vertex2.
                                Position = ClickedSphere->RelativeLocation; }
443                 }
444
445                 VToMove.Position = ClickedSphere->RelativeLocation;
446                 UpdateCubeVertexLocation(VToMove);
447                 UpdateVsSpheresLocations();
448                 UpdateVertexArrowsLocationsAndRotations();
449                 mesh->SetProceduralMeshTriangles(CurrentTriangles);
450         }
451 }
452
453 void AProceduralCubeActor::MoveVertexAlongRotatedAxis(FVector
        AxisOfMovement, UStaticMeshComponent* ClickedSphere,
        FProceduralMeshVertex VToMove, float MovementSign)
454 {
455         if (MovementSign != 0 && (AxisOfMovement.Equals(FVector(1, 0, 0))
                || AxisOfMovement.Equals(FVector(0, 1, 0)) || AxisOfMovement.
                Equals(FVector(0, 0, 1))))
456         {
```

```
457                 if (MovementSign < 0) { MovementSign = -1; }
458                 else { MovementSign = 1; }
459 
460                 ClickedSphere->SetRelativeLocation(ClickedSphere->
                        RelativeLocation + AxisOfMovement * MovementSign);
461 
462                 TArray<FProceduralMeshTriangle> CurrentTriangles = mesh->
                        GetProceduralMeshTriangles();
463 
464                 for (int i = 0; i < CurrentTriangles.Num(); i++)
465                 {
466                         if (CurrentTriangles[i].Vertex0.Id == VToMove.Id)
                                                { CurrentTriangles[i].Vertex0.
                                Position = ClickedSphere->RelativeLocation; }
467                         else if (CurrentTriangles[i].Vertex1.Id == VToMove.
                                Id)  { CurrentTriangles[i].Vertex1.Position =
                                ClickedSphere->RelativeLocation; }
468                         else if (CurrentTriangles[i].Vertex2.Id == VToMove.
                                Id)  { CurrentTriangles[i].Vertex2.Position =
                                ClickedSphere->RelativeLocation; }
469                 }
470 
471                 VToMove.Position = ClickedSphere->RelativeLocation;
472                 UpdateCubeVertexLocation(VToMove);
473                 UpdateVsSpheresLocations();
474                 UpdateVertexArrowsLocationsAndRotations();
475                 mesh->SetProceduralMeshTriangles(CurrentTriangles);
476         }
477 }
478 
479 TArray<FProceduralMeshVertex> AProceduralCubeActor::MoveFace(float
        MovementSign, TArray<FProceduralMeshVertex> VertexesArray,
        UStaticMeshComponent* FaceArrow)
480 {
481         // Calculate movement direction (orthogonal vector)
482         FVector v0v1 = VertexesArray[1].Position - VertexesArray[0].
                Position;
483         FVector v1v2 = VertexesArray[2].Position - VertexesArray[1].
                Position;
484         FVector Direction = UnitVector(v0v1.CrossProduct(v0v1, v1v2));
485 
486         //FVector Direction;
487         FRotator ArrowRot = FaceArrow->RelativeRotation;
488         if (85 < abs(ArrowRot.Yaw) && abs(ArrowRot.Yaw) < 95) { Direction =
                 FVector(1, 0, 0); }
489         else if (85 < abs(ArrowRot.Pitch) && abs(ArrowRot.Pitch) < 95) {
                Direction = FVector(0, 1, 0); }
490         else if (85 < abs(ArrowRot.Roll) && abs(ArrowRot.Roll) < 95) {
                Direction = FVector(0, 0, 1); }
491 
492         if (MovementSign < 0) { Direction = - Direction; }
493 
494         // Move given vertexes = update vertexes and p's positions (it is
                possible that MovementSign = 0, so in this case no movement
                should take place)
```

```
495         if (MovementSign != 0)
496         {
497                 TArray<FProceduralMeshTriangle> MeshTriangles = mesh->
                    GetProceduralMeshTriangles();
498                 VertexesArray[0].Position = FindAndMoveVertex(Direction,
                    VertexesArray[0], MeshTriangles);
499                 VertexesArray[1].Position = FindAndMoveVertex(Direction,
                    VertexesArray[1], MeshTriangles);
500                 VertexesArray[2].Position = FindAndMoveVertex(Direction,
                    VertexesArray[2], MeshTriangles);
501                 VertexesArray[3].Position = FindAndMoveVertex(Direction,
                    VertexesArray[3], MeshTriangles);
502                 mesh->SetProceduralMeshTriangles(MeshTriangles);
503         }
504         return VertexesArray;
505 }
506
507 FVector AProceduralCubeActor::CalculateFaceMiddlePoint(TArray<
    FProceduralMeshVertex> FaceVertexes)
508 {
509         // Calculate 'vertex to vertex' vectors
510         FVector V01 = FaceVertexes[1].Position - FaceVertexes[0].Position;
511         FVector V02 = FaceVertexes[2].Position - FaceVertexes[0].Position;
512         FVector V03 = FaceVertexes[3].Position - FaceVertexes[0].Position;
513
514         // Calculate 'vertex to vertex' distances
515         float D01 = sqrt(V01[0] * V01[0] + V01[1] * V01[1] + V01[2] * V01
            [2]);
516         float D02 = sqrt(V02[0] * V02[0] + V02[1] * V02[1] + V02[2] * V02
            [2]);
517         float D03 = sqrt(V03[0] * V03[0] + V03[1] * V03[1] + V03[2] * V03
            [2]);
518
519         // Find 'vertex to vertex' max distance and return the 'middle'
            point of it
520         if (D01 > D02 && D01 > D03) { return FVector((FaceVertexes[0].
            Position + FaceVertexes[1].Position) / 2); }
521         if (D02 > D01 && D02 > D03) { return FVector((FaceVertexes[0].
            Position + FaceVertexes[2].Position) / 2); }
522         return FVector((FaceVertexes[0].Position + FaceVertexes[3].Position
            ) / 2);
523 }
524
525 TArray<FProceduralMeshVertex> AProceduralCubeActor::
    FindFaceVertexesFromArrowLocation(FVector ArrowLocation)
526 {
527         TArray<FProceduralMeshVertex> FaceVertexes;
528
529         if (ArrowLocation == FrontFaceArrow->RelativeLocation)          {
            FaceVertexes.Add(v0); FaceVertexes.Add(v1); FaceVertexes.Add(v2)
            ; FaceVertexes.Add(v3); }
530         else if (ArrowLocation == BackFaceArrow->RelativeLocation)      {
            FaceVertexes.Add(v4); FaceVertexes.Add(v5); FaceVertexes.Add(v6)
            ; FaceVertexes.Add(v7); }
531         else if (ArrowLocation == LeftFaceArrow->RelativeLocation)      {
```

