

GRADO EN INGENIERÍA EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

HERRAMIENTAS PARA EL DISEÑO INDUSTRIAL EN REALIDAD VIRTUAL

Autor: Pedro Bodega Palomeque

Director: José Antonio Rodríguez Mondejar

Madrid

Julio de 2024

CUADRO DE FIRMAS

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título HERRAMIENTAS PARA EL DISEÑO INDUSTRIAL EN REALIDAD VIRTUAL en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2023/2024 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada

de otros documentos está debidamente referenciada.

Fdo.: Pedro Bodega Palomeque Fecha: 16 / 07 / 2024



Autorizada la entrega del proyecto

JOSÉ ANTONIO RODRÍGUEZ MONDÉJAR

Fdo.: José Antonio Rodríguez Mondéjar

Firmado por RODRIGUEZ MONDEJAR JOSE ANTONIO - ***5731** el día 16/07/2024 con un certificado emitido por AC FNMT Usuarios



GRADO EN INGENIERÍA EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

HERRAMIENTAS PARA EL DISEÑO INDUSTRIAL EN REALIDAD VIRTUAL

Autor: Pedro Bodega Palomeque

Director: José Antonio Rodríguez Mondejar

Madrid

Julio de 2024

UNIVERSIDAD PONTIFICIA COMILLAS
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)
GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

HERRAMIENTAS PARA EL DISEÑO INDUSTRIAL EN REALIDAD VIRTUAL

Autor: Bodega Palomeque, Pedro
Director: Rodríguez Mondéjar, José Antonio.

Entidad Colaboradora: ICAI – Universidad Pontificia Comillas.

RESUMEN

Este proyecto se centra en el desarrollo de un programa de realidad virtual que mejore la experiencia de diseño en el sector industrial. Para su implementación, se hace uso del programa Unity [1] y de las gafas de realidad virtual Meta Quest 2 [2]. El lenguaje de programación usado es C#. El objetivo es proporcionar una experiencia de diseño más intuitiva y realista, al editar los objetos directamente en 3D, y no a través de una pantalla como en los programas CAD 3D tradicionales.

Palabras clave: Realidad Virtual, dibujo técnico, Unity, diseño industrial, Meta Quest, CAD 3D

Introducción

El sector del dibujo técnico ha evolucionado a lo largo de los años. Al principio, los diseños se hacían de manera manual con lápices y reglas, en planos bidimensionales. Con la revolución tecnológica de los ordenadores, se empezaron a implementar programas en 2D como AutoCAD [3], donde todo el proceso se hacía en ordenador. Para facilitar la interpretación de las piezas, también se desarrollaron programas que permitían el diseño directamente en 3D, interactuando a través de una pantalla. Entre ellos destacan SolidWorks [4] y Siemens NX [5].

El propósito de este proyecto es dar un paso más a esta tendencia natural que ha sufrido el sector del diseño industrial en los últimos años. Se piensa que la nueva manera de diseñar será dentro de un entorno de realidad virtual (RV) inmersivo, donde el usuario podrá diseñar de una manera más natural y realista. Este enfoque soluciona algunos problemas de interpretación que se pueden dar al visualizar objetos 3D a través de pantallas 2D. Además, se busca mejorar la experiencia de diseño, que se cree que es más intuitiva al hacer uso de controladores en lugar de teclado y ratón.

Este enfoque innovador va de la mano con la creciente tendencia que está experimentando el sector de la realidad virtual. El mercado mundial de software de RV estaba valorado en 7.800 y 9.980 millones de dólares en 2021 y 2022 respectivamente. Se espera que este crecimiento continúe más rápidamente, esperando una valoración de 25.320 millones de dólares en 2026 [6].

Resumen

Definición del proyecto

En este proyecto se busca implementar un entorno inmersivo donde los diseñadores puedan diseñar en RV. Para ello, se ha buscado inspiración en las herramientas de programas CAD 3D como SolidWorks o NX.

Esta propuesta es innovadora, ya que no existe ningún programa profesional que aborde el mismo problema. Las propuestas existentes se centran en proporcionar una experiencia de diseño más creativa y poco precisa. Entre estos programas se encuentran GravitySketch [7] y Logitech VR Ink Pilot Edition [8]. Estas propuestas son muy útiles para una fase previa al diseño definitivo de los componentes, cuando no se requiere de precisión. Sin embargo, no presentan la exactitud necesaria para el diseño final de un producto en el sector del diseño industrial. Por ello, en este proyecto se busca implementar herramientas intuitivas, pero a la vez precisas.

Debido a la alta complejidad de los programas CAD profesionales, no se busca recrear todas sus funcionalidades dentro de la realidad virtual. Al ser un programa pionero en el ámbito, se va a tratar de implementar las funcionalidades más básicas, que permitan diseñar una gran variedad de piezas distintas.

Entre las funcionalidades implementadas destacan:

- **Creación de formas básicas**: se le permite al usuario crear formas básicas como cubos, cilindros y esferas, como base para estructuras más complejas.

- **Trasladar, rotar y redimensionar intuitivo**: Método para mover las piezas de manera aproximada y rápida. También permite redimensionar la pieza en los distintos ejes. Muy útil para colocar piezas por el escenario, pero poco preciso. El usuario interacciona con unos ejes que se crean en las piezas.

- **Trasladar, rotar y redimensionar exacto**: Esta herramienta es más lenta, pero mucho más precisa. Se le permite al usuario introducir manualmente los valores de posición y dimensión a través de un teclado numérico.

- **Vaciado**: Herramienta muy útil para hacer agujeros o para eliminar material de una pieza.

- **Eliminar**: Función básica para eliminar una pieza o grupo.

- **Duplicar**: duplica la pieza seleccionada.

- **Agrupar**: permite agrupar varias piezas para que se comporten como una sola.

- **Movimiento por puntos**: restringe la orientación de una pieza en función de otra. Este movimiento puede ser seleccionando 1, 2 o 3 puntos de cada pieza. Si se selecciona 1 punto en cada pieza, dichos puntos se vuelven coincidentes.

Al seleccionar los segundos puntos, hace que las líneas que unen los puntos de cada pieza sean paralelas. Al seleccionar los tres puntos en cada pieza, los planos definidos por los 3 puntos seleccionados se vuelven paralelos.

- **Activar / desactivar cotas**: Activa y desactiva las cotas en las piezas que se seleccionen. Las cotas se muestran en las dimensiones x, y, z locales de dicha pieza.

- **Guardar pieza**: exporta la pieza o grupo seleccionado en un fichero predeterminado en formato .fbx.

- **Importar en SolidWorks**: Se explica de manera detallada el proceso para importar las piezas en programas CAD 3D profesionales como SolidWorks. Para ello, hay que exportar la pieza por el programa intermedio Fusion 360, y finalmente importarla en SolidWorks.

En la figura se puede observar el entorno inmersivo diseñado en el programa Unity.

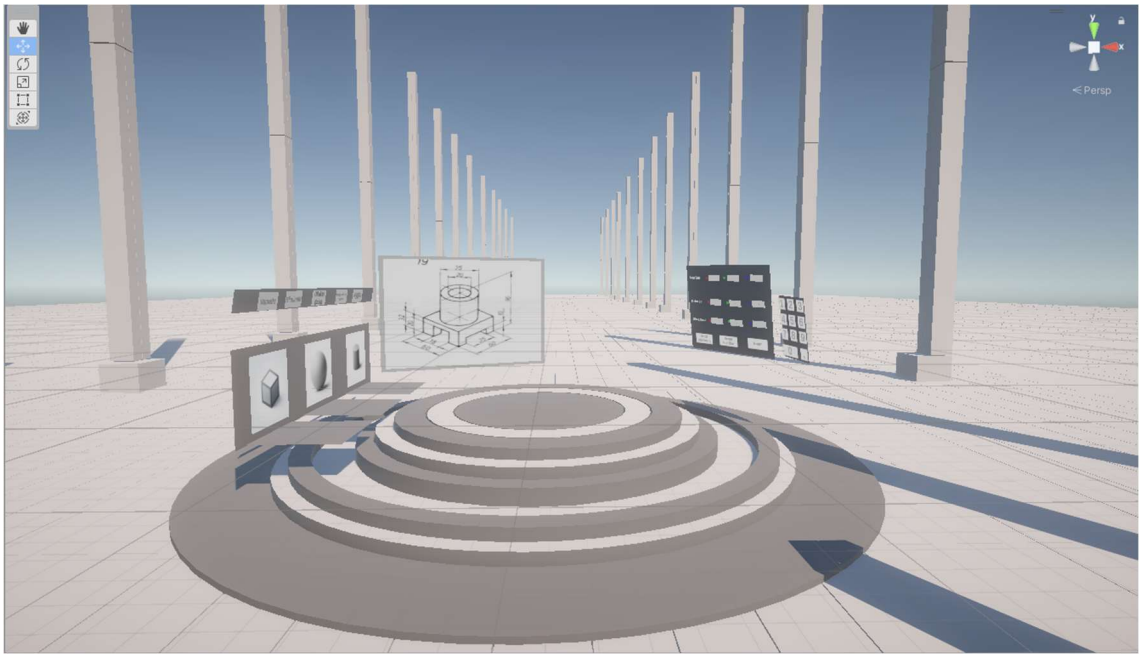


Ilustración 1: Visión general del escenario de diseño

Descripción del proyecto

El programa se desarrolla en el entorno de Unity, usando C# como lenguaje de programación. Para implementar las distintas funcionalidades, se han escrito 22 diferentes scripts que se comunican entre ellos. Desde el principio se ha buscado independencia y generalización al

escribir código, es por ello que se ha hecho uso del script "EventHub". Es el corazón del programa, ya que es el intermediario entre el resto de scripts, cuando quieren llamar a métodos entre sí.

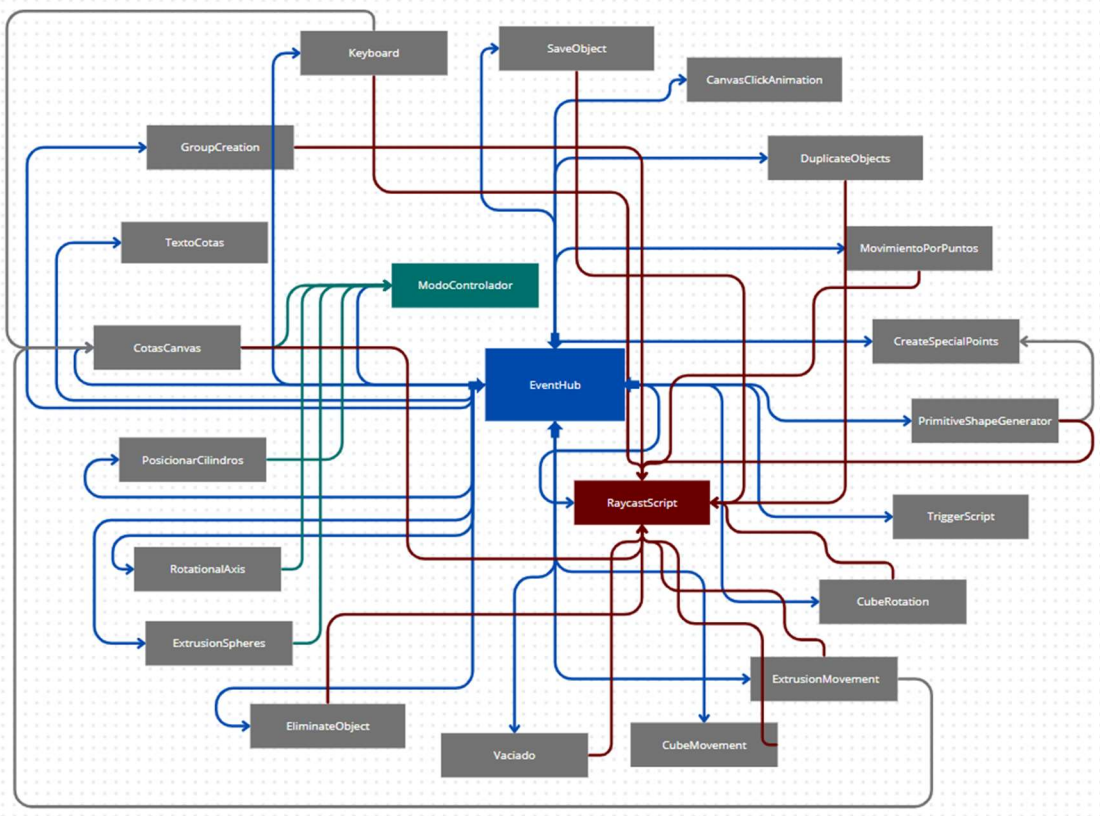


Ilustración 2: Diagrama de intercomunicación entre todos los scripts

Se ha decidido que la interacción entre el usuario y el entorno sea a través de un raycast (o láser). Se piensa que este método es intuitivo y natural, y permite interactuar con objetos en la distancia. Las distintas herramientas y funcionalidades implementadas se pueden activar y desactivar a través del raycast. Éstas estarán repartidas por el entorno inmersivo.

Debido a que no se habilitan todas las herramientas para diseñar componentes complejos, se explica el proceso para exportar dichas piezas a programas como SolidWorks. Este proceso no es trivial, ya que hay que lidiar con un problema en el formato de los archivos. Los objetos 3D de Unity se definen por sus mallas, y los programas CAD profesionales definen sus modelos 3D mediante parámetros. Los objetos definidos por mallas no se adaptan bien las superficies curvas, ya que las aproximan mediante triángulos. Esta pérdida de información puede ser recuperada gracias al programa Fusion 360, que interpreta la superficie de la pieza y la convierte en formato paramétrico.

El programa tiene en cuenta las diferentes posibilidades que tiene el usuario al interactuar con los distintos elementos del escenario y está preparado para que no se activen funcionalidades indeseadas. De esta manera, se ha procurado eliminar la mayoría de errores debido a un uso

indebido del programa. Esto ha sido gracias a un estricto orden de comunicación entre scripts, y a un seguimiento riguroso de los distintos diagramas de flujo, mostrados en los apartados correspondientes.

Resultados

Las herramientas implementadas han demostrado ser intuitivas, y de un uso más natural que los programas CAD tradicionales. También han demostrado la posibilidad de crear un gran número distinto de componentes básicos. Se recuerda que el objetivo no era igualar la complejidad de programas profesionales como SolidWorks. Es por ello que se da la posibilidad de que los usuarios exporten sus piezas a dicho programa a través del programa intermedio Fusion 360.

Se ha encontrado un fallo indeseado en las mallas de las piezas cuando se realiza la operación de vaciado repetidamente sobre el mismo objeto. Este error es debido al complejo cálculo de la malla resultante, problema que sigue abierto en la comunidad de desarrolladores de programas 3D. Este fallo puede ser corregido en la mayoría de los casos en el programa Fusion 360, el cual presenta distintas herramientas para reparar las mallas de los objetos.

En cuanto a la precisión, se ha descubierto una pequeña variación en las medidas de la pieza una vez exportada en SolidWorks. Esto es debido al paso del formato por mallas al formato paramétrico. Para solucionar este problema hay que aumentar la complejidad y detalle de las mallas, en función de las tolerancias requeridas para cada proyecto.

Son estos problemas con las mallas los que hacen replantearse las bases de este proyecto. Y se plantea que futuras investigaciones opten por trabajar directamente en el formato paramétrico directamente, ahorrando así posibles problemas que surgen al trabajar con mallas. También se insta a aumentar las funcionalidades, profesionalizando así el programa

Conclusiones

Se han implementado herramientas para que el usuario pueda manipular objetos 3D de una manera precisa, que le permitan diseñar una gran variedad de piezas distintas. Entre ellas se encuentra la herramienta de vaciado, movimiento por puntos, y la integración con software CAD 3D como SolidWorks para detallar la pieza con funcionalidades más complejas. Aunque hay algunos fallos relacionados con las mallas, se consiguen los objetivos propuestos.

Con todo esto, se ha creado un programa innovador, que abre las puertas a una nueva manera de diseñar en el sector del diseño industrial. Este proyecto tiene el potencial de revolucionar el diseño técnico y se espera que futuras investigaciones expandan esta herramienta para incluir funcionalidades adicionales y así profesionalizarla.

Resumen

Con este proyecto se es pionero en la implementación de la realidad virtual en el diseño industrial, abriendo nuevas posibilidades a los diseñadores y mejorando así la eficiencia en el desarrollo de productos.

Bibliografía

[1] Unity, «Unity, main page,» 2024. [En línea]. Available: https://unity.com/es. [Último acceso: 15 07 2024].

[2] Meta, «Meta Quest 2,» 2024. [En línea]. Available: https://www.meta.com/es/quest/products/quest-2/. [Último acceso: 15 07 2024].

[3] Autodesk, «Autodesk AutoCAD,» 2024. [En línea]. Available: https://www.autodesk.com/es/products/autocad/overview?term=1-YEAR&tab=subscription. [Último acceso: 15 07 2024].

[4] SolidWorks, «SolidWorks,» 2024. [En línea]. Available: https://www.solidworks.com/es. [Último acceso: 15 07 2024].

[5] Siemens, «Software NX,» 2024. [En línea]. Available: https://plm.sw.siemens.com/es-ES/nx/. [Último acceso: 15 07 2024].

[6] The Business Research Company, Virtual Realitiy Software Global Market Report 2024, 2024.

[7] GravitySketch, «Gravity Sketch,» 2024. [En línea]. Available: https://www.gravitysketch.com/. [Último acceso: 15 07 2024].

[8] Logitech, «Unleash Creativity in True 3D, with Logitech VR Ink Pilot Edition,» Logitech, 21 04 2022. [En línea]. Available: https://youtu.be/MduBtmijXZc?si=O3aeAjT5VSYxuQ92. [Último acceso: 2024 07 15].

Resumen

Abstract

This project focuses on the development of a virtual reality program that enhances the design experience in the industrial sector. For its implementation, Unity [1] and the Meta Quest 2 [2] virtual reality headset are used. The programming language employed is C#. The goal is to provide a more intuitive and realistic design experience by editing objects directly in 3D, unlike traditional 3D CAD programs that work through a screen.

Keywords: Virtual Reality, technical drawing, Unity, industrial design, Meta Quest, 3D CAD

Introduction

The technical drawing sector has evolved over the years. Initially, designs were created manually with pencils and rulers on two-dimensional planes. With the technological revolution of computers, 2D programs like AutoCAD [3] were implemented, where the entire process was done on a computer. To facilitate the interpretation of parts, programs that allowed direct 3D design through a screen were also developed. Among them, SolidWorks [4] and Siemens NX [5] stand out.

The purpose of this project is to take a further step in this natural trend that the industrial design sector has undergone in recent years. It is believed that the new way of designing will be within an immersive virtual reality (VR) environment, where the user can design more naturally and realistically. This approach solves some interpretation problems that can arise when viewing 3D objects through 2D screens. Additionally, it aims to improve the design experience, which is thought to be more intuitive when using controllers instead of keyboard and mouse.

This innovative approach aligns with the growing trend in the virtual reality sector. The global VR software market was valued at $7.8 billion and $9.98 billion in 2021 and 2022, respectively. This growth is expected to continue more rapidly, with a projected valuation of $25.32 billion by 2026 [2].

Definition of the project

This project aims to implement an immersive environment where designers can work in VR. To achieve this, inspiration has been drawn from 3D CAD tools like SolidWorks and NX.

This proposal is innovative because no existing professional program addresses the same problem. Current solutions focus on providing a more creative and less precise design experience. Examples of such programs include Gravity Sketch [7] and Logitech VR Ink Pilot Edition [9]. These tools are very useful for preliminary design phases where precision is not required. However, they lack the necessary accuracy for the final design of products in the industrial design sector. Therefore, this project aims to implement tools that are both intuitive and precise.

Resumen

Due to the high complexity of professional CAD programs, the goal is not to recreate all their functionalities within virtual reality. Being a pioneering program in this field, the focus will be on implementing the most basic functionalities that allow for the design of a wide variety of different parts.

Implemented Functionalities**:**

- **Creation of Basic Shapes**: Users can create basic shapes like cubes, cylinders, and spheres as the foundation for more complex structures.

- **Intuitive Translate, Rotate, and Resize**: A method for quickly and approximately moving pieces. It also allows resizing along different axes. This is very useful for positioning pieces in the scene but is less precise. Users interact with axes created on the pieces.

- **Precise Translate, Rotate, and Resize**: This tool is slower but much more accurate. It allows users to manually enter position and dimension values using a numeric keypad.

- **Hollowing**: A very useful tool for making holes or removing material from a piece.

- **Delete**: Basic function to delete a piece or group.

- **Duplicate**: Duplicates the selected piece.

- **Group**: Allows grouping several pieces so they behave as one.

- **Point-to-Point Movement**: Restricts the orientation of a piece based on another. This movement can be done by selecting 1, 2, or 3 points on each piece. If 1 point is selected on each piece, those points become coincident. Selecting the second points makes the lines connecting the points on each piece parallel. Selecting three points on each piece makes the planes defined by the three selected points parallel.

- **Activate/Deactivate Dimensions**: Activates and deactivates dimensions on the selected pieces. Dimensions are shown in the local x, y, z dimensions of the piece.

- **Save Piece**: Exports the selected piece or group to a predetermined file in .fbx format.

- **Import into SolidWorks**: A detailed explanation of the process for importing pieces into professional 3D CAD programs like SolidWorks. This involves exporting the piece through an intermediate program, Fusion 360, and finally importing it into SolidWorks.

In the figure, the immersive environment designed in the Unity program can be observed.

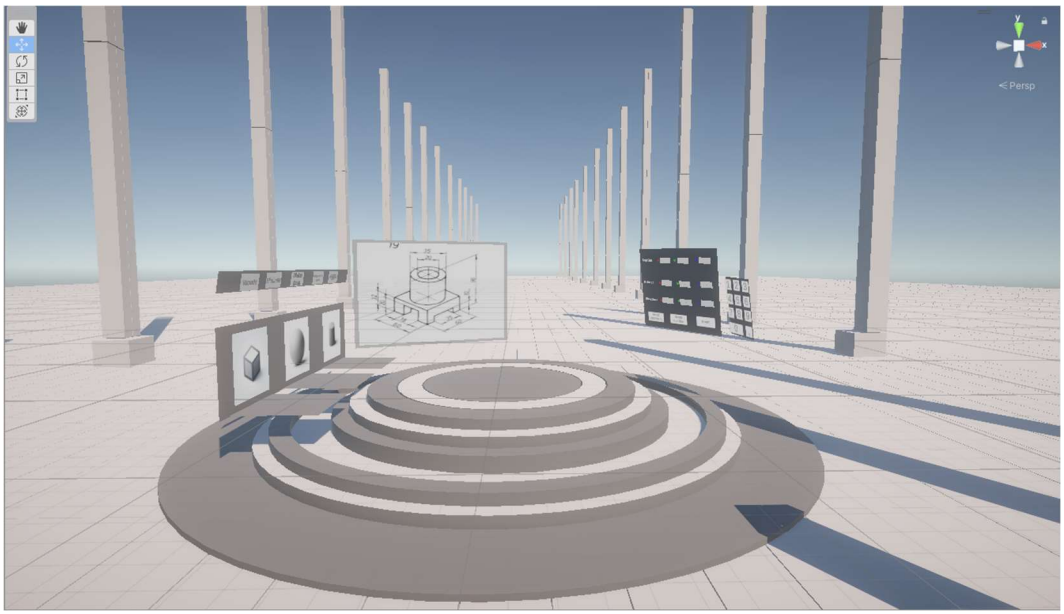


Ilustración 3: General visión of the design space

Description of the project

The program is developed in the Unity environment, using C# as the programming language. To implement the various functionalities, 22 different scripts have been written that communicate with each other. From the beginning, the goal was to achieve independence and generalization when writing code, which is why the “EventHub” script has been used. It is the heart of the program, acting as the intermediary between the other scripts when they need to call methods from each other.

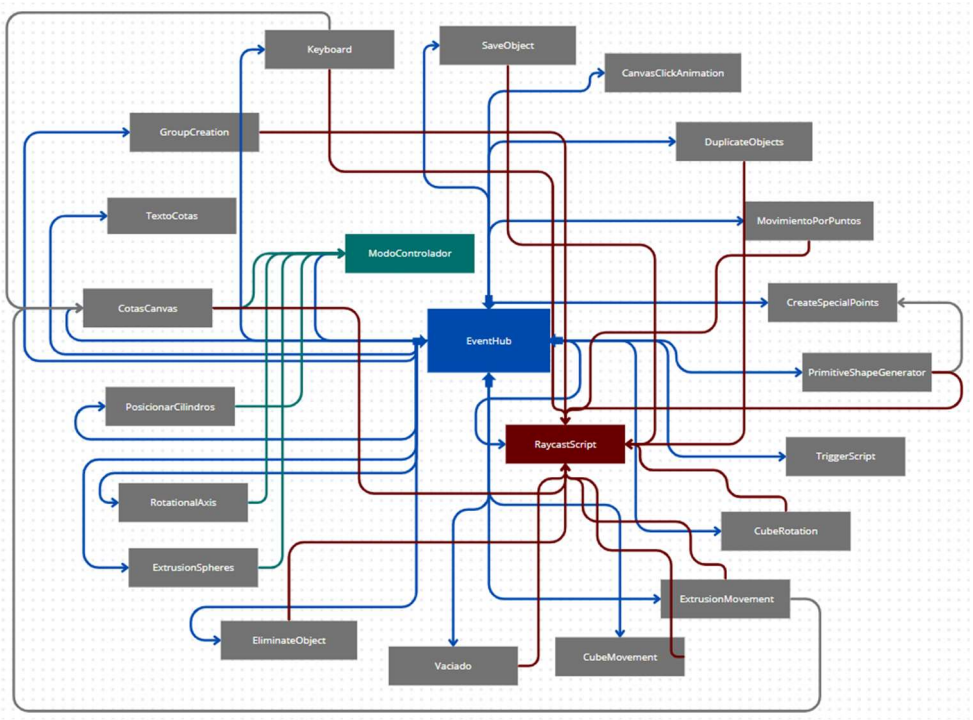


Ilustración 4: Diagram of the communication between the scripts

The user interaction with the environment has been designed to be through a raycast (or laser). This method is considered intuitive and natural, allowing interaction with objects at a distance. The various implemented tools and functionalities can be activated and deactivated through the raycast. These will be distributed throughout the immersive environment.

Since not all tools for designing complex components are enabled, the process for exporting these pieces to programs like SolidWorks is explained. This process is not trivial, as there is an issue with file format compatibility. 3D objects in Unity are defined by their meshes, while professional CAD programs define their 3D models using parameters. Mesh-defined objects do not adapt well to curved surfaces as they approximate them using triangles. This loss of information can be recovered thanks to Fusion 360, which interprets the surface of the piece and converts it into a parametric format.

The program takes into account the various possibilities that the user has when interacting with the different elements of the scene and is designed to prevent unwanted functionalities from being activated. This way, most errors due to improper use of the program have been eliminated. This has been achieved through a strict order of communication between scripts and a rigorous follow-up of the various flowcharts, shown in the corresponding sections.

Results

The implemented tools have proven to be intuitive and offer a more natural usage experience compared to traditional CAD programs. They have also demonstrated the ability to create a large number of different basic components. It is important to remember that the objective was not

to match the complexity of professional programs like SolidWorks. Therefore, users are given the option to export their pieces to such programs through the intermediary program Fusion 360.

An undesirable issue has been found in the meshes of the pieces when the hollowing operation is repeatedly performed on the same object. This error is due to the complex calculation of the resulting mesh, a problem that remains unresolved in the 3D software developer community. This issue can be corrected in most cases in Fusion 360, which offers various tools for repairing object meshes.

Regarding precision, a slight variation in the measurements of the piece has been discovered once exported to SolidWorks. This is due to the transition from mesh format to parametric format. To solve this problem, it is necessary to increase the complexity and detail of the meshes according to the tolerances required for each project.

These mesh-related issues prompt a reconsideration of the project's foundations. It is suggested that future research should work directly with the parametric format, thereby avoiding potential problems that arise from working with meshes. Additionally, it is encouraged to increase functionalities, thus professionalizing the program.

Conclusions

Tools have been implemented to allow users to manipulate 3D objects precisely, enabling them to design a wide variety of different parts. Among these tools are hollowing, point-to-point movement, and integration with 3D CAD software like SolidWorks to detail the piece with more complex functionalities. Although there are some issues related to the meshes, the proposed objectives are achieved.

With all this, an innovative program has been created, opening the doors to a new way of designing in the industrial design sector. This project has the potential to revolutionize technical design, and it is hoped that future research will expand this tool to include additional functionalities and thus professionalize it.

This project pioneers the implementation of virtual reality in industrial design, opening new possibilities for designers and thereby improving efficiency in product development.

Bibliography

[1] Unity, «Unity, main page,» 2024. [En línea]. Available: https://unity.com/es. [Último acceso: 15 07 2024].

[2] Meta, «Meta Quest 2,» 2024. [En línea]. Available: https://www.meta.com/es/quest/products/quest-2/. [Último acceso: 15 07 2024].

[3] Autodesk, «Autodesk AutoCAD,» 2024. [En línea]. Available: https://www.autodesk.com/es/products/autocad/overview?term=1-YEAR&tab=subscription. [Último acceso: 15 07 2024].

[4] SolidWorks, «SolidWorks,» 2024. [En línea]. Available: https://www.solidworks.com/es. [Último acceso: 15 07 2024].

[5] Siemens, «Software NX,» 2024. [En línea]. Available: https://plm.sw.siemens.com/es-ES/nx/. [Último acceso: 15 07 2024].

[6] The Business Research Company, Virtual Realitiy Software Global Market Report 2024, 2024.

[7] GravitySketch, «Gravity Sketch,» 2024. [En línea]. Available: https://www.gravitysketch.com/. [Último acceso: 15 07 2024].

[8] Logitech, «Unleash Creativity in True 3D, with Logitech VR Ink Pilot Edition,» Logitech, 21 04 2022. [En línea]. Available: https://youtu.be/MduBtmijXZc?si=O3aeAjT5VSYxuQ92. [Último acceso: 2024 07 15].