```
            FaceVertexes.Add(v0); FaceVertexes.Add(v1); FaceVertexes.Add(v6)
            ; FaceVertexes.Add(v7); }
532         else if (ArrowLocation == RightFaceArrow->RelativeLocation)     {
            FaceVertexes.Add(v2); FaceVertexes.Add(v3); FaceVertexes.Add(v4)
            ; FaceVertexes.Add(v5); }
533         else if (ArrowLocation == TopFaceArrow->RelativeLocation)       {
            FaceVertexes.Add(v1); FaceVertexes.Add(v2); FaceVertexes.Add(v5)
            ; FaceVertexes.Add(v6); }
534         else if (ArrowLocation == BottomFaceArrow->RelativeLocation){
            FaceVertexes.Add(v0); FaceVertexes.Add(v3); FaceVertexes.Add(v4)
            ; FaceVertexes.Add(v7); }
535
536         return FaceVertexes;
537
538         /* -- use if coplanar vertexes --
539         // Store each euclidean distance from each cube vertex to the given
                arrow
540         FProceduralMeshVertex CubeVertexes[] = { v0, v1, v2, v3, v4, v5, v6
            , v7 };
541         float VertexesDistances[8];
542         for (int i = 0; i < 8; i++) { VertexesDistances[i] =
            EuclideanDistance(CubeVertexes[i].Position, ArrowLocation); }
543
544         // Calculate the 4 nearer vertex
545         float MinDist; int iMinDist; FProceduralMeshVertex VertexMinDist;
546         TArray<FProceduralMeshVertex> FaceVertexes;
547         while (FaceVertexes.Num() < 4)
548         {
549                 MinDist = VertexesDistances[0];
550                 iMinDist = 0;
551                 for (int i = 1; i < 8; i++)
552                 {
553                         if (MinDist > VertexesDistances[i])
554                         {
555                                 MinDist = VertexesDistances[i];
556                                 iMinDist = i;
557                         }
558                 }
559                 VertexesDistances[iMinDist] = FLT_MAX;
560                 FaceVertexes.Add(CubeVertexes[iMinDist]);
561         }
562         return FaceVertexes;
563         */
564 }
565
566 FRotator AProceduralCubeActor::GetOrtogonalFaceDirectionFromFaceVertex(
       FVector GivenLocation, TArray<FProceduralMeshVertex> VertexesArray)
567 {
568         // Face Vectors (using 3 face vertex)
569         FVector V01 = FVector(VertexesArray[1].Position - VertexesArray[0].
            Position);
570         FVector V12 = FVector(VertexesArray[2].Position - VertexesArray[1].
            Position);
571
572         // Rotator to set face orthogonal direction
```

```
573             return ConvertToPitchRollYawRotator(UnitVector(CrossProd(V12, V01))
                    );
574     }
575
576     AProceduralCubeActor* AProceduralCubeActor::ExtrudeFaceOfCube(
            UStaticMeshComponent* ClickedArrow)
577     {
578             AProceduralCubeActor* NewCube = GetWorld()->SpawnActor<
                    AProceduralCubeActor>(AProceduralCubeActor::StaticClass(),
                    GetActorLocation(), GetActorRotation());
579             EnableInput(CustomPController);
580
581             int32 ArrowToExtrIndex = ExtrusionFromGivenFaceVertexes(NewCube,
                    FindFaceVertexesFromArrowLocation(ClickedArrow->RelativeLocation
                    ));
582             HideAllComponents(); // Hide all parent cube components
583
584             // Generate extruded cube and delete original cube clicked arrow to
                     extrude
585             switch (ArrowToExtrIndex)
586             {
587                     case 0: NewCube->ArrowOnClickEvent(FrontFaceArrow, false,
                            true, 0, 0);  FrontFaceArrow->SetWorldScale3D(FVector(0,
                             0, 0));      FrontFaceArrow->SetActive(false);
                            break;
588                     case 1: NewCube->ArrowOnClickEvent(BackFaceArrow, false,
                            true, 0, 0);   BackFaceArrow->SetWorldScale3D(FVector(0,
                             0, 0));       BackFaceArrow->SetActive(false);
                            break;
589                     case 2: NewCube->ArrowOnClickEvent(LeftFaceArrow, false,
                            true, 0, 0);   LeftFaceArrow->SetWorldScale3D(FVector(0,
                             0, 0));       LeftFaceArrow->SetActive(false);
                            break;
590                     case 3: NewCube->ArrowOnClickEvent(RightFaceArrow, false,
                            true, 0, 0);  RightFaceArrow->SetWorldScale3D(FVector(0,
                             0, 0));      RightFaceArrow->SetActive(false);
                            break;
591                     case 4: NewCube->ArrowOnClickEvent(TopFaceArrow, false,
                            true, 0, 0);    TopFaceArrow->SetWorldScale3D(FVector(0,
                             0, 0));        TopFaceArrow->SetActive(false);
                            break;
592                     case 5: NewCube->ArrowOnClickEvent(BottomFaceArrow, false,
                            true, 0, 0); BottomFaceArrow->SetWorldScale3D(FVector(0,
                             0, 0));     BottomFaceArrow->SetActive(false);
                            break;
593             }
594             return NewCube;
595     }
596
597     FVector AProceduralCubeActor::UnitVector(FVector GivenVector)
598     {
599             float VModule = VectorModule(GivenVector);
600             return FVector(GivenVector.X / VModule, GivenVector.Y / VModule,
                    GivenVector.Z / VModule);
601     }
```