TABLA DE CONTENIDO

Cuadro de Firmas 2
Resumen 4
Tabla de Ilustraciones 19
1 Introducción 21
1.1 Planteamiento del problema. 21
1.2 Objetivo del proyecto. 22
1.3 Justificación del proyecto. 22
1.4 Herramientas para el diseño implementadas 23
1.5 Principales problemas 25
2 Estado del Arte 26
2.1 Diseño paramétrico en Unity, Archimatix: 30
3 Metodología 31
3.1 Unity y C# 31
3.2 Visual Studio 32
3.3 Fusion 360 32
3.4 Notion 33
3.5 Meta Quest 2 33
3.6 Diseño del sistema y estructura del software 33
4 Implementación del Modelo 37
4.1 Configurar las Meta Quest para modo desarrollador 37
4.2 Preparar el entorno de Unity 37
4.3 XR Interaction Toolkit 37
4.4 Habilitar botones de los controladores 38
4.5 Script para leer los botones y EventHub 39
4.6 Creación y configuración del raycast 41
4.7 Movimiento del usuario 45
4.8 Creación de las formas primitivas 46
4.8.1 Canvas 47
4.8.2 Formas Primitivas 49
4.9 Traslación de las piezas 51
4.9.1 Creación de ejes de traslación 52

4.9.2 Movimiento de Traslación ... 53
4.10 Rotación de las piezas ... 56
4.10.1 Creación de ejes de rotación ... 57
4.10.2 Movimiento de rotación ... 58
4.11 Redimensionado de las piezas ... 60
4.11.1 Creación de ejes de redimensionado ... 61
4.11.2 Redimensionado ... 63
4.11.3 Problema del escalado indeseado de los hijos ... 65
4.12 Modos de edición ... 66
4.13 Activar / Desactivar objetos ... 70
4.14 Eliminar los objetos creados ... 73
4.15 Vaciado ... 75
4.15.1 Problema de origen de referencia al hacer vaciado ... 78
4.15.2 Problema de bugs en la malla ... 79
4.16 Movimiento por puntos ... 81
4.16.1 Creación de los puntos característicos ... 82
4.16.2 La lógica del movimiento por puntos ... 83
4.17 Creación de grupos ... 88
4.18 Añadir Cotas ... 90
4.19 Trasladar, Rotar y Redimensionar de manera exacta ... 93
4.20 Duplicar Objetos ... 97
4.21 Guardado ... 99
4.22 Animaciones ... 100
4.23 Escenario ... 101
4.24 Exportar piezas a SolidWorks ... 104
4.24.1 Implementación del programa intermediario Fusion 360 ... 105
4.24.2 Importar en SolidWorks ... 108
5 Análisis de Resultados ... 109
5.1 Maximizar la creación de componentes ... 109
5.2 Ayuda de visualización 3D, herramientas intuitivas ... 109
5.3 Precisión y exactitud ... 110
5.4 Otros fallos ... 112
5.5 Pruebas y feedback ... 112
5.6 Reflexiones finales y futuras implementaciones ... 112

Resumen

6 Presupuesto 114
7 Alineamiento con los Objetivos de Desarrollo Sostenible (ODS) 115
8 Conclusiones y Recomendaciones 116
9 Bibliografía 117
10 Anexos 120
10.1 EventHub 120
10.2 TriggerScript 122
10.3 RaycastScript 124
10.4 PrimitiveShapeGenerator 129
10.5 PosicionarCilindros 132
10.6 CubeMovement 134
10.7 RotationalAxis 138
10.8 CubeRotation 140
10.9 ExtrusionSpheres 143
10.10 ExtrusionMovement 147
10.11 ModoControlador 155
10.12 EliminateObject 161
10.13 Vaciado 162
10.14 CreateSpecialPoints 168
10.15 MovimientoPorPuntos 173
10.16 GroupCreation 178
10.17 TextoCotas 185
10.18 CotasCanvas 188
10.19 Keyboard 191
10.20 DuplicateObjects 198
10.21 SaveObject 200
10.22 CanvasClickAnimation 202

TABLA DE ILUSTRACIONES

Ilustración 1: Visión general del escenario de diseño 6
Ilustración 2: Diagrama de intercomunicación entre todos los scripts 7
Ilustración 3: General visión of the design space 12
Ilustración 4: Diagram of the communication between the scripts 13
Ilustración 5: Diferentes fases de la evolución del dibujo técnico. A) Diseño sobre papel 2D. B) Diseño en AutoCad 2D. C) Diseño en SolidEdge 3D. 21
Ilustración 6: Prueba de diseño por Pedro Bodega en Gravity Sketch 28
Ilustración 7: Modelo 3D colaborativo en Gravity Sketch 28
Ilustración 8: Interfaz de Logitech VR Ink Pilot Edition 29
Ilustración 9: Diagrama de intercomunicación entre todos los scripts 34
Ilustración 10: Canvas con las distintas herramientas disponibles 35
Ilustración 11: Ejemplo de instalación de XR Interaction Toolkit 38
Ilustración 12: Configuración de los "Input Actions" en Unity 39
Ilustración 13: Diagrama a alto nivel de la lógica al pulsar un botón 40
Ilustración 14: Imagen que ilustra el raycast impactando en un objeto en Unity 42
Ilustración 15: Diagrama a alto nivel del funcionamiento del raycast 42
Ilustración 16: Raycast impactando en un eje a través de la pieza transparente 44
Ilustración 17: Estructura de la función creación de las formas primitivas 47
Ilustración 18: Imagen de los 3 canvas para crear las formas primitivas 47
Ilustración 19: Ejemplo de traslación de la pieza en el eje Y local 51
Ilustración 20: Diagrama a alto nivel de la función traslación de piezas 52
Ilustración 21: Ilustración gráfica del funcionamiento de la lógica de traslación 54
Ilustración 22: Estructura a alto nivel de la función rotación de las piezas 57
Ilustración 23: Usuario rotando la pieza usando los ejes de rotación 58
Ilustración 24: Ilustración gráfica del funcionamiento de la lógica de rotación 59
Ilustración 25: Estructura a alto nivel de la función redimensionado de las piezas 61
Ilustración 26: Usuario modificando la altura de un cilindro mediante los ejes de redimensionado 62
Ilustración 27: Jerarquía de los hijos de las piezas 66
Ilustración 28: Estructura a alto nivel de los modos de edición 67
Ilustración 29: Los 3 modos de edición. A) Traslación. B) Rotación. C) Redimensionado 67
Ilustración 30: Ejemplo de pieza siendo activada al ser impactada por el raycast 70
Ilustración 31: Estructura a alto nivel de la función activar objetos 71
Ilustración 32: Estructura a alto nivel de la función de eliminar objetos 74
Ilustración 33: Pasos para eliminar una pieza. A) Seleccionarla. B) Disparar al canvas "Eliminate" 74
Ilustración 34: Ejemplo de funcionamiento de la operación booleana resta (Vaciado) [28]. 75
Ilustración 35: Estructura a alto nivel de la función de vaciado 76
Ilustración 36: Pasos para utilizar la herramienta vaciado. A) Disparar al canvas. B) Disparar al "objeto1". C) Disparar al "objeto2" 76
Ilustración 37: Resultado de la pieza de (Ilustración 36) tras haber aplicado el vaciado 77
Ilustración 38: Ejemplo de error en la malla de una pieza tras haber aplicado varias veces la operación de vaciado 80

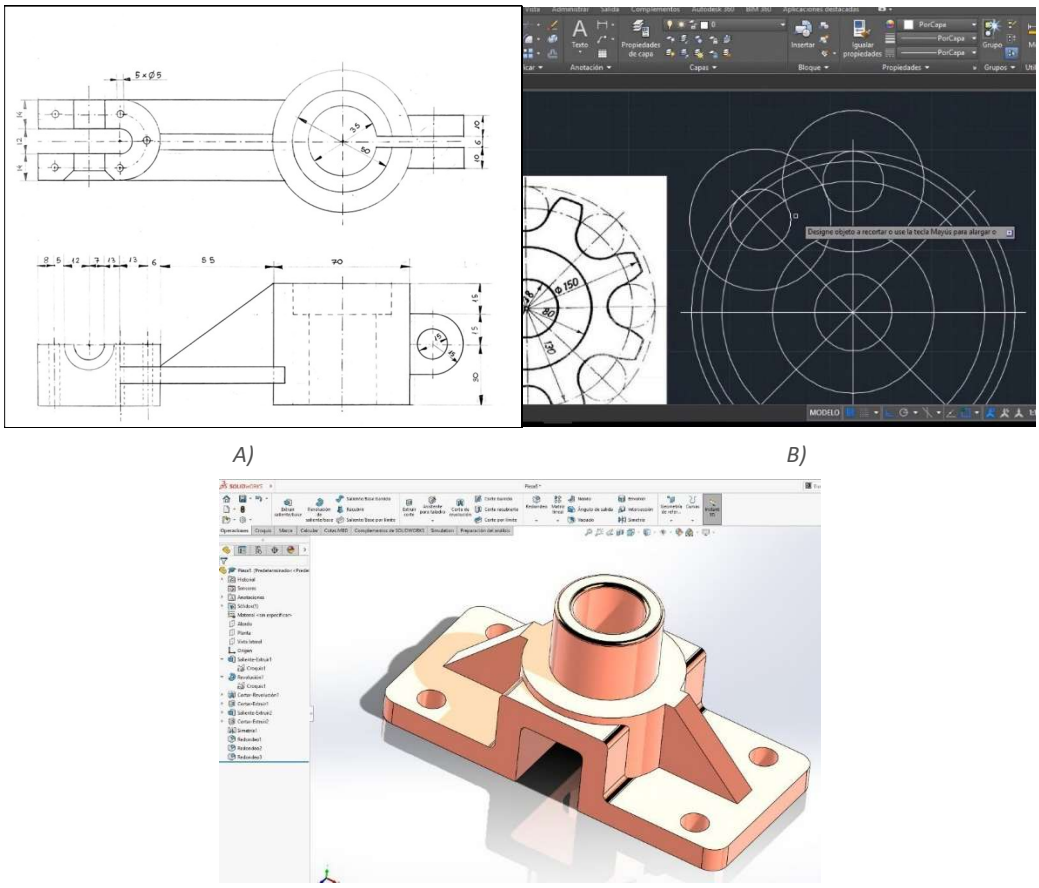
Tabla de Ilustraciones

Ilustración 39: Malla de la pieza de la Ilustración 38 vista en Fusion 360 81
Ilustración 40: Funcionamiento de la herramienta de movimiento por puntos en RobotStudio 81
Ilustración 41: Estructura a alto nivel de la función movimiento por puntos 82
Ilustración 42: Puntos característicos creados en: A) Cubo. B) Cilindro. C) Esfera 83
Ilustración 43: Activación del movimiento por puntos 84
Ilustración 44: Pasos ejecutando el movimiento por puntos. A) 1 punto en cada pieza. B) 2 puntos en cada Pieza. C) 3 puntos en cada pieza 86
Ilustración 45: Resultado de las piezas de la Ilustración 44 después de finalizar el movimiento por puntos 87
Ilustración 46: Estructura a alto nivel de la función de creación de grupos 88
Ilustración 47: Pasos para crear un grupo. A) Seleccionar en el canvas "Make group". B) Seleccionar las piezas y disparar en el canvas "Click to finish". 88
Ilustración 48: Ejemplo de rotación de un grupo. Se comporta como una pieza individual 90
Ilustración 49: Estructura a alto nivel de la función de añadir cotas 90
Ilustración 50: Usuario activando las cotas, disparando al raycast 91
Ilustración 51: Cotas activas en un cubo 92
Ilustración 52: Cotas siendo modificadas al momento al redimensionar la pieza 93
Ilustración 53: Estructura a alto nivel del modo exacto 94
Ilustración 54: Pasos para usar la "función exacta". A) Disparar al canvas "Exact positioning". B) Seleccionar la pieza. C) Disparar a la casilla con el valor que se quiere cambiar. D) Introducir el nuevo valor en el teclado numérico. E) Disparar a "Enter" 95
Ilustración 55: Configuración de eventos a alto nivel en el Inspector de Unity 96
Ilustración 56: Estructura a alto nivel de la función de duplicar objetos 98
Ilustración 57: Ejemplo de uso de la herramienta de duplicar objetos. A) Disparar al canvas "Duplicate". B) Resultado de los objetos duplicados 99
Ilustración 58: Estructura a alto nivel de la función de guardado 100
Ilustración 59: Estructura a alto nivel de la ejecución de animaciones 100
Ilustración 60: Inspiración para diseñar el escenario 102
Ilustración 61: Elementos diseñados en el escenario. A) Columna. B) Base. C) Suelo 102
Ilustración 62: Canvas con las distintas herramientas disponibles 102
Ilustración 63: Canvas y bases para generar las formas primitivas 103
Ilustración 64: Teclado numérico y display con información de las piezas 103
Ilustración 65: Imagen ejemplo de boceto de pieza para usar como referencia 104
Ilustración 66: Visión general del escenario de diseño 104
Ilustración 67: Pieza exportada a SolidWorks sin convertir primero a formato paramétrico. A) Pieza en Unity. B) Pieza en SolidWorks 105
Ilustración 68: Pieza .fbx diseñada en Unity 106
Ilustración 69: Pieza importada a Fusion 360 106
Ilustración 70: Pieza reparada y con los grupos de cara generados en Fusion 360 107
Ilustración 71: Pieza convertida a formato paramétrico en Fusion 360 108
Ilustración 72: Pieza Importada en SolidWorks 108
Ilustración 73: Boceto de la figura diseñada en la Ilustración 72 110
Ilustración 74: Medición de la figura en Fusion 360, vista desde arriba 111

1 INTRODUCCIÓN

1.1 Planteamiento del problema.

El dibujo técnico ha ido evolucionando a lo largo de la historia, cambiando los métodos usados para diseñar. Al principio, lo diseños se hacían exclusivamente en papel, con lápiz y reglas. Posteriormente, con el desarrollo de los ordenadores, los diseñadores se empezaron a apoyar de herramientas usando programas "Computer-Aided Design" CAD. Entre ellos cabe destacar AutoCAD [3]. Esto supuso un gran avance en el sector, ya que permitía diseñar con mucha mayor precisión y en un tiempo mucho menor



A) *B)*

C)

Ilustración 5: Diferentes fases de la evolución del dibujo técnico. A) Diseño sobre papel 2D. B) Diseño en AutoCad 2D. C) Diseño en SolidEdge 3D.

Últimamente se desarrollaron numerosos programas para dejar de diseñar en un plano (2D) para diseñar directamente en 3D. Esta mejora permitió que el proceso fuese mucho más

intuitivo. Entre estos programas se encuentran SolidWorks [4], Siemens NX [5]. Ello facilitó la tarea de los diseñadores, pudiendo ahora diseñar de una manera mucho más intuitiva y eficiente. Aunque se diseña una pieza directamente en 3D, la interfaz con el usuario es una pantalla de un ordenador, lo cual es realmente 2D.

Nota: denomino "pieza" al modelo que el usuario está diseñando.

Siguiendo la línea de mejora del sector, en este proyecto se busca ir un paso más allá en el desarrollo del dibujo técnico. Ello se plantea permitiendo al usuario diseñar directamente en un entorno 3D por medio de unas gafas de realidad virtual (RV). En este caso las Meta Quest 2 [10].

1.2 OBJETIVO DEL PROYECTO.

En este proyecto se busca el diseño y la creación de un programa de realidad virtual que permita a los usuarios diseñar piezas de ingeniería directamente en un entorno 3D. De esta manera, los usuarios podrán diseñar de una manera mucho más intuitiva, y con muchos menos problemas de interpretación, ya que no están viendo a través de una pantalla 2D.

El objetivo inicial de este proyecto era conectar en tiempo real programas como NX y Unity [1], para poder usar en realidad virtual todas las herramientas que ofrece NX. Sin embargo, por problemas de licencias, no se ha podido seguir este camino. Por ello, el nuevo objetivo definido es la creación de un programa de RV que disponga de las herramientas y operaciones básicas para poder diseñar una gran variedad de piezas de diseño industrial. Además de buscar opciones para exportar estas piezas en programas como NX o SolidWorks para poder añadir detalles de funciones más complejas.

Debido a la complejidad de este tipo de aplicaciones, no se busca recrear todas las funciones que estas ofrecen, sino implementar las operaciones más generales que permitan diseñar el mayor número de tipos de piezas posibles. Todo ello siempre buscando las maneras más intuitivas para facilitar el proceso de diseño.

Con todo ello, se busca dar el primer paso hacia la siguiente manera de diseñar, siguiendo la tendencia que ha tenido el sector del diseño industrial hasta el momento.

1.3 JUSTIFICACIÓN DEL PROYECTO.

De esta manera, se reducirán considerablemente los problemas de interpretación de los modelos. También se facilitará el trabajo de los diseñadores, para que todo el proceso sea mucho más intuitivo. Nuestro cerebro necesita menos esfuerzo de visualización si estamos observando las piezas directamente en 3D, en lugar de verlas a través de una pantalla.

Este proyecto busca que en el futuro se pueda mejorar y profesionalizar, para que sea la herramienta preferida por los diseñadores. Para que ello se cumpla, se tiene que dar un requisito, que las gafas de realidad virtual sean asequibles y de uso común.

Los datos y predicciones sobre el uso de dispositivos RV y de entornos inmersivos son prometedores. Un estudio [11] descubrió que el 66% de las compañías está adoptando tecnologías de realidad virtual con fines de formación. Esto demuestra el interés general por esta nueva tecnología, y que las empresas ven potencial a la realidad virtual. Esto también se demuestra en las inversiones económicas en este sector. El mercado mundial de software de RV estaba valorado en 7.800 y 9.980 millones de dólares en 2021 y 2022 respectivamente. Se espera que este crecimiento continúe más rápidamente, esperando una valoración de 25.320 millones de dólares en 2026 [6].

Con estas evidencias se demuestra la prometedora proyección del sector de las tecnologías inmersivas. Por ello, con este proyecto se pretende contribuir a este progreso, anticipándose a las futuras demandas de los profesionales del diseño industrial.

1.4 HERRAMIENTAS PARA EL DISEÑO IMPLEMENTADAS

En esta sección se va a hacer un resumen de las herramientas disponibles a los usuarios para diseñar las piezas:

- **Seleccionar / disparar** (4.4, 4.5, 4.13): Para que el usuario interactúe con el entorno, deberá pulsar el gatillo del controlador derecho apuntando al objeto que quiera "disparar".

- **Creación de piezas básicas** (4.8): El usuario puede "disparar" con el controlador a las imágenes de 3 formas primitivas: cubo, esfera y cilindro. Acto seguido, se genera dicha pieza.

- **Traslación, rotación y redimensionado intuitivo** (4.9, 4.10, 4.11, 4.12): el usuario puede "disparar" a una pieza, la cual se activa. Al activarse, se crean ejes para moverla, rotarla o redimensionarla de una manera intuitiva y rápida, pero poco precisa. El usuario puede cambiar entre los distintos ejes pulsando el botón A del controlador derecho. Esta herramienta es muy útil para mover las piezas por el escenario y redimensionarlas de una manera aproximada.

- **Vaciado** (4.15): El usuario puede realizar un vaciado entre 2 piezas. Esta herramienta es muy útil para piezas con agujeros, o cuando se necesite eliminar material. Para usarlo, tiene que activar la función de vaciado, disparando al raycast correspondiente. Acto seguido tiene que seleccionar 2 piezas. La primera que seleccione será la original y la segunda la que realice la función de vaciado sobre la primera.

- **Movimiento por puntos** (4.16): esta herramienta sirve para recolocar las piezas de manera relativa entre ellas. Esta herramienta es muy útil para hacer que dos ejes sean coaxiales, o fijar que dos caras sean paralelas. Cuando el usuario selecciona la herramienta, se generan unos puntos característicos de todas las figuras en la escena. El usuario puede elegir que el movimiento sea seleccionando 1, 2 o 3 puntos en cada pieza. Cuando selecciona 1 punto en cada pieza, estos puntos se vuelven coincidentes.

Cuando selecciona 2 hace que ambos segmentos sean paralelos. Cuando selecciona 3 hace que los planos que forman dichos puntos sean paralelos.

- **Creación de grupos** (4.17): El usuario puede unir varias piezas en un mismo grupo. De esta manera, el nuevo grupo se comportará como una pieza individual. Es muy útil cuando hay partes ya completadas de una pieza compleja. En lugar de mover o redimensionar todas las piezas que la componen, mueve la pieza en su totalidad. Para activar esta función disparar al canvas correspondiente y seleccionar todas las piezas que quiera agrupar. Cuando estén todas las piezas que quiera seleccionadas, deberá disparar de nuevo al canvas de creación de grupos para crear el grupo.

- **Añadir cotas** (4.18): Para que el usuario tenga información sobre las medidas de cada pieza, se ha habilitado la función de cotas, que muestra las dimensiones de ancho, largo y altura de las piezas que estén seleccionadas. Para poder activar esta función, deberá disparar al canvas correspondiente. Cuando seleccione una pieza, se le mostrarán sus dimensiones. Para desactivar la opción, deberá disparar al mismo canvas.

- **Trasladar, rotar y redimensionar exacto** (4.19): Para añadir la precisión necesaria en el diseño industrial, se da la posibilidad de que los usuarios cambien los parámetros de las piezas de manera manual, a través de un teclado numérico. Para ello deben de activar la función en el canvas “Exact positioning” y seleccionar la pieza que quieren editar. Para cambiar uno de sus parámetros deberán disparar a la casilla correspondiente a dicho parámetro, introducir el nuevo valor en el teclado numérico y disparar al botón “Enter”. Se puede resetear la posición y rotación si se dispara al botón “Reset position”.

- **Duplicar objetos** (4.20): El usuario tiene la opción de duplicar piezas o grupos para ahorrar trabajo repetitivo. Para ello, primero deben de seleccionar la pieza que quieren duplicar. Posteriormente, deben de disparar al canvas “Duplicate”. Acto seguido, se crea una copia de la pieza previamente seleccionada.

- **Guardar / Exportar piezas** (4.21): Se ha habilitado la función para que el usuario pueda exportar las piezas creadas en formato .fbx. Para ello, debe de seleccionar la pieza que quiere exportar y posteriormente disparar al canvas “Save”.

- **Importar en SolidWorks** (4.24): Se explica de manera detallada el proceso para importar las piezas en programas CAD 3D profesionales como SolidWorks. Para ello, hay que pasar la pieza por el programa intermedio Fusion 360, y finalmente importarla en SolidWorks.

1.5 PRINCIPALES PROBLEMAS

En este apartado voy a introducir los principales problemas que tuve mientras realizaba el proyecto. Ignoro los problemas triviales que surgieron por falta de conocimiento y me centro en aquellos que cualquier persona se encontrará al hacer un programa similar.

- **Problema de dimensionado** (4.11.3): Al cambiar la escala de un objeto, también se cambia la escala de sus hijos. Esto provoca deformaciones indeseadas en los ejes de las piezas, que están asignados como hijos de las piezas. La solución fue crear un objeto vacío intermedio y aplicarle la escala inversa.

- **Problema de conversión a SolidWorks** (4.24): Este es un problema fundamental al trabajar con mallas e intentar exportar a SolidWorks, donde sus archivos están definidos con parámetros. He encontrado una solución a través del programa Fusion 360.

- **Problema de vaciado y las mallas** (4.15.2): Al hacer la operación de vaciado muchas veces sobre la misma pieza, se pueden observar errores en la malla. Esto es debido a la complejidad de volver a calcular las mallas al hacer el vaciado. Se puede intentar limpiar la malla con algunas funciones para disminuir el problema.

- **Problema de posición de los ejes** (4.15.1): Al crear la pieza resultante con el vaciado y los grupos, su centro de referencia no coincidía con su centro de masas. Esto se solucionó moviendo la pieza al origen, para crearla ahí, y luego devolverla al lugar donde estaba.

- **Gran número de funciones y posibilidades**: este fue un problema de gestión para evitar que el usuario activase funciones que no debería. El proyecto tiene muchas herramientas y hay que evitar que el usuario active una cuando no debe. Esto lo conseguí principalmente a través del "EventHub", un script que actúa como intermediario cuando un script quiere usar un método de otro. Así se consigue independencia entre los scripts. La lógica de "RaycastScript" también fue muy importante para mantener un control y orden de las funciones que está usando el usuario.

2 ESTADO DEL ARTE

Debido a la complejidad del problema en su totalidad y a la recién aparición de las tecnologías de RV, todavía no existe ningún programa que trate de afrontar los aspectos tratados en este proyecto. Cabe destacar que en este proyecto se busca crear un programa que permita diseñar piezas de ingeniería. Como es de saber general, en este sector se trabajan con tolerancias muy estrictas, por lo que todas las medidas tienen que ser precisas y exactas. Por ello, no se busca en ningún caso crear una herramienta de diseño artístico, como las muchas ya existentes en el mercado.

Primero de todo, cabe destacar que muchos programas de diseño industrial como SolidWorks o NX ya empiezan a incorporar funciones de realidad virtual.

Ambos programas han añadido en sus últimas versiones la opción de visualizar en un entorno inmersivo los modelos ya creados. En cuanto a la interacción con el diseño, se limita a poder rotarlo para examinarlo desde diferentes ángulos y detalles. También tienen incorporada la opción de colaboración remota, para que varios diseñadores revisen el modelo en el mismo espacio inmersivo al mismo tiempo.

Estas actualizaciones demuestran una vez más el rumbo hacia la realidad virtual que está tomando el sector del diseño industrial. Estas funcionalidades son muy útiles, pero no son suficientes para cumplir con el objetivo de este proyecto. Se pretende ir más allá y poder también modificar dichas piezas dentro del entorno RV. A fecha de hoy, estos programas todavía no tienen incorporadas estas funcionalidades.

Debido a que se pretende simular el entorno de programas de diseño 3D ya existentes, conviene hacer un repaso de algunas de las operaciones existentes en estas aplicaciones. Algunas de estas sirvieron de inspiración para incorporar las herramientas básicas en este proyecto.

Me he inspirado en SolidWorks y NX en las siguientes funcionalidades:

- **Reescalar y extruir (4.11)**: Funciones que sirven para redimensionar las piezas. He unido estas 2 herramientas en una sola, para que permita cambiar la dimensión de la pieza en distintos ejes.

- **Vaciado (4.15)**: una opción muy útil que permite generar una gran variedad distinta de piezas. Se basa en la operación booleana de la resta. Cuando dos objetos están en contacto, se le resta a uno de los objetos la parte de la materia que interseque en ambos cuerpos.

- **Generar grupos (4.17)**: Permite generar un objeto que sea el conjunto de muchos objetos más pequeños. Muy útil cuando se ha acabado una sección de la pieza.

- **Ejes / movimiento de objetos (4.9, 4.10**): en todos los programas se permite mover y rotar los objetos en función de sus ejes de traslación y rotación. Para este proyecto me he ceñido a los ejes locales.

- **Cotas (4.18)**: herramienta fundamental para saber las dimensiones de los objetos. Me he limitado a añadir las cotas en los ejes cartesianos locales.

- **Duplicar objetos (4.20)**: función muy útil cuando el proceso de creación implica diseñar repetidamente la misma pieza.

Me he inspirado en RobotStudio [12] en:

- **Movimiento utilizando puntos (4.16):** esta herramienta sirve para especificar la posición relativa de un objeto respecto a otro. Ello se basa en la posición de varios puntos significativos en cada pieza. De esta manera se puede restringir, por ejemplo, que dos caras de dos piezas sean paralelas, siendo un vértice coincidente en ambas piezas. Esta herramienta permite el movimiento relativo utilizando 1, 2 o 3 puntos.

En la industria del diseño de la realidad virtual, las aplicaciones que se han desarrollado son de uso artístico. Son muy útiles para la creación de modelos donde no se requieran medidas exactas o para bocetos previos. Sin embargo, estos programas no se pueden usar directamente para el diseño industrial. Aun así, también podemos inspirarnos en algunas de las funciones que incorporan para hacer el programa lo más intuitivo y completo posible. Cuatro de los programas más desarrollados hasta la fecha para el diseño artístico de RV son: Gravity Sketch [7], Logitech VR Ink Pilot Edition [9], Shapelab [13] y Quill [14].

- **Gravity Sketch**: es un programa que permite a los usuarios diseñar modelos 3D desde cero. Todo ello dentro del entorno de realidad virtual. Su característica más distintiva es que permite esbozar en el espacio 3D de una manera intuitiva, como si estuviésemos pintado en el aire. Además de los bocetos, también permite rellenar con material las zonas esbozadas.

  Lo he probado antes de iniciar el proyecto para poder inspirarme en alguna de sus funciones. Entre ellas, me he inspirado en la creación de formas primitivas básicas y la posterior modificación de dichas formas. Entre dichas formas primitivas podemos incluir cubos, esferas, cilindros y muchas más. En mi proyecto me centré en las tres citadas.

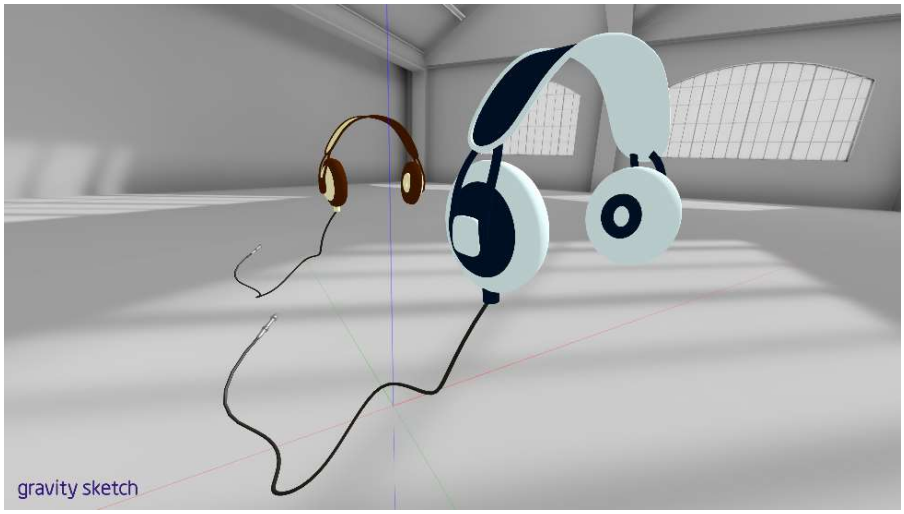


Ilustración 6: Prueba de diseño por Pedro Bodega en Gravity Sketch

Después de probar el programa y completar uno de sus tutoriales, he concluido que no sirve para el propósito de este proyecto. Es un programa bastante completo e intuitivo, sin embargo, carece de medidas. Ello no quita que no pueda tener un uso en el diseño de RV. Es ideal para bocetar los diseños, ya que es bastante intuitivo. Puede ser muy útil para los futuros diseñadores de RV en ingeniería para hacer bocetos previos, y que posteriormente diseñen la pieza final en un programa como el que se plantea en este proyecto.