```
602
603 float AProceduralCubeActor::EuclideanDistance(FVector P, FVector Q)
604 {
605         return sqrt((P.X - Q.X)*(P.X - Q.X) + (P.Y - Q.Y)*(P.Y - Q.Y) + (P.
                Z - Q.Z)*(P.Z - Q.Z));
606 }
607
608 float AProceduralCubeActor::VectorModule(FVector GivenVector)
609 {
610         return sqrt(GivenVector.X*GivenVector.X + GivenVector.Y*GivenVector
                .Y + GivenVector.Z*GivenVector.Z);
611 }
612
613 FVector AProceduralCubeActor::CrossProd(FVector U, FVector V)
614 {
615         return FVector(U[1] * V[2] - U[2] * V[1], U[2] * V[0] - U[0] * V
                [2], U[0] * V[1] - U[1] * V[0]);
616 }
617
618 bool AProceduralCubeActor::VectorContainsElement(TArray<int32> GivenVector,
      int32 GivenElement)
619 {
620         bool ElementFound = false;
621         for (int32 i = 0; i < GivenVector.Num() && !(ElementFound); i++)
622         {
623                 if (GivenVector[i] == GivenElement) { ElementFound = true;
                        }
624         }
625         return ElementFound;
626 }
627
628 FRotator AProceduralCubeActor::ConvertToPitchRollYawRotator(FVector
       VToConvert)
629 {
630         FRotator VConverted;
631         VConverted.Roll = 90 * VToConvert[2];
632         VConverted.Pitch = 90 * VToConvert[1];
633         VConverted.Yaw = 90 * VToConvert[0];
634         if (VConverted.Pitch != 0) // Necessary to correct from (x,y,z) to
                (r,p,y)
635         {
636                 VConverted.Pitch = - VConverted.Pitch;
637                 VConverted.Roll = VConverted.Roll + VConverted.Pitch;
638                 VConverted.Yaw = VConverted.Yaw + VConverted.Pitch;
639         }
640         return VConverted;
641 }
642
643 void AProceduralCubeActor::ArrowOnClickEvent_Implementation(
       UStaticMeshComponent* ClickedArrow, bool KeepMoving, bool KeepExtrMov,
       float LMBMovDir, float RMBMovDir)
644 {
645         // Set arrow movement
646         float Movement = 0.0;
647         if (KeepMoving) { Movement = LMBMovDir; }
```

```
648             else if (KeepExtrMov) { Movement = RMBMovDir; }
649
650             // Move Arrow
651             MoveFace(Movement, FindFaceVertexesFromArrowLocation(ClickedArrow->
                    RelativeLocation),ClickedArrow);
652     }
653
654     void AProceduralCubeActor::PlayerOnHover()
655     {
656             FHitResult HitRes;
657             CustomPController->GetHitResultUnderCursorByChannel(UEngineTypes::
                    ConvertToTraceType(ECollisionChannel::ECC_Visibility), true,
                    HitRes);
658             UStaticMeshComponent* SelectedComponent = Cast<UStaticMeshComponent
                    >(HitRes.GetComponent());
659             AProceduralCubeActor* SelectedCube = Cast<AProceduralCubeActor>(
                    HitRes.GetActor());
660
661             if (!VertexMovementState && !CubeFacesEditionState)
662             {
663                     HideVertexSpheres();
664                     HideFacesArrows();
665                     if (SelectedComponent != NULL && SelectedComponent->
                            ComponentHasTag(TEXT("VertexSphere")))
666                     {
667                             // Selected sphere at v0 selected
668                             if (SelectedComponent->GetName() == V0Sphere->
                                    GetName())
669                             {
670                                     HideFacesArrows();
671                                     KeepOnHover = true;
672                                     V0Sphere->SetHiddenInGame(false);
673                                     CustomPController->InputComponent->
                                            BindAction("LeftMB", IE_Pressed, this, &
                                            AProceduralCubeActor::KeepingOnHover);
674                                     KeepOnHover = false;
675                             }
676
677                             // Selected sphere at v1 selected
678                             else if (SelectedComponent->GetName() == V1Sphere->
                                    GetName())
679                             {
680                                     HideFacesArrows();
681                                     KeepOnHover = true;
682                                     V1Sphere->SetHiddenInGame(false);
683                                     CustomPController->InputComponent->
                                            BindAction("LeftMB", IE_Pressed, this, &
                                            AProceduralCubeActor::KeepingOnHover);
684                                     KeepOnHover = false;
685                             }
686
687                             // Selected sphere at v2 selected
688                             else if (SelectedComponent->GetName() == V2Sphere->
                                    GetName())
689                             {
```

```
690                                 HideFacesArrows();
691                                 KeepOnHover = true;
692                                 V2Sphere->SetHiddenInGame(false);
693                                 CustomPController->InputComponent->
                                        BindAction("LeftMB", IE_Pressed, this, &
                                        AProceduralCubeActor::KeepingOnHover);
694                                 KeepOnHover = false;
695                         }
696
697                         // Selected sphere at v3 selected
698                         else if (SelectedComponent->GetName() == V3Sphere->
                                GetName())
699                         {
700                                 HideFacesArrows();
701                                 KeepOnHover = true;
702                                 V3Sphere->SetHiddenInGame(false);
703                                 CustomPController->InputComponent->
                                        BindAction("LeftMB", IE_Pressed, this, &
                                        AProceduralCubeActor::KeepingOnHover);
704                                 KeepOnHover = false;
705                         }
706
707                         // Selected sphere at v4 selected
708                         else if (SelectedComponent->GetName() == V4Sphere->
                                GetName())
709                         {
710                                 HideFacesArrows();
711                                 KeepOnHover = true;
712                                 V4Sphere->SetHiddenInGame(false);
713                                 CustomPController->InputComponent->
                                        BindAction("LeftMB", IE_Pressed, this, &
                                        AProceduralCubeActor::KeepingOnHover);
714                                 KeepOnHover = false;
715                         }
716
717                         // Selected sphere at v5 selected
718                         else if (SelectedComponent->GetName() == V5Sphere->
                                GetName())
719                         {
720                                 HideFacesArrows();
721                                 KeepOnHover = true;
722                                 V5Sphere->SetHiddenInGame(false);
723                                 CustomPController->InputComponent->
                                        BindAction("LeftMB", IE_Pressed, this, &
                                        AProceduralCubeActor::KeepingOnHover);
724                                 KeepOnHover = false;
725                         }
726
727                         // Selected sphere at v6 selected
728                         else if (SelectedComponent->GetName() == V6Sphere->
                                GetName())
729                         {
730                                 HideFacesArrows();
731                                 KeepOnHover = true;
732                                 V6Sphere->SetHiddenInGame(false);
```