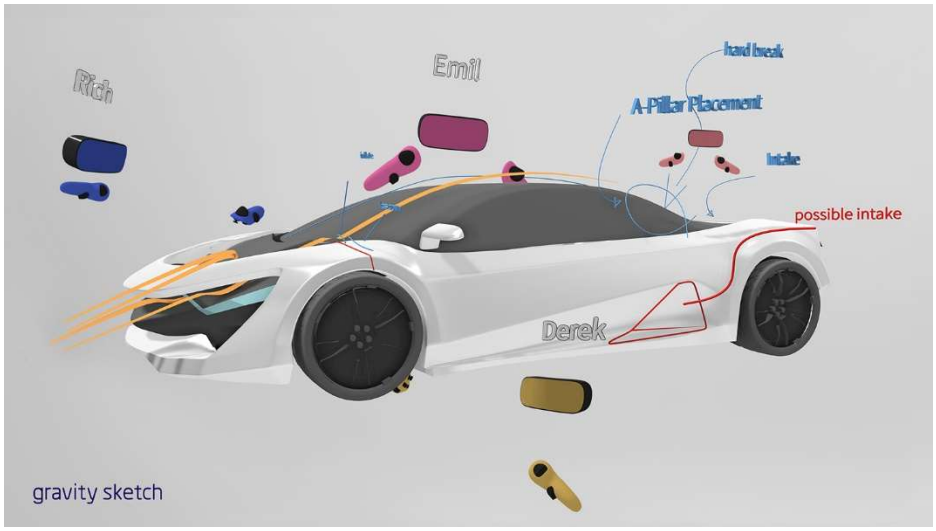


Ilustración 7: Modelo 3D colaborativo en Gravity Sketch

- **Logitech VR Ink Pilot Edition**: es un programa con un objetivo similar a Gravity Sketch. Ambos se centran en la parte más creativa del diseño industrial, hacer un primer boceto en 3D del modelo. Todo ello sin medidas exactas.

Aunque pueda parecer que se pueden añadir pocas funciones a Gravity Sketch, Logitech VR Ink Pilot logra darle un enfoque ligeramente diferente a este proceso de diseño. Primero, cabe recalcar que Logitech es una empresa líder en tecnología mundial. Su apuesta por el diseño en RV no hace más que reafirmar que esta posiblemente va a ser la manera predilecta del diseño industrial en el futuro.

La principal innovación que presenta este programa es que no hace uso de los controladores que vienen con las gafas de RV, sino que requiere del uso de un lápiz táctil VR. Ello permite sea un proceso más intuitivo, estando ya acostumbrados a los lápices tradicionales. Al igual que Gravity Sketch, permite diseñar directamente en 3D, pero en este caso también en un plano 2D. Se puede dibujar el modelo por capas en diferentes planos en 2D con diferentes vistas, y posteriormente componer el modelo 3D resultante. Todo ello, sin medidas exactas. Esta idea no la he incorporado al proyecto debido a su complejidad, pero puede ser de gran interés para futuras actualizaciones. Con ello, se permitiría crear una gran variedad de piezas mucho más complejas.

Por último, Logitech asegura que con este programa, son un 40% más rápidos y precisos en el proceso de diseño [8].

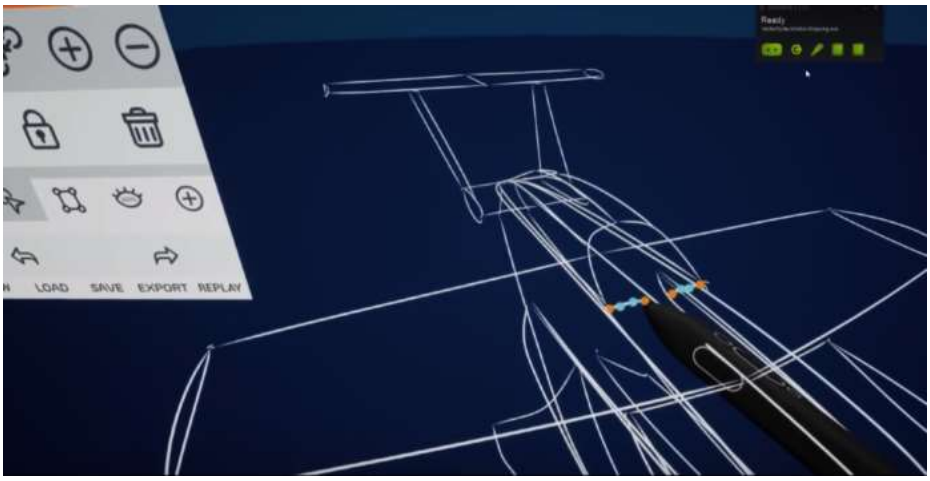


Ilustración 8: Interfaz de Logitech VR Ink Pilot Edition

- **Shapelab**: este programa se centra más en el diseño artístico de modelos, no está pensado para usar en diseño industrial. Por resumirlo en pocas palabras, es como una versión en realidad virtual de programas como Blender [15] y Fusion 360.

  Sin embargo, tiene una función en específico que puede ser útil adaptar a nuestro proyecto. Para diseñar en Shapelab, se parte de una masa, y se va "amoldando". Ello se hace de manera muy intuitiva, seleccionando con el

controlador la zona que se quiere estirar, tal y como lo haríamos en el mundo real. Para imaginarlo mejor, es similar a como si quisiésemos estirar un bloque de plastilina.

En este concepto me he inspirado para implementar las herramientas de escalado en mi programa. Para simplificarlo, he reducido el escalado en los ejes x, y, z.

Otra herramienta interesante de Shapelab es la de generar material en el espacio 3D. Se genera de una manera similar a como si se estuviese "pintando en el aire". El problema que tiene es que ninguna medida es exacta, por lo que tampoco es válido para el objetivo de este proyecto.

- **Quill**: es otro programa que se centra más en el diseño artístico, similar a Shapelab. Se diseña en 3D dentro de un entorno RV. En el diseño industrial, solo sería útil para bocetar el modelo. Sus herramientas son similares a las ya mencionadas en otros los programas en esta misma sección.

2.1 Diseño paramétrico en Unity, Archimatix:

Muchos de los programas CAD como SolidWorks, trabajan con el diseño paramétrico. Esto consiste en que el modelo está definido por parámetros, como altura, ancho, radio, etc. Con este diseño paramétrico, sería posible especificar las dimensiones del modelo de manera exacta, condición fundamental para que sea útil en el diseño industrial. El entorno con el que trabajo, Unity, cuenta con un "add on" o "plugin" que permite diseñar de manera paramétrica en Unity. El "add on" se llama Archimatix [16].

Después de probarlo, funciona exitosamente, de manera similar a como se edita en SolidWorks. Sin embargo, tiene un problema para implementarlo en este proyecto. Está hecho para diseñar en la interfaz de Unity, con ratón y ordenador, no para editar los modelos dentro de la realidad virtual. Por ello, no es útil en este proyecto, ya que el propósito principal es que la edición ocurra en un mundo inmersivo.

3 METODOLOGÍA

3.1 UNITY Y C#

Para el entorno de desarrollo he optado por utilizar el programa Unity. Es una plataforma que permite a los desarrolladores diseñar aplicaciones y programas en 2D, 3D y en RV.

Las dos grandes plataformas para desarrollar aplicaciones en RV son Unity y Unreal Engine [17]. Me decidí por la primera por los siguientes motivos:

- Amplia gama de plugins y comunidad de desarrolladores.
- Muchas librerías ya vienen incorporadas para poder programar a más alto nivel.
- Lenguaje de programación C# en lugar de C++. C# es más fácil de aprender e implementar.
- No necesito los renderizados de alta calidad, característica destacada de Unreal Engine.

Unity es un programa que encuentra el equilibrio entre una interfaz de usuario amigable, y una gran cantidad de herramientas. Otra de las ventajas de este programa es que permite implementar una función de muchas maneras distintas. Al principio esto puede suponer agobio e indecisión por la amplia gama de posibilidades, pero a medida que se va usando el programa, se descubren métodos para realizar la misma operación más rápido. Un ejemplo es la implementación de los botones explicada en la sección 4.19. Aprendí a llamar a métodos a través de los botones a un nivel más alto. Al principio lo hacía mediante eventos.

Cuando me adentré en este proyecto, no había tenido ningún contacto con la realidad virtual, Unity, ni con C#. Por ello, fue necesario un periodo de aprendizaje de estas herramientas antes de abordar el proyecto.

Para aprender el entorno de Unity y de C#, completé el curso de Udemy [18]: "Learn Unity Games Engine & C# By Creating A VR Space Shooter" [19]. Es un curso de 5,5 horas de duración de vídeo que enseña a programar un juego básico en Unity desde 0. En el curso combina herramientas de alto nivel (con la interfaz de Unity) y más bajo nivel (código). También explica a hacer varias funciones de diferentes maneras. El curso fue muy útil para comprender las posibilidades dentro de Unity, y para aprender a escribir los códigos de una manera ordenada y conexionada.

A pesar de que Unity permite hacer muchas funciones en su interfaz sin necesidad de programar, para poder hacer el programa lo más general y personalizable posible, opté por programar la mayor parte del programa. C# en Unity trabaja principalmente con programación orientada a objetos. Unity ya viene incorporado con una inmensa cantidad de librerías con clases y métodos

ya creados. Esto facilitó inmensamente la creación del proyecto. Sin embargo, añadió la dificultad de tener que descubrir qué métodos y clases ya están creados para facilitar el trabajo. Para todo ello fue de gran ayuda el curso de Udemy [19].

Por lo general, se crea una clase por cada script que se escribe en C# en Unity. Los scripts son generalmente objetos en el programa. Contienen información de cómo se tiene que comportar el objeto en distintas circunstancias. Adicionalmente, los scripts suelen estar de alguna manera relacionados entre ellos. El funcionamiento de cómo están estructurados los scripts en este proyecto se explicará más adelante 3.6.

3.2 VISUAL STUDIO

Como editor de código he decidido utilizar Visual Studio [20]. Es una herramienta sencilla de empezar a utilizar, con detector de fallos de compilación antes de incluso compilar. Otra ventaja de Visual Studio es que se puede conectar con Unity. De esta manera, ayuda a autocompletar y corregir los métodos de librerías de Unity. También detecta nombres de scripts y objetos que estén en el proyecto con el que estemos trabajando en Unity. Para habilitar esta función hay que seguir una serie de pasos [21]:

En **Visual Studio**: Tools → Get Tools and Features → Desarrollo de juego con Unity → Modificar en instalar.

En **Unity**: Edit → Preferences → External Tools → External Script Editor → Microsoft Visual Studio.

Con estos pasos, Unity y Visual Studio ya están conectados en tiempo real.

3.3 FUSION 360

Otro de los objetivos del proyecto es poder pasar la pieza creada en Unity a programas CAD como SolidWorks o NX. En este caso escogí SolidWorks porque estaba más familiarizado. El paso del archivo de la pieza desde Unity a SolidWorks no es trivial, y tiene un problema fundamental de falta de información en el formato de Unity 4.24. Por ello, es necesario utilizar el programa Fusion 360 para corregir este fallo. Es un programa muy útil y completo para el diseño industrial. Permite trabajar con modelos 3D basados en parámetros, como también trabajar con mallas y superficies. En mi proyecto Fusion 360 sólo sirve como un intermediario para corregir el archivo y exportarlo a SolidWorks. Es esta habilidad que tiene para trabajar con mallas y también modelos basados en parámetros la razón por la que lo uso.

3.4 NOTION

Para la organización y seguimiento del proyecto he utilizado Notion [22]. No es imprescindible para este proyecto, pero ha sido parte de la metodología de trabajo. Para seguir de manera ordenada el trabajo realizado todos los días, he ido escribiendo el progreso en Notion. Para cada día incluyo también las fuentes consultadas, problemas y copia y explicación de los scripts escritos hasta la fecha. Adicionalmente, resumo los puntos clave de las reuniones con el tutor de este proyecto José Antonio Rodríguez Mondejar. Por último, también tenía un registro de las herramientas aún por implementar y las terminadas.

3.5 META QUEST 2

Como dispositivo de realidad virtual he utilizado las Meta Quest 2. La principal razón es la gran comunidad de desarrolladores y el largo periodo en el que han estado a la venta. Son compatibles con una inmensa cantidad de aplicaciones de realidad virtual, incluido Unity. También existen numerosos foros en internet donde se ayuda con problemas relacionados con las Meta Quest 2.

3.6 DISEÑO DEL SISTEMA Y ESTRUCTURA DEL SOFTWARE

Desde el comienzo, he procurado escribir el código de una manera ordenada y clara, para tener el menor número de problemas al añadir posteriormente nuevas funcionalidades. El programa cuenta con un total de 22 scripts de código. Estos scripts están comunicados entre sí, por lo que hay que ser muy ordenado para que al cambiar algo en un script, afecte lo mínimo posible al resto. En la Ilustración 9 se puede observar todas las conexiones de los scripts entre sí. Se muestra cada relación con más detalle en los diagramas expuestos al principio de cada función. Se aclara que las flechas en la Ilustración 9 que salen de “EventHub” son bidireccionales; existe comunicación en ambos sentidos.

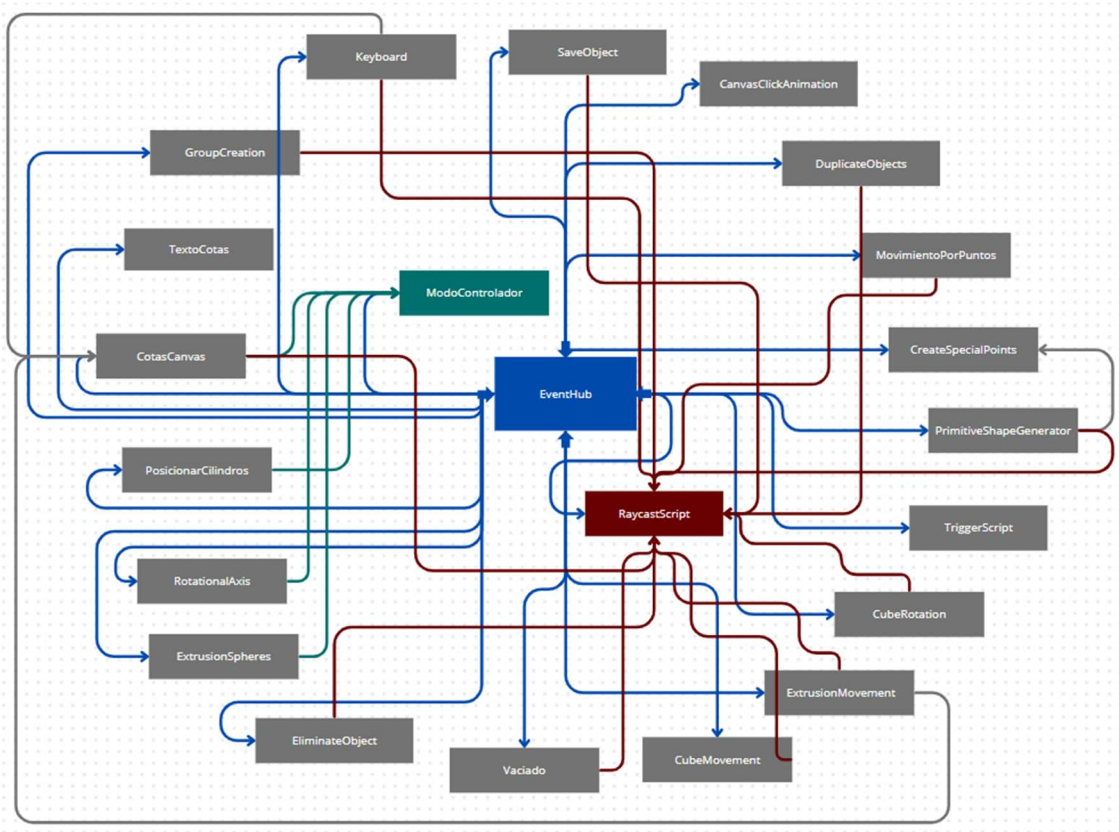


Ilustración 9: Diagrama de intercomunicación entre todos los scripts

Todos los scripts contienen exclusivamente la definición de una clase, con diferentes argumentos y métodos. A partir de ahora utilizaré indistintamente el nombre del script o el de la clase que contiene (es el mismo). Todas las clases son dinámicas menos una, que es estática. Es a la que he denominado EventHub. Este script es el corazón del programa y es el encargado de conectar todos los scripts entre sí. Al ser una clase estática, no va asignado a ningún objeto del programa. Por el contrario, el resto de clases (o scripts) van asignados a los objetos, pudiendo tener así distintas instancias de cada clase.