```
733                                     CustomPController->InputComponent->
                                            BindAction("LeftMB", IE_Pressed, this, &
                                            AProceduralCubeActor::KeepingOnHover);
734                                     KeepOnHover = false;
735                             }
736                             // Selected sphere at v7 selected
737                             else if (SelectedComponent->GetName() == V7Sphere->
                                    GetName())
738                             {
739                                     HideFacesArrows();
740                                     KeepOnHover = true;
741                                     V7Sphere->SetHiddenInGame(false);
742                                     CustomPController->InputComponent->
                                            BindAction("LeftMB", IE_Pressed, this, &
                                            AProceduralCubeActor::KeepingOnHover);
743                                     KeepOnHover = false;
744                             }
745
746                     }
747                     else if (SelectedCube != NULL && SelectedCube->GetName() ==
                             this->GetName())
748                     {
749                             CustomPController->InputComponent->BindAction("
                                    LeftMB", IE_Pressed, this, &AProceduralCubeActor
                                    ::InitCubeFacesEditionState);
750                     }
751                     FTimerHandle Handle;
752                     FTimerDelegate Delegate = FTimerDelegate::CreateUObject(
                            this, &AProceduralCubeActor::PlayerOnHover);
753                     GetWorldTimerManager().SetTimer(Handle, Delegate, 0.05f,
                            false);
754             }
755     }
756
757     void AProceduralCubeActor::PlayerOnExitHover()
758     {
759             // Check that system is not waiting to move a vertex
760             KeepOnHover = false;
761             if (!VertexMovementState) { HideAllComponents(); }
762     }
763
764     void AProceduralCubeActor::KeepingOnHover()
765     {
766             FHitResult HitRes;
767             CustomPController->GetHitResultUnderCursorByChannel(UEngineTypes::
                    ConvertToTraceType(ECollisionChannel::ECC_Visibility), true,
                    HitRes);
768             UStaticMeshComponent* SelectedSphere = Cast<UStaticMeshComponent>(
                    HitRes.GetComponent());
769
770             if (SelectedSphere == NULL) { return; }
771
772             if (SelectedSphere->GetName() == V0Sphere->GetName() && !
                    VertexMovementState)
773             {
```

```
774                     V0Sphere->SetHiddenInGame(true);
775                     V0Sphere_Arrow0->SetHiddenInGame(false); V0Sphere_Arrow1->
                            SetHiddenInGame(false); V0Sphere_Arrow2->SetHiddenInGame
                            (false);
776                     InitVertexMovementState(SelectedSphere);
777             }
778             else if (SelectedSphere->GetName() == V1Sphere->GetName() && !
                    VertexMovementState)
779             {
780                     V1Sphere->SetHiddenInGame(true);
781                     V1Sphere_Arrow0->SetHiddenInGame(false); V1Sphere_Arrow1->
                            SetHiddenInGame(false); V1Sphere_Arrow2->SetHiddenInGame
                            (false);
782                     InitVertexMovementState(SelectedSphere);
783             }
784             else if (SelectedSphere->GetName() == V2Sphere->GetName() && !
                    VertexMovementState)
785             {
786                     V2Sphere->SetHiddenInGame(true);
787                     V2Sphere_Arrow0->SetHiddenInGame(false); V2Sphere_Arrow1->
                            SetHiddenInGame(false); V2Sphere_Arrow2->SetHiddenInGame
                            (false);
788                     InitVertexMovementState(SelectedSphere);
789             }
790             else if (SelectedSphere->GetName() == V3Sphere->GetName() && !
                    VertexMovementState)
791             {
792                     V3Sphere->SetHiddenInGame(true);
793                     V3Sphere_Arrow0->SetHiddenInGame(false); V3Sphere_Arrow1->
                            SetHiddenInGame(false); V3Sphere_Arrow2->SetHiddenInGame
                            (false);
794                     InitVertexMovementState(SelectedSphere);
795             }
796             else if (SelectedSphere->GetName() == V4Sphere->GetName() && !
                    VertexMovementState)
797             {
798                     V4Sphere->SetHiddenInGame(true);
799                     V4Sphere_Arrow0->SetHiddenInGame(false); V4Sphere_Arrow1->
                            SetHiddenInGame(false); V4Sphere_Arrow2->SetHiddenInGame
                            (false);
800                     InitVertexMovementState(SelectedSphere);
801             }
802             else if (SelectedSphere->GetName() == V5Sphere->GetName() && !
                    VertexMovementState)
803             {
804                     V5Sphere->SetHiddenInGame(true);
805                     V5Sphere_Arrow0->SetHiddenInGame(false); V5Sphere_Arrow1->
                            SetHiddenInGame(false); V5Sphere_Arrow2->SetHiddenInGame
                            (false);
806                     InitVertexMovementState(SelectedSphere);
807             }
808             else if (SelectedSphere->GetName() == V6Sphere->GetName() && !
                    VertexMovementState)
809             {
810                     V6Sphere->SetHiddenInGame(true);
```

```
811                 V6Sphere_Arrow0->SetHiddenInGame(false); V6Sphere_Arrow1->
                        SetHiddenInGame(false); V6Sphere_Arrow2->SetHiddenInGame
                        (false);
812                 InitVertexMovementState(SelectedSphere);
813         }
814         else if (SelectedSphere->GetName() == V7Sphere->GetName() && !
                VertexMovementState)
815         {
816                 V7Sphere->SetHiddenInGame(true);
817                 V7Sphere_Arrow0->SetHiddenInGame(false); V7Sphere_Arrow1->
                        SetHiddenInGame(false); V7Sphere_Arrow2->SetHiddenInGame
                        (false);
818                 InitVertexMovementState(SelectedSphere);
819         }
820
821 }
822
823 void AProceduralCubeActor::HideVertexSpheres()
824 {
825         V0Sphere->SetHiddenInGame(true);        V1Sphere->SetHiddenInGame(
                true);        V2Sphere->SetHiddenInGame(true);        V3Sphere->
                SetHiddenInGame(true);
826         V4Sphere->SetHiddenInGame(true);        V5Sphere->SetHiddenInGame(
                true);        V6Sphere->SetHiddenInGame(true);        V7Sphere->
                SetHiddenInGame(true);
827 }
828
829 void AProceduralCubeActor::HideFacesArrows()
830 {
831         FrontFaceArrow->SetHiddenInGame(true);  BackFaceArrow->
                SetHiddenInGame(true);   LeftFaceArrow->SetHiddenInGame(true);
832         RightFaceArrow->SetHiddenInGame(true);  TopFaceArrow->
                SetHiddenInGame(true);    BottomFaceArrow->SetHiddenInGame(true)
                ;
833 }
834
835 void AProceduralCubeActor::ShowFacesArrows()
836 {
837         FrontFaceArrow->SetHiddenInGame(false); BackFaceArrow->
                SetHiddenInGame(false);  LeftFaceArrow->SetHiddenInGame(false);
838         RightFaceArrow->SetHiddenInGame(false); TopFaceArrow->
                SetHiddenInGame(false);   BottomFaceArrow->SetHiddenInGame(false
                );
839 }
840
841 void AProceduralCubeActor::HideSpheresArrows()
842 {
843         V0Sphere_Arrow0->SetHiddenInGame(true); V0Sphere_Arrow1->
                SetHiddenInGame(true); V0Sphere_Arrow2->SetHiddenInGame(true);
844         V1Sphere_Arrow0->SetHiddenInGame(true); V1Sphere_Arrow1->
                SetHiddenInGame(true); V1Sphere_Arrow2->SetHiddenInGame(true);
845         V2Sphere_Arrow0->SetHiddenInGame(true); V2Sphere_Arrow1->
                SetHiddenInGame(true); V2Sphere_Arrow2->SetHiddenInGame(true);
846         V3Sphere_Arrow0->SetHiddenInGame(true); V3Sphere_Arrow1->
                SetHiddenInGame(true); V3Sphere_Arrow2->SetHiddenInGame(true);
```

```
847         V4Sphere_Arrow0->SetHiddenInGame(true); V4Sphere_Arrow1->
                SetHiddenInGame(true); V4Sphere_Arrow2->SetHiddenInGame(true);
848         V5Sphere_Arrow0->SetHiddenInGame(true); V5Sphere_Arrow1->
                SetHiddenInGame(true); V5Sphere_Arrow2->SetHiddenInGame(true);
849         V6Sphere_Arrow0->SetHiddenInGame(true); V6Sphere_Arrow1->
                SetHiddenInGame(true); V6Sphere_Arrow2->SetHiddenInGame(true);
850         V7Sphere_Arrow0->SetHiddenInGame(true); V7Sphere_Arrow1->
                SetHiddenInGame(true); V7Sphere_Arrow2->SetHiddenInGame(true);
851 }
852
853 void AProceduralCubeActor::HideAllComponents()
854 {
855         HideVertexSpheres();
856         HideFacesArrows();
857         HideSpheresArrows();
858 }
859
860 void AProceduralCubeActor::InitVertexMovementState(UStaticMeshComponent*
        SelectedSphere)
861 {
862         VertexMovementState = true;
863         QuitVertexMovementState();
864 }
865
866 void AProceduralCubeActor::QuitVertexMovementState()
867 {
868         CustomPController->InputComponent->BindAction("ExitQ", IE_Pressed,
                this, &AProceduralCubeActor::UnsetVertexMovementState);
869         if (!VertexMovementState)
870         {
871                 FTimerHandle Handle;
872                 FTimerDelegate Delegate = FTimerDelegate::CreateUObject(
                        this, &AProceduralCubeActor::QuitVertexMovementState);
873                 GetWorldTimerManager().SetTimer(Handle, Delegate, 0.05f,
                        false);
874         }
875 }
876
877 void AProceduralCubeActor::UnsetVertexMovementState()
878 {
879         VertexMovementState = false;
880         PlayerOnExitHover();
881 }
882
883 void AProceduralCubeActor::InitCubeFacesEditionState()
884 {
885         if (VertexMovementState) { return; }
886
887         HideAllComponents();
888         ShowFacesArrows();
889         CubeFacesEditionState = true;
890         VertexMovementState = true; // Set to true in order to forbid
                vertex movement while cube editing
891         QuitCubeFacesEditionState();
892 }
```

```
893
894 void AProceduralCubeActor::QuitCubeFacesEditionState()
895 {
896         CustomPController->InputComponent->BindAction("ExitQ", IE_Pressed,
               this, &AProceduralCubeActor::UnsetCubeFacesEditionState);
897         if (!CubeFacesEditionState)
898         {
899                 FTimerHandle Handle;
900                 FTimerDelegate Delegate = FTimerDelegate::CreateUObject(
                       this, &AProceduralCubeActor::QuitCubeFacesEditionState);
901                 GetWorldTimerManager().SetTimer(Handle, Delegate, 0.05f,
                       false);
902         }
903 }
904
905 void AProceduralCubeActor::UnsetCubeFacesEditionState()
906 {
907         CubeFacesEditionState = false;
908         VertexMovementState = false;
909         PlayerOnExitHover();
910 }
911
912 void AProceduralCubeActor::Tick(float deltaSeconds)
913 {
914         Super::Tick(deltaSeconds);
915 }
```

Código A.3: CustomPlayerController.h

```
1  #pragma once
2
3  #include "GameFramework/PlayerController.h"
4  #include "ProceduralMeshComponent.h"
5  #include "CustomPlayerController.generated.h"
6
7  UCLASS()
8  class PROCEDURALMESH_API ACustomPlayerController : public APlayerController
9  {
10         GENERATED_BODY()
11         ACustomPlayerController(const FObjectInitializer& ObjectInitializer
               );
12
13 public:
14
15         // --------- VARIABLES --------- \\
16
17         UPROPERTY(BlueprintReadWrite, Category = "CustomPlayerController")
18         bool KeepMovingFace;
19
20         UPROPERTY(BlueprintReadWrite, Category = "CustomPlayerController")
21         bool KeepLeftMovingVertex;
22
23         UPROPERTY(BlueprintReadWrite, Category = "CustomPlayerController")
24         bool KeepRightMovingVertex;
```