El EventHub se encarga de controlar todos los eventos del programa, simplemente actúa como un intermediario. Cuando se quiere disparar el método de un script, cuando se da cierta condición en otro script, se usan los eventos. Esta comunicación se podría hacer directamente entre ambos scripts, sin embargo, ello resulta en desorden y complejidad innecesaria. Para ello he creado el EventHub. Cuando se cumple cierta condición en un script, se dispara un evento, definido en el EventHub. Inmediatamente, el EventHub hace disparar un método definido en otro script distinto.

Para conseguir un programa más personalizable y general, la mayor parte está escrito en C# con scripts, pero también se ha utilizado la interfaz de Unity (más alto nivel) para pequeñas implementaciones.

Una de ellas es el uso del paquete XR Interaction Toolkit, que facilita la lectura de los controladores del Meta Quest 2. Después de configurarlo 4.3, he creado un script (TriggerScript) que está a la espera constante de que se pulse alguno de los botones de interés. Cuando éstos son pulsados, se activa un evento. En los distintos scripts, puedo escribir que estén constantemente escuchando a estos eventos, y que activen un método cuando son disparados. Este es un ejemplo de la verdadera utilidad de los eventos.

Cuando se pulsa el gatillo del controlador, se dispara un raycast definido en el script "RaycastScript". El raycast es una clase ya definida en Unity. Se comporta de manera similar a un láser que sale desde el controlador, y va en dirección recta hasta donde éste esté apuntando. Las interacciones entre el usuario y el entorno, se hacen siempre a través de ese raycast. Eso incluye cualquier interacción con la interfaz para seleccionar las distintas herramientas o para modificar las piezas.

El script "Raycast" es el encargado de generar este raycast y de toda la lógica necesaria para establecer prioridades en el caso de que el raycast intersecciones con más de un objeto. También se encarga de disparar eventos en diferentes scripts. Esto es muy útil para ejecutar una acción cuando el raycast ha impactado en un objeto determinado. A partir de ahora se va a usar el término disparar para definir la acción de pulsar el gatillo, lo cual hace salir el raycast. No confundir con la acción de cuando un evento se dispara, que quiere decir que se ha activado.

En este proyecto, el usuario tiene la posibilidad de crear 3 formas primitivas: cubo, cilindro o esfera. Puede crear estas formas disparando hacia los "canvas" (o pantallas) que tienen la imagen de dicha forma. Una vez creada la forma primitiva, si le dispara, esa pieza se activa (selecciona). Al activarse, el usuario puede alternar entre 3 modos pulsando repetidamente el botón "A" del controlador. El primer modo sirve para mover la pieza en los ejes cardinales locales. El siguiente modo sirve para rotar la pieza. El último modo sirve para redimensionar la pieza en los distintos ejes.

Además, el usuario tiene más canvas en el escenario, que le permiten habilitar diferentes herramientas para modificar las piezas. Entre las funciones se encuentran:



Ilustración 10: Canvas con las distintas herramientas disponibles

- **Vaciado (4.15)**: El vaciado necesita que 2 objetos estén interseccionando entre sí. Devuelve un objeto que es el resultado del primero, menos la parte interseccionando. Muy útil para hacer agujeros o para eliminar material de una pieza.

- **Eliminar (4.14)**: Función básica para eliminar una pieza o grupo.

- **Duplicar (4.20)**: duplica la pieza seleccionada.

- **Agrupar (4.17)**: permite agrupar varias piezas para que se comporten como una sola.

- **Movimiento por puntos (4.16)**: restringe la orientación de una pieza en función de otra. Este movimiento puede ser seleccionando 1, 2 o 3 puntos de cada pieza. Si se selecciona 1 punto en cada pieza, dichos puntos se vuelven coincidentes. Al seleccionar los segundos puntos, hace que las líneas que unen los puntos de cada pieza sean paralelas. Al seleccionar los tres puntos en cada pieza, los planos definidos por los 3 puntos seleccionados se vuelven paralelos.

- **Activar / desactivar cotas (4.18)**: Activa y desactiva las cotas en las piezas que estén activas (o seleccionadas). Las cotas se muestran en las dimensiones x, y, z locales de dicha pieza.

- **Guardar pieza (4.21)**: exporta la pieza o grupo seleccionado en un fichero predeterminado en formato .fbx.

Adicionalmente, también cuenta con la opción de input manual de posición, rotación y dimensión. Para ello, se muestra un display con la información de estos parámetros (en x, y, z) para la pieza seleccionada. El usuario puede interaccionar con un teclado a través del raycast para modificar manualmente las características (posición, rotación y dimensión) de la pieza.

Como complemento, el usuario puede ver dentro del mundo inmersivo una imagen del plano o boceto de la pieza que esté diseñando. Esto es de vital importancia para trabajar de manera más cómoda y fluida.

En general, hay un gran número de funciones y herramientas implementadas en el proyecto. Esto da lugar a un gran número de scripts que están intercomunicados entre ellos. Además, el usuario tiene una gran libertad en sus acciones, como interactuar con diferentes objetos en las distintas funciones. La gran combinación de interacciones del usuario con el entorno es inmensa, las cuales hay que tener en cuenta. El código está preparado para evitar que el usuario interaccione con elementos cuando no debería.

Por último, se escribieron un gran número de mensajes de aviso de error con su causa por si el programa fallaba. Este aspecto fue de gran ayuda mientras se escribía el código.

4 IMPLEMENTACIÓN DEL MODELO

En esta sección voy a explicar todas las herramientas que he implementado de manera detallada. También explicaré los problemas principales que tuve, y la solución encontrada. Esta sección puede también ser utilizada como guía para las personas que traten de abordar un proyecto similar.

4.1 CONFIGURAR LAS META QUEST PARA MODO DESARROLLADOR

Para poder compilar y ejecutar un programa de Unity en las Meta Quest, hace falta habilitar el modo desarrollador en dichas gafas. Para ello, hay que seguir los siguientes pasos:

Primero hay que ir a la página web de Oculus: "https://developer.oculus.com/" [23] desde cualquier dispositivo. Posteriormente, iniciar sesión y verificar dicho inicio de sesión. En el menú desplegable "Select Organization", crear una nueva organización (el nombre elegido para la organización es irrelevante). Conectar las gafas de Meta Quest al móvil o al ordenador y abrir la aplicación de Meta Quest. Hacer click en el nombre del dispositivo mostrado en la zona superior → Headset settings → Developer Mode → Activar Debug Mode.

4.2 PREPARAR EL ENTORNO DE UNITY

En el Unity Hub, hay que crear un nuevo proyecto. Para la plantilla del proyecto he decidido usar 3D Universal Render Pipeline (URP). Es el renderizado más común y estándar en Unity, que soporta un gran número de materiales y texturas.

Una vez creado el proyecto, hay que habilitar que se pueda ejecutar dentro de las Meta Quest 2. Para ello hay que instalar el paquete de Unity XR Plug-in Management: File → Build Settings → Project Settings → XR Plug-in Management → Install. Una vez instalado, hacer click en el icono de Android → Oculus. Esto es necesario para que el programa se pueda ejecutar en las Meta Quest.

4.3 XR INTERACTION TOOLKIT

XR Interaction Toolkit es un paquete descargable en Unity muy útil para implementar la realidad virtual y las Meta Quest en el proyecto. Proporciona la base para evitar escribir la configuración inicial de los controladores y la cámara. Por ejemplo, detecta que al mover la cabeza con las gafas Meta Quest, se mueva la cámara de Unity de la misma forma. También viene con el código escrito para leer los botones de los controladores de una forma mucho más sencilla. Por último, se viene incluido con clases y métodos que son de gran utilidad.

Para instalarlo, en Unity: Window → Package Manager → Buscar "XR Interaction Toolkit" → Samples → Import Starter Assets → Install.

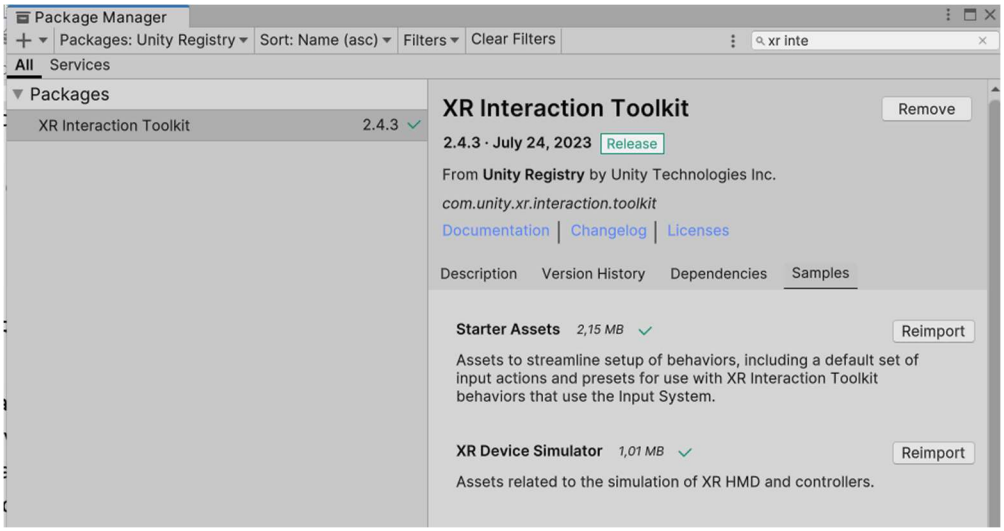


Ilustración 11: Ejemplo de instalación de XR Interaction Toolkit

Para añadir la cámara XR: Eliminar Main Camera → Click derecho → XR → XR Origin (Action-based).

Esto también nos añadirá los controladores derecho e izquierdo. Por defecto, la configuración de estos controladores viene vacía, pero utilicé el ejemplo que viene con los Assets de XR para añadir la configuración base de los controladores. El nombre de estos scripts son XRI Default Right (o Left) Controller. Añadir estos scripts en Project Settings → Preset Manager, y volver a añadir XR Origin. De esta manera ya aparecerá con los valores predeterminados.

4.4 HABILITAR BOTONES DE LOS CONTROLADORES

Para este proyecto solo he necesitado usar del controlador un gatillo y un botón. Como ya he explicado antes, el gatillo se encargará de disparar el raycast y el botón para alternar entre los modos de mover, rotar y dimensionar. Sin embargo, lo que voy a explicar en este apartado solo sirve para que el programa interprete que se ha pulsado dicho botón.

En Project ir a Assets → Samples → XR Interaction Toolkit → 2.4.3(depende de la versión) → Starter Assets → XRI Default Input Actions. En Action Maps crear uno nuevo [24].

Para detector los botones, hay que crear nuevas "Actions", una para cada botón. Según el tipo de botón, tenemos que definir un "Action Type". Para el caso del botón he usado Action Type: Button. Esto es porque el botón solo puede tener valor de 0 o 1, y me devuelve dicho valor. Más documentación acerca de "Action Type" [25]. En el caso del gatillo he seleccionado "Action

Type": Value, "Control Type": Any. Esto hace que en el momento que el valor del gatillo deja de ser 0, se activa una vez el botón. Más documentación acerca de "Control Type" [26].

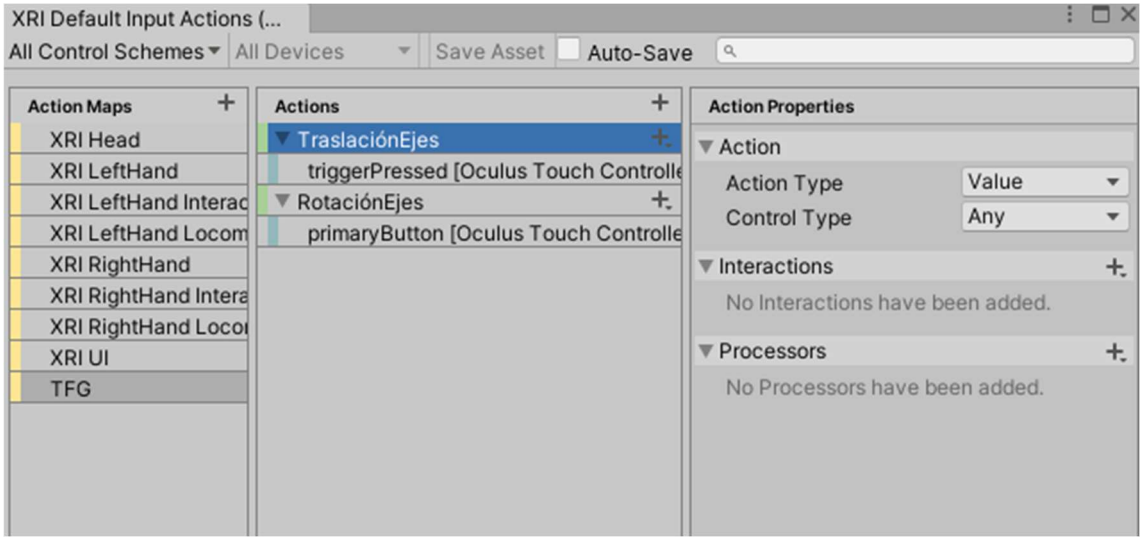


Ilustración 12: Configuración de los "Input Actions" en Unity

Ahora hay que especificar los botones del controlador que queremos usar. Dentro de cada acción, seleccionar la opción de "Add Binding". En "Path" → XR Controller → Oculus Touch Controller. Para el gatillo he seleccionado la opción de "triggerPressed" y para el botón "primaryButton". Éstos corresponden con el gatillo y con el botón A de los controladores respectivamente. Con esto se han creado 2 instancias de la clase InputActionReference, localizadas en:

Assets → Samples → XR Interaction Toolkit → 2.4.3(depende de la versión) → Starter Assets.