```
25
26          UPROPERTY(BlueprintReadWrite, Category = "CustomPlayerController")
27          bool KeepExtrMovement;
28
29          UPROPERTY(BlueprintReadWrite, Category = "CustomPlayerController")
30          float LMBMovementDirection;
31
32          UPROPERTY(BlueprintReadWrite, Category = "CustomPlayerController")
33          float RMBMovementDirection;
34
35          UPROPERTY(BlueprintReadWrite, Category = "CustomPlayerController")
36          UStaticMeshComponent* LMB_SelectedArrow;
37
38          UPROPERTY(BlueprintReadWrite, Category = "CustomPlayerController")
39          UStaticMeshComponent* RMB_SelectedArrow;
40
41          UPROPERTY(EditAnywhere, Category = "CustomPlayerController")
42          class AProceduralCubeActor* LMB_CubeToEdit;
43
44          UPROPERTY(EditAnywhere, Category = "CustomPlayerController")
45          class AProceduralCubeActor* RMB_CubeToEdit;
46
47          // --------- LEFT MOUSE BUTTON FUNCTIONS  --------- \\
48
49          UFUNCTION(BlueprintCallable, Category = "CustomPlayerController")
50          void EnableKeepMovings(); // Sets to true KeepMovingFace or
               KeepMovingVertex
51
52          UFUNCTION(BlueprintCallable, Category = "CustomPlayerController")
53          void DisableKeepMovings(); // Sets to false KeepMovingFace or
               KeepMovingVertex
54
55          UFUNCTION(BlueprintCallable, Category = "CustomPlayerController")
56          void UpdateLMBMovementDirection(float value);
57
58          // --------- RIGHT MOUSE BUTTON FUNCTIONS --------- \\
59
60          UFUNCTION(BlueprintCallable, Category = "CustomPlayerController")
61          void SetKeepExtrMovementToTrue();
62
63          UFUNCTION(BlueprintCallable, Category = "CustomPlayerController")
64          void SetKeepExtrMovementToFalse();
65
66          UFUNCTION(BlueprintCallable, Category = "CustomPlayerController")
67          void UpdateRMBMovementDirection(float value);
68
69          UFUNCTION(BlueprintCallable, Category = "CustomPlayerController")
70          void Refresh();
71
72          // --------- (override) --------- \\
73
74          void SetupInputComponent();
75
76          //  --------- AUX FUNCTIONS --------- \\
77
```

```
78          UFUNCTION(BlueprintCallable, Category = "CustomPlayerController")
79          FVector RecognizeArrowDirectionInWorld(UStaticMeshComponent*
                SelectedArrow, AProceduralCubeActor* SelectedCube);
80
81          UFUNCTION(BlueprintCallable, Category = "CustomPlayerController")
82          UStaticMeshComponent* RecognizeSphereFromArrow(UStaticMeshComponent
                * SelectedArrow, AProceduralCubeActor* SelectedCube);
83
84          UFUNCTION(BlueprintCallable, Category = "CustomPlayerController")
85          FProceduralMeshVertex RecognizeVertexFromArrow(UStaticMeshComponent
                * SelectedArrow, AProceduralCubeActor* SelectedCube);
86
87  };
```

Código A.4: CustomPlayerController.cpp

```
1   #include "ProceduralMesh.h"
2   #include "ProceduralCubeActor.h"
3   #include "CustomPlayerController.h"
4
5   ACustomPlayerController::ACustomPlayerController(const FObjectInitializer&
        ObjectInitializer)
6           : Super(ObjectInitializer) {
7
8           bEnableClickEvents = true;
9           bEnableMouseOverEvents = true;
10          bShowMouseCursor = true;
11
12          EnableInput(this);
13
14          KeepMovingFace = false;
15          KeepExtrMovement = false;
16          KeepLeftMovingVertex = false;
17          KeepRightMovingVertex = false;
18
19          LMBMovementDirection = 0.0;
20          RMBMovementDirection = 0.0;
21
22          LMB_CubeToEdit = NULL;
23          LMB_SelectedArrow = NULL;
24
25          RMB_CubeToEdit = NULL;
26          RMB_SelectedArrow = NULL;
27
28  }
29
30  // --------- LEFT MOUSE BUTTON FUNCTIONS --------- \\
31
32  void ACustomPlayerController::EnableKeepMovings()
33  {
34          FHitResult HitRes;
35          GetHitResultUnderCursorByChannel(UEngineTypes::ConvertToTraceType(
                ECollisionChannel::ECC_Visibility), true, HitRes);
36
```

```
37          LMB_CubeToEdit = Cast<AProceduralCubeActor>(HitRes.GetActor());
38          LMB_SelectedArrow = Cast<UStaticMeshComponent>(HitRes.GetComponent
                ());
39
40          if (LMB_CubeToEdit != NULL && LMB_SelectedArrow != NULL)
41          {
42                  if (LMB_SelectedArrow->ComponentHasTag(TEXT("FaceArrow")))
43                  {
44                          KeepMovingFace = true;
45                          Refresh();
46                  }
47                  else if (LMB_SelectedArrow->ComponentHasTag(TEXT("
                        VertexArrow")))
48                  {
49                          KeepLeftMovingVertex = true;
50                          Refresh();
51                  }
52          }
53  }
54
55  void ACustomPlayerController::DisableKeepMovings()
56  {
57          KeepMovingFace = false;
58          KeepLeftMovingVertex = false;
59          LMB_SelectedArrow = NULL;
60          LMB_CubeToEdit = NULL;
61  }
62
63  void ACustomPlayerController::UpdateLMBMovementDirection(float value)
64  {
65          LMBMovementDirection = value;
66  }
67
68  // --------- RIGHT MOUSE BUTTON FUNCTIONS --------- \\
69
70  void ACustomPlayerController::SetKeepExtrMovementToTrue()
71  {
72          FHitResult HitRes;
73          GetHitResultUnderCursorByChannel(UEngineTypes::ConvertToTraceType(
                ECollisionChannel::ECC_Visibility), true, HitRes);
74
75          RMB_CubeToEdit = Cast<AProceduralCubeActor>(HitRes.GetActor());
76          RMB_SelectedArrow = Cast<UStaticMeshComponent>(HitRes.GetComponent
                ());
77
78          if (RMB_CubeToEdit != NULL && RMB_SelectedArrow != NULL )
79          {
80                  if (RMB_SelectedArrow->ComponentHasTag(TEXT("FaceArrow")))
81                  {
82                          KeepExtrMovement = true;
83                          AProceduralCubeActor* ExtrudedCube = RMB_CubeToEdit
                                ->ExtrudeFaceOfCube(RMB_SelectedArrow);
84
85                          TArray<FProceduralMeshVertex> FaceVertexes =
                                RMB_CubeToEdit->
```