4.5 SCRIPT PARA LEER LOS BOTONES Y EVENTHUB

He creado un script (y una clase) llamado TriggerScript (10.2) que sirve para disparar eventos cada vez que se pulsa el botón o el gatillo. Este script tiene que ir adherido a Left Controller y Right Controller, dentro de XR Origin → Camera Offset.

Dos de los argumentos fundamentales son:

```
public InputActionReference triggerActionReference; //Lectura del gatillo

public InputActionReference sButtonActionReference; //Lectura del botón
```

Estos argumentos son de tipo público y de la clase "InputActionReference", por lo que se deben de asignar el Inspector de Unity. Las instancias son las creadas en 4.4.

En el EventHub se define el delegado y los eventos estáticos:

```
public delegate void TriggerAction();
public static event TriggerAction OnTriggerPressed;
public static event TriggerAction OnTriggerReleased;
```

También se define, para configurar los eventos:

```
public static void TriggerPressed() => OnTriggerPressed?.Invoke();
public static void TriggerReleased() => OnTriggerReleased?.Invoke();
```

Estas líneas de codigo son necesarias para poder ejecutar funciones en distintos scripts cada vez que se pulsa el botón o el trigger.

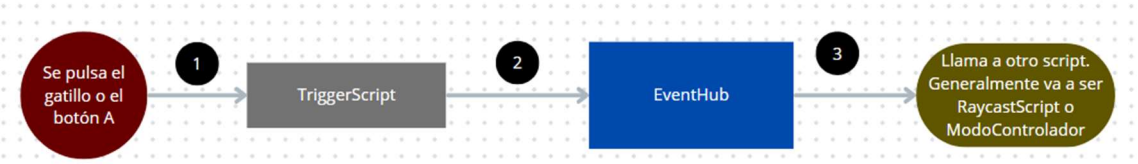


Ilustración 13: Diagrama a alto nivel de la lógica al pulsar un botón

Finalmente, es necesario escribir el código en "TriggerScript" para que los eventos estén escuchando a los botones:

```
    private void OnEnable()
    {
        triggerActionReference.action.started += OnTriggerPressed;
        triggerActionReference.action.canceled += OnTriggerReleased;
        triggerActionReference.action.Enable();

        sButtonActionReference.action.started += OnButtonPressed;
        sButtonActionReference.action.Enable();
    }
```

```
    private void OnDisable()
    {
        triggerActionReference.action.started -= OnTriggerPressed;
        triggerActionReference.action.canceled -= OnTriggerReleased;
        triggerActionReference.action.Disable();

        sButtonActionReference.action.started -= OnButtonPressed;
        sButtonActionReference.action.Disable();
    }
```

Esto hace que cada vez que cuando se pulse y suelte el gatillo se ejecuten los métodos OnTriggerPressed y OnTriggerReleased respectivamente. Cuando se pulsa el botón se ejecuta OnbuttonPressed. Estos métodos son los encargados de disparar los eventos, especialmente en las siguientes líneas de código:

```
EventHub.TriggerPressed(); //Dispara el evento de pulsar gatillo
EventHub.TriggerReleased(); //Dispara el evento de soltar gatillo
EventHub.TriggerModeChange(); //Dispara el evento de pulsar el botón
```

4.6 CREACIÓN Y CONFIGURACIÓN DEL RAYCAST

El raycast es un láser que sale de cada uno de los controladores. Más específicamente, es un objeto de Unity que pertenece a la clase UnityEngine, propia de Unity. Se define por un origen y una dirección. Esta clase tiene métodos para saber los obstáculos con los que ha chocado el raycast y para sacar información útil de ellos. El raycast en sí es transparente, le he añadido forma y color de láser para que sea más intuitivo aprovechando funciones del "XR Interaction Toolkit".

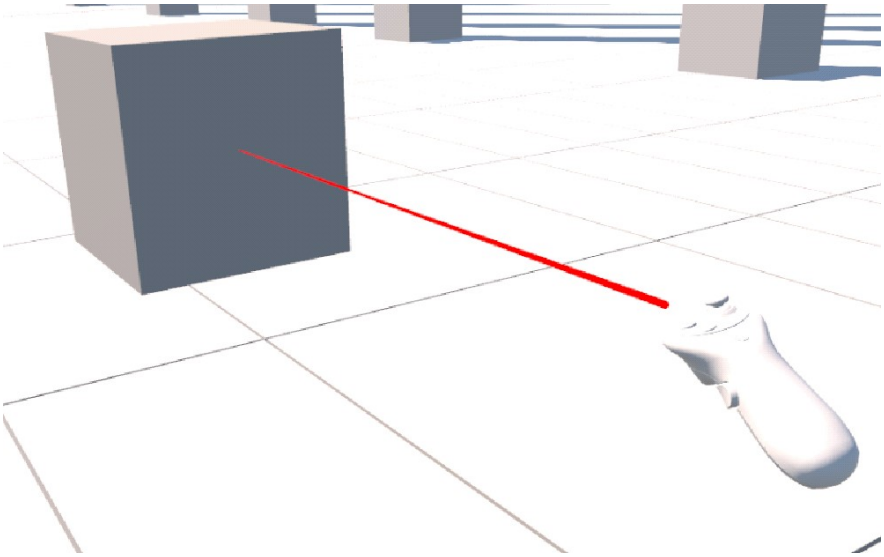


Ilustración 14: Imagen que ilustra el raycast impactando en un objeto en Unity

Para la posición y dirección del raycast, he creado un objeto vacío llamado rightRaycastOrigin y lo he asignado como hijo del objeto "Right Controller". De tal manera que la jerarquía es:

XR Origin → Camera Offset → Right Controller → rightRaycastOrigin.

Es importante recalcar que la posición y rotación inicial local de rightRaycastOrigin tiene que ser 0 en todos los ejes. Al asignarlo como hijo de Right Controller, se moverá de igual manera que el controlador derecho, por lo que el raycast siempre estará apuntando hacia donde apunte el controlador.

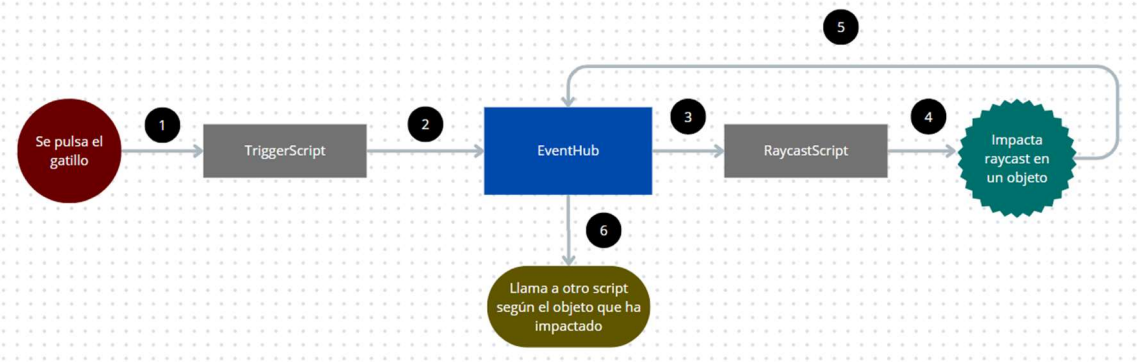


Ilustración 15: Diagrama a alto nivel del funcionamiento del raycast

He decidido no añadir raycast al controlador izquierdo porque no se necesitan tantos botones y así se evitan posibles problemas de superposición de órdenes de los botones de ambos controladores (por ejemplo, que estén pulsados a la vez los botones de ambos controladores).

El raycast lo he definido en el script llamado "RaycastScript" (10.3):

```
RaycastHit[] hits = Physics.RaycastAll(raycastOrigin.position, raycastOrigin.forward, 800); //Guarda en hits todos los objetos que han sido golpeados por el raycast
```

El script "RaycastScript" lo he asignado al objeto "Right Controller". El argumento principal de este script es:

```
[SerializeField] private Transform raycastOrigin; //Objeto con la posición y dirección del raycast
```

"[SerializeField]" se escribe para que se le pueda añadir raycastOrigin en el inspector. En este caso he añadido el objeto que he creado antes: rightRaycastOrigin.

El script "Raycast" está constantemente escuchando al evento de pulsar el gatillo, y cuando lo pulsa, ejecuta la función "HandleTriggerPressed".:

```
void OnEnable()
{
    EventHub.OnTriggerPressed += HandleTriggerPressed;
}

void OnDisable()
{
    EventHub.OnTriggerPressed -= HandleTriggerPressed;
}
```

El script "RaycastScript" se encarga de toda la lógica para activar diferentes funciones o acciones en función de los objetos en los que haya impactado el raycast. Todo ello funciona disparando eventos, que a través del "EventHub" activa métodos en otros scripts.

En la función "HandleTriggerPressed", primero se almacena en la variable "hits" todos los objetos con los que ha impactado el raycast y posteriormente se ordenan todos los objetos en función de su distancia con el usuario:

```
//Ordena por distancia para ejecutar el primero que encuentre
var sortedHits = hits.OrderBy(hit => hit.distance).ToArray();
```

Cuando el raycast impacta en una pieza y no haya ninguna herramienta activa, esta pieza se activará (seleccionará). Para ello dispara los eventos:

```
EventHub.ResetActivation(); //Desactiva todas las piezas

EventHub.ActivateObjectHit(hit); //Activa la pieza impactada por el raycast
```

Otro script está suscrito a estos eventos. Cuando son disparados, hace que todas las piezas que antes estaban activas, se desactiven y solo active la impactada por el raycast. Al activarla, la pieza se vuelve semitransparente.

He dividido los objetos que pueden interactuar con el raycast en 2 tipos, los que son prioritarios y los que no. De manera general, los no prioritarios son los canvas (paneles) y las piezas (las piezas que se estén diseñando), mientras que los prioritarios son los objetos que al interactuar con ellos modifican las propiedades de las piezas (ej. Ejes para cambiar su posición).

La principal razón para dividir de esta manera los objetos es permitir a los usuarios seleccionar los ejes para mover las piezas, en la zona en la que los ejes están dentro de las piezas. En otras palabras, cuando se selecciona una pieza, se vuelve semitransparente y según el modo en el que se encuentre, aparecen ejes de translación, rotación o dimensionado. Parte de esos ejes pasa por el interior de las piezas, y como las piezas son semitransparentes, los ejes son visibles. Si los ejes no fuesen prioritarios, se ejecutaría el código del objeto más cercano, la pieza (es el código que activa la pieza, pero ya está activada).

Al establecer los ejes como prioritarios, cuando el raycast impacta en el eje y en la pieza, se ejecuta el código asociado a los ejes, que es el que permite mover la pieza. De esta manera, es mucho más cómodo mover y rotar las piezas.

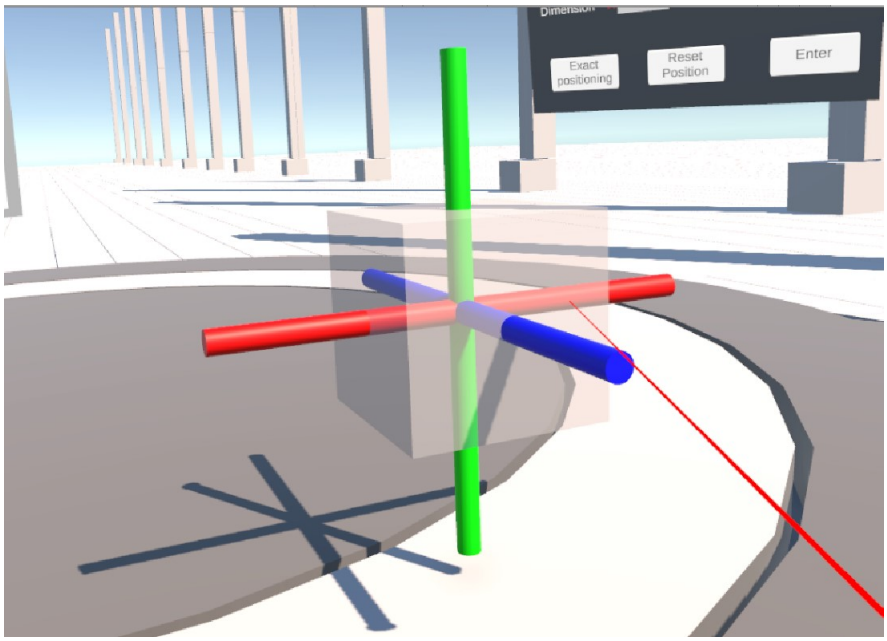


Ilustración 16: Raycast impactando en un eje a través de la pieza transparente

Cuando el raycast no impacta en ningún objeto, se deseleccionan todas las piezas:

```
        //Si no choca con nada, desactiva los objetos

         if (hits.Length == 0 && !vaciado.isVaciado && !group.isGrouping
&& !specialPoints.isActive && !keyboard.exactPositioningActive)

        {

            EventHub.ResetActivation();

        }
```

Cada objeto en la escena suele tener un script asociado o una característica que lo distingue del resto de objetos. Cuando el raycast impacta con los objetos, busco estas características distintivas para saber con qué tipo de objeto ha interseccionado. Por ejemplo, si quiero ver si un objeto tiene un script determidado (En este caso "PrimitiveShapeGenerator"):

```
if (hit.collider.GetComponent<PrimitiveShapeGenerator>())
```

Si en cambio se quiere ver si el objeto tiene un Tag determinado (En este caso "PrimaryObject"):

```
if (hit.collider.gameObject.tag == "PrimaryObject")
```

" RaycastScript" está escrito de tal manera que una vez que encuentre un objeto con el que interactuar, deje de buscar en el resto de objetos con los que ha impactado el raycast. Ello está hecho con una serie de condicionales "if" y "else if" que están siendo recorridos por cada uno de los objetos con los que ha impactado el raycast, mediante:

```
foreach (RaycastHit hit in sortedHits)
```

Una vez que ha entrado en un "if" o "else if", sale también del bucle "foreach", acabando así la función hasta que vuelva a ser llamada.

El funcionamiento del resto de eventos disparados dentro de "RaycastScript" serán explicadas cuando se expliquen las dichas funciones, por lo que será un script de vital importancia y de conexión de todo el proyecto.

4.7 MOVIMIENTO DEL USUARIO

Para facilitar la movilidad por el escenario, he añadido la posibilidad de que el usuario se puede mover a través de los joysticks de los controladores. Esta parte se ha realizado a alto nivel, sin programar en C#. Para ello, he aprovechado de algunos scripts que ya vienen con el paquete XR Interaction Toolkit.

Primero se crea un objeto vacío en la jerarquía de Unity y se le añade el script "Locomotion System". Este script viene con XR. Desde el Inspector, asignamos el campo de "XR Origin" con el objeto del proyecto: "XR Origin (XR Rig)".

En el objeto "XR Origin (XR Rig)" tenemos que añadir dos componentes o scripts: "Continuous Move Provider" y "Continuous Turn Provider". Al añadir estos scripts hay dos opciones, Action-based y Device-based. El primero funciona con la mayoría de gafas RV del mercado y permite un giro continuo moviendo el joystick. El Device-based activa la rotación y el movimiento en función de la posición del controlador. La función que más encaja en este proyecto es Action-based.

Estos scripts solicitan que se les añada un objeto de la clase Locomotion System, que es el creado recientemente.