```
                                FindFaceVertexesFromArrowLocation(
                                RMB_SelectedArrow->RelativeLocation);
86                              int32 IndexOf_RMB_SelectedArrow = RMB_CubeToEdit->
                                IdentifyFaceFromVertexes(FaceVertexes[0],
                                FaceVertexes[1], FaceVertexes[2], FaceVertexes
                                [3]);
87                              switch (IndexOf_RMB_SelectedArrow)
88                              {
89                                      case 0: RMB_SelectedArrow = ExtrudedCube->
                                        FrontFaceArrow;       break;
90                                      case 1: RMB_SelectedArrow = ExtrudedCube->
                                        BackFaceArrow;        break;
91                                      case 2: RMB_SelectedArrow = ExtrudedCube->
                                        LeftFaceArrow;        break;
92                                      case 3: RMB_SelectedArrow = ExtrudedCube->
                                        RightFaceArrow;       break;
93                                      case 4: RMB_SelectedArrow = ExtrudedCube->
                                        TopFaceArrow;         break;
94                                      case 5: RMB_SelectedArrow = ExtrudedCube->
                                        BottomFaceArrow;      break;
95                              }
96
97                              RMB_CubeToEdit->VertexMovementState = false;
98                              RMB_CubeToEdit->CubeFacesEditionState = false;
99                              RMB_CubeToEdit = ExtrudedCube;
100                             Refresh();
101                     }
102
103                     else if (RMB_SelectedArrow->ComponentHasTag(TEXT("
                        VertexArrow")))
104                     {
105                             KeepRightMovingVertex = true;
106                             Refresh();
107                     }
108             }
109     }
110
111     void ACustomPlayerController::SetKeepExtrMovementToFalse()
112     {
113             KeepExtrMovement = false;
114             RMB_SelectedArrow = NULL;
115             KeepRightMovingVertex = false;
116             RMB_CubeToEdit = NULL;
117     }
118
119     void ACustomPlayerController::UpdateRMBMovementDirection(float value)
120     {
121             RMBMovementDirection = value;
122     }
123
124     // --------- (override) --------- \\
125
126     void ACustomPlayerController::SetupInputComponent(/*class UInputComponent *
            InputComponent*/)
127     {
```

```
128         Super::SetupInputComponent();
129         check(InputComponent);
130 
131         InputComponent->BindAxis("XAxis", this, &ACustomPlayerController::
               UpdateLMBMovementDirection);
132         InputComponent->BindAxis("RMBAxis", this, &ACustomPlayerController
               ::UpdateRMBMovementDirection);
133         InputComponent->BindAction("LeftMB", IE_Pressed, this, &
               ACustomPlayerController::EnableKeepMovings);
134         InputComponent->BindAction("LeftMB", IE_Released, this, &
               ACustomPlayerController::DisableKeepMovings);
135         InputComponent->BindAction("RightMB", IE_Pressed, this, &
               ACustomPlayerController::SetKeepExtrMovementToTrue);
136         InputComponent->BindAction("RightMB", IE_Released, this, &
               ACustomPlayerController::SetKeepExtrMovementToFalse);
137 }
138 
139 void ACustomPlayerController::Refresh()
140 {
141         if (!(KeepMovingFace) && !(KeepExtrMovement) && !(
               KeepLeftMovingVertex) && !(KeepRightMovingVertex)) { return; }
142 
143         if (KeepMovingFace)                                 {
               LMB_CubeToEdit->MoveFace(LMBMovementDirection, LMB_CubeToEdit->
               FindFaceVertexesFromArrowLocation(LMB_SelectedArrow->
               RelativeLocation), LMB_SelectedArrow);      }
144         else if (KeepExtrMovement)              {       RMB_CubeToEdit->
               MoveFace(RMBMovementDirection, RMB_CubeToEdit->
               FindFaceVertexesFromArrowLocation(RMB_SelectedArrow->
               RelativeLocation), RMB_SelectedArrow); RMB_CubeToEdit->
               HideAllComponents(); }
145         else if (KeepLeftMovingVertex)  {       LMB_CubeToEdit->
               MoveVertexAlongWorldAxis(RecognizeArrowDirectionInWorld(
               LMB_SelectedArrow, LMB_CubeToEdit), RecognizeSphereFromArrow(
               LMB_SelectedArrow, LMB_CubeToEdit), RecognizeVertexFromArrow(
               LMB_SelectedArrow, LMB_CubeToEdit), LMBMovementDirection); }
146         else if (KeepRightMovingVertex) {       RMB_CubeToEdit->
               MoveVertexAlongRotatedAxis(RecognizeArrowDirectionInWorld(
               RMB_SelectedArrow, RMB_CubeToEdit), RecognizeSphereFromArrow(
               RMB_SelectedArrow, RMB_CubeToEdit), RecognizeVertexFromArrow(
               RMB_SelectedArrow, RMB_CubeToEdit), RMBMovementDirection); }
147 
148         FTimerHandle Handle;
149         FTimerDelegate Delegate = FTimerDelegate::CreateUObject(this, &
               ACustomPlayerController::Refresh);
150         GetWorldTimerManager().SetTimer(Handle, Delegate, 0.05f, false);
151 }
152 
153 FVector ACustomPlayerController::RecognizeArrowDirectionInWorld(
    UStaticMeshComponent* SelectedArrow, AProceduralCubeActor* SelectedCube)
154 {
155         if (
156                 (SelectedArrow->GetName() == SelectedCube->V0Sphere_Arrow0
                    ->GetName()) || (SelectedArrow->GetName() ==
                    SelectedCube->V1Sphere_Arrow0->GetName()) ||
```

```
157                     (SelectedArrow->GetName() == SelectedCube->V2Sphere_Arrow0
                           ->GetName()) || (SelectedArrow->GetName() ==
                           SelectedCube->V3Sphere_Arrow0->GetName()) ||
158                     (SelectedArrow->GetName() == SelectedCube->V4Sphere_Arrow0
                           ->GetName()) || (SelectedArrow->GetName() ==
                           SelectedCube->V5Sphere_Arrow0->GetName()) ||
159                     (SelectedArrow->GetName() == SelectedCube->V6Sphere_Arrow0
                           ->GetName()) || (SelectedArrow->GetName() ==
                           SelectedCube->V7Sphere_Arrow0->GetName())
160                     )
161             {
162                     return FVector(0, 0, 1);
163             }
164             else if (
165                     (SelectedArrow->GetName() == SelectedCube->V0Sphere_Arrow1
                           ->GetName()) || (SelectedArrow->GetName() ==
                           SelectedCube->V1Sphere_Arrow1->GetName()) ||
166                     (SelectedArrow->GetName() == SelectedCube->V2Sphere_Arrow1
                           ->GetName()) || (SelectedArrow->GetName() ==
                           SelectedCube->V3Sphere_Arrow1->GetName()) ||
167                     (SelectedArrow->GetName() == SelectedCube->V4Sphere_Arrow1
                           ->GetName()) || (SelectedArrow->GetName() ==
                           SelectedCube->V5Sphere_Arrow1->GetName()) ||
168                     (SelectedArrow->GetName() == SelectedCube->V6Sphere_Arrow1
                           ->GetName()) || (SelectedArrow->GetName() ==
                           SelectedCube->V7Sphere_Arrow1->GetName())
169                     )
170             {
171                     return FVector(0, 1, 0);
172             }
173             else
174             {
175                     return FVector(1, 0, 0);
176             }
177     }
178
179     UStaticMeshComponent* ACustomPlayerController::RecognizeSphereFromArrow(
            UStaticMeshComponent* SelectedArrow, AProceduralCubeActor* SelectedCube)
180     {
181
182             if (SelectedArrow->RelativeLocation.Equals(SelectedCube->p0)) {
                    return SelectedCube->V0Sphere; }
183             if (SelectedArrow->RelativeLocation.Equals(SelectedCube->p1)) {
                    return SelectedCube->V1Sphere; }
184             if (SelectedArrow->RelativeLocation.Equals(SelectedCube->p2)) {
                    return SelectedCube->V2Sphere; }
185             if (SelectedArrow->RelativeLocation.Equals(SelectedCube->p3)) {
                    return SelectedCube->V3Sphere; }
186             if (SelectedArrow->RelativeLocation.Equals(SelectedCube->p4)) {
                    return SelectedCube->V4Sphere; }
187             if (SelectedArrow->RelativeLocation.Equals(SelectedCube->p5)) {
                    return SelectedCube->V5Sphere; }
188             if (SelectedArrow->RelativeLocation.Equals(SelectedCube->p6)) {
                    return SelectedCube->V6Sphere; }
189             return SelectedCube->V7Sphere;
```

```
190 }
191
192 FProceduralMeshVertex ACustomPlayerController::RecognizeVertexFromArrow(
       UStaticMeshComponent* SelectedArrow, AProceduralCubeActor* SelectedCube)
193 {
194         if (SelectedArrow->RelativeLocation.Equals(SelectedCube->p0)) {
               return SelectedCube->v0; }
195         if (SelectedArrow->RelativeLocation.Equals(SelectedCube->p1)) {
               return SelectedCube->v1; }
196         if (SelectedArrow->RelativeLocation.Equals(SelectedCube->p2)) {
               return SelectedCube->v2; }
197         if (SelectedArrow->RelativeLocation.Equals(SelectedCube->p3)) {
               return SelectedCube->v3; }
198         if (SelectedArrow->RelativeLocation.Equals(SelectedCube->p4)) {
               return SelectedCube->v4; }
199         if (SelectedArrow->RelativeLocation.Equals(SelectedCube->p5)) {
               return SelectedCube->v5; }
200         if (SelectedArrow->RelativeLocation.Equals(SelectedCube->p6)) {
               return SelectedCube->v6; }
201         return SelectedCube->v7;
202 }
```

Anexo B

Un ejemplo de uso de cuaterniones en Unreal Engine: `FTransform`

Con el fin de evidenciar el uso de los cuaterniones en motores gráficos como Unreal Engine, presento en el Código B.1 que sigue un fragmento inicial del código fuente de `FTransform`, cuyos elementos están formados por un valor de escalado, una rotación definida con un cuaternión y una traslación definida mediante un vector usual. `FTransform` es una función disponible para el programador que aplica una transformación sobre un objeto componiendo, en ese orden, un escalado, una rotación y una traslación. Tal como se detalla en los comentarios del código, la rotación que se utiliza en `FTransform` ha de estar definida por un cuaternión.

Código B.1: FTransform

```
1  // Copyright 1998-2015 Epic Games, Inc. All Rights Reserved.
2
3  #pragma once
4
5  /** This returns Quaternion Inverse of X **/
6  #define MAKE_QUATINV_VECTORREGISTER(X) VectorMultiply(GlobalVectorConstants
       ::QINV_SIGN_MASK, X)
7
8  /**
9   * Transform composed of Scale, Rotation (as a quaternion), and Translation
        .
10  *
11  * Transforms can be used to convert from one space to another, for example
        by transforming
12  * positions and directions from local space to world space.
13  *
14  * Transformation of position vectors is applied in the order:  Scale ->
        Rotate -> Translate.
15  * Transformation of direction vectors is applied in the order: Scale ->
        Rotate.
16  *
17  * Order matters when composing transforms: C = A * B will yield a
        transform C that logically
18  * first applies A then B to any subsequent transformation. Note that this
```

```
     is the opposite order of quaternion (FQuat) multiplication.
 *
 * Example: LocalToWorld = (DeltaRotation * LocalToWorld) will change
     rotation in local space by DeltaRotation.
 * Example: LocalToWorld = (LocalToWorld * DeltaRotation) will change
     rotation in world space by DeltaRotation.
 */

MS_ALIGN(16) struct FTransform
{
#if !defined(COREUOBJECT_API)
        #define MAYBE_COREUOBJECT_API
#else
        #define MAYBE_COREUOBJECT_API COREUOBJECT_API
#endif
        friend MAYBE_COREUOBJECT_API class UScriptStruct*
           Z_Construct_UScriptStruct_UObject_FTransform();

protected:
        /** Rotation of this transformation, as a quaternion */
        VectorRegister  Rotation;
        /** Translation of this transformation, as a vector */
        VectorRegister  Translation;
        /** 3D scale (always applied in local space) as a vector */
        VectorRegister Scale3D;
public:
        /**
         * The identity transformation (Rotation = FQuat::Identity,
            Translation = FVector::ZeroVector, Scale3D = (1,1,1))
         */
        static CORE_API const FTransform Identity;

        (...)
```