Por similitud con la mayoría de los controladores en aplicaciones y juegos, se ha decidido que el joystick izquierdo sea el encargado del movimiento y el derecho de la rotación. Para ello, en cada script (Turn Provider y Move Provider), seleccionamos la referencia del controlador que vaya a hacer la acción. Para simplificar el proceso en la referencia, añadimos los InputActionReferences que vienen por defecto con XRI. Estos son: XRI RightHand/Turn y XRI LeftHand/Move.

Finalmente, se pueden modificar algunas características para moverse o girar. Al rotar se puede especificar desde el Inspector la velocidad de giro. Después de varias pruebas la he establecido en 80. La velocidad de movimiento la establecí en 4. Recomendable también deshabilitar la opción de volar. Puede provocar mucho mareo. Por lo general, a más movimiento, más mareo.

Con esto el usuario es capaz de andar libremente por el escenario para observar las piezas desde distintos ángulos sin tener que rotarlas. Puede ser muy útil para observar detalles o posibles errores.

4.8 CREACIÓN DE LAS FORMAS PRIMITIVAS

En este proyecto, la principal manera de diseñar parte de unas piezas base. Se han añadido 3 posibilidades: cubo, esfera y cilindro. En un futuro, se podría ampliar este espectro, como así también permitir al usuario añadir sus propias formas primitivas para que trabaje utilizándolas como una base.

El usuario es capaz de seleccionar en 3 diferentes canvas la pieza que quiere crear. Para ello, tiene que disparar con el raycast en la pantalla con la imagen de la figura que quiere crear, y dicha pieza aparecerá en una zona designada del escenario.

El script que organiza todo este proceso se llama "PrimitiveShapeGenerator" (10.4). Este script va asignado a los 3 canvas. Los canvas se convierten en instancias de esta clase.

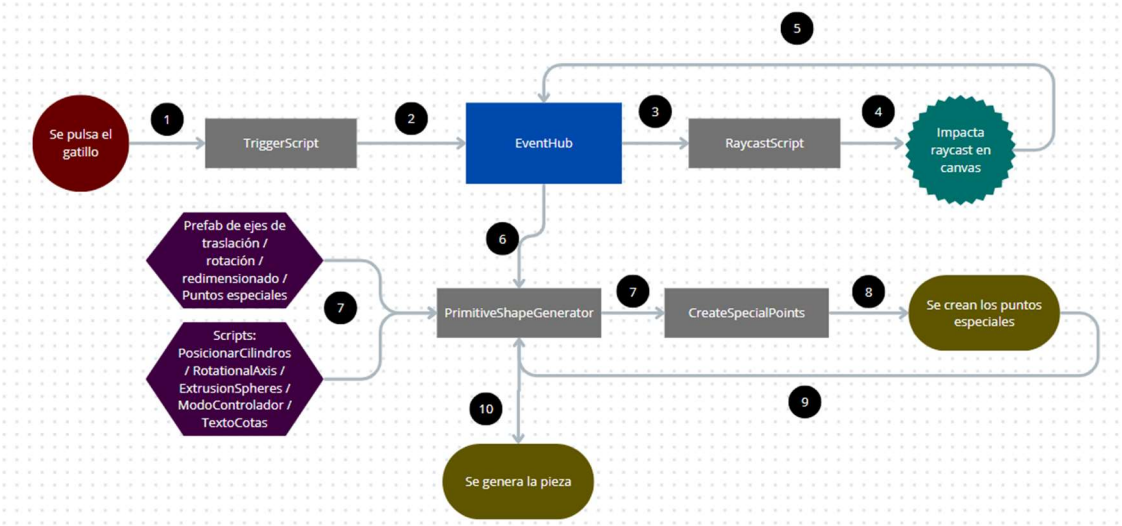


Ilustración 17: Estructura de la función creación de las formas primitivas

4.8.1 Canvas

En primer lugar, voy a detallar cómo detecta el canvas que ha sido impactado por el raycast. Para crear el canvas con la imagen en el escenario, y que sea un objeto detectable por el raycast, hay que hacer lo siguiente:

Primero, crear el canvas: click derecho en la jerarquía → UI → Canvas. A este canvas hay que añadirle un "collider 3D" para que pueda interactuar con el raycast. Un collider es una zona que el usuario puede delimitar, y cuando el raycast impacta con dicha zona, envía la información del objeto que tienen el collider. En este caso es el canvas. Posteriormente le añadí un como hijo una imagen de la figura que corresponda: click derecho en la jerarquía → UI → Image. Con esto ya está el canvas creado en el escenario y se puede pasar al código.

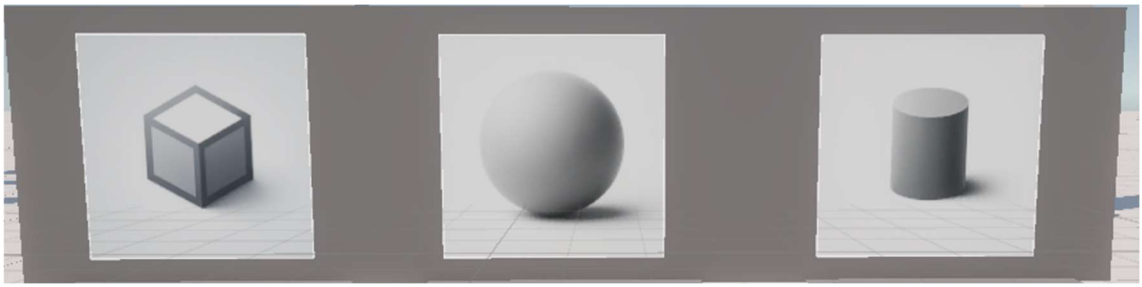


Ilustración 18: Imagen de los 3 canvas para crear las formas primitivas

En el script "PrimitiveShapeGenerator", habilité unos eventos que estén escuchando al impacto del raycast en los canvas. Si impacta, devuelve como argumento la información del objeto canvas y activa la función "`void HandleCanvasHit(RaycastHit hit)`".

```
    void HandleCanvasHit(RaycastHit hit)
    {
        // Obtiene el nombre del objeto golpeado
        string hitObjectName = hit.collider.gameObject.name;

        // Decide qué tipo de figura primitiva crear basado en el nombre del objeto
        if (hitObjectName == "Cube_icon" && gameObject.name == "Cube_icon")
        {
            CreatePrimitiveShape(PrimitiveType.Cube);
        }
         else if (hitObjectName == "Sphere_icon" && gameObject.name == "Sphere_icon")
        {
            CreatePrimitiveShape(PrimitiveType.Sphere);
        }
         else if (hitObjectName == "Cylinder_icon" && gameObject.name == "Cylinder_icon")
        {
            CreatePrimitiveShape(PrimitiveType.Cylinder);
        }
    }
```

Como se puede observar en la función, primero obtiene el nombre del canvas que ha impactado con el raycast, y en función del nombre, pasa distintos argumentos a la función "`void CreatePrimitiveShape(PrimitiveType type)`". Aclaro que cada canvas tiene el nombre de la figura que representa.

Además, voy a explicar el código de "RaycastScript" y "EventHub" para manejar el evento y activar la función. En "RaycastScript" busco si el raycast no ha impactado en ningún objeto prioritario. Si ese es el caso, busco si el objeto que ha impactado, tiene como componente el script "PrimitiveShapeGenerator". En caso afirmativo, disparo el evento y le paso la información del objeto canvas. Quedaría de la siguiente forma:

```
PrimitiveShapeGenerator hitShapeGenerator = hit.collider.GetComponent<PrimitiveShapeGenerator>();


else if (hitShapeGenerator != null)
{
        EventHub.ShapeGeneratorHit(hit);
        break;
}
```

El "EventHub" actúa como intermediario para disparar el evento. En este caso queda como:

```
public delegate void RaycastEvent(RaycastHit hitInfo); //Se define el delegado
public static event RaycastEvent OnShapeGeneratorHit;


public static void ShapeGeneratorHit(RaycastHit hitInfo) => OnShapeGeneratorHit?.Invoke(hitInfo); //Dispara el método en "PrimitiveShapeGenerator"
```

4.8.2 Formas Primitivas

En esta sección se va a explicar cómo funciona la parte del código que crear las formas primitivas, se corresponde con la función "`void CreatePrimitiveShape(PrimitiveType type)`", del script "PrimitiveShapeGenerator".

A esta función le llega como argumento el tipo de figura que debe crear: cubo, esfera o cilindro. La parte clave del código que crea la figura primitiva es:

```
GameObject shape = GameObject.CreatePrimitive(type);
```

Posteriormente le añado un material al objeto. El material es estándar, pero modificado para poder cambiarle la transparencia cuando la pieza es activada 4.13.

También añado los scripts que se encargan de generar los ejes de traslación 4.9, rotación 4.10 y dimensionado 4.11.

Estos ejes se corresponden con figuras prefabricadas. Son cilindros para los ejes de traslación, toroides para los ejes de rotación y óvalos para los ejes de dimensionado. Como estos elementos ya están prefabricados, se los debo incluir a los scripts que los crean:

```
public GameObject toroidePrefab;        // Ejes de rotación         public
GameObject grabAxisXPrefab;      // Ejes de traslación

public GameObject extrusionSphere;      // Ejes de dimensionado
```

Estos objetos prefabricados se deben añadir desde el Inspector. El código, por ejemplo, con los ejes de traslación queda así:

```
//Añade el script para crear los ejes de traslación

var posicionadorCilindros = shape.AddComponent<PosicionarCilindros>();

// Asigna el prefab de eje de traslación

posicionadorCilindros.cilindroPrefab = grabAxisXPrefab;
```

La lógica que explica el funcionamiento de cómo se crean y se destruyen los ejes se explicará en 4.12.

También debo añadir otros scripts de vital importancia, como: “ModoControlador”, que se encarga de alternar los modos entre mover, rotar y dimensionar; y “TextoCotas”, que se ocupa de añadir u ocultar las cotas en la pieza. Para poder luego identificar las piezas, les añado el tag de "PrimaryObject".

Para solucionar un problema de distorsión de los ejes al dimensionar la pieza, se le debe añadir un objeto vacío llamado “emptyScale”, que sea hijo de la pieza, pero padre de los ejes. El problema está explicado en 4.11.3.

Finalmente, añado los últimos elementos de las piezas. Esta parte final difiere según sea cubo, cilindro o esfera, por lo que se divide en bucles if. Primero las posiciono en la zona del escenario designada para ellas. Posteriormente elimino los collider que vienen por defecto y creo unos nuevos que se adaptan mejor a su forma. Algunos collider que vienen por defecto no se ajustan bien al volumen de la pieza, por lo que luego interactúan mal con el raycast. Para solucionarlo he añadido un mesh collider, que es un poco más complejo, pero se ajusta perfectamente a la superficie.

Finalmente, he creado también los puntos significativos de cada pieza. Estos puntos se usarán posteriormente para la función de movimiento por puntos 4.16.

4.9 TRASLACIÓN DE LAS PIEZAS

Una de las funcionalidades más básicas que hay que implementar en un programa de diseño industrial es la posibilidad de mover las piezas en función de los ejes. En este proyecto me he centrado en el movimiento de las piezas en sus ejes locales. La figura tiene 3 ejes, que son cilindros que corresponden con los ejes cardinales.

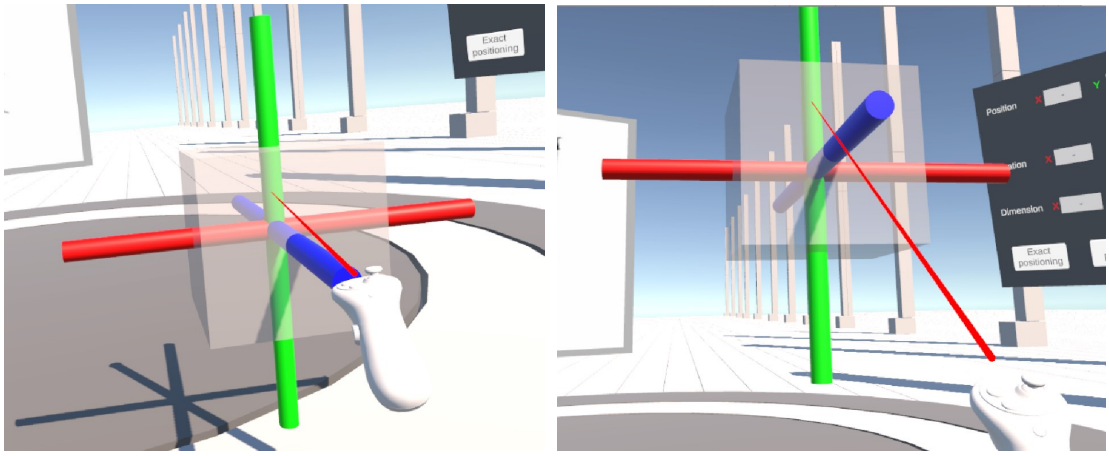


Ilustración 19: Ejemplo de traslación de la pieza en el eje Y local

El usuario dispara con el raycast en el cilindro que apunta en la dirección que quiere mover la pieza. Mientras mantiene pulsado y mueve el controlador, la pieza se moverá siguiendo el movimiento del usuario, solo en la dirección del eje seleccionado. Para poder hacer ello, primero hay que crear los ejes (cilindros) dentro de la pieza.

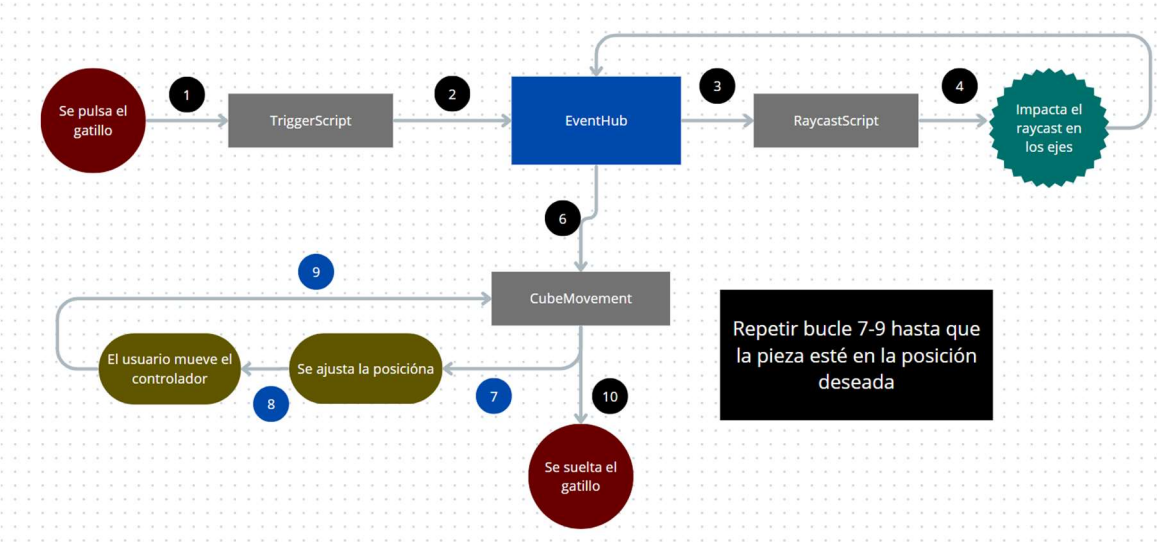


Ilustración 20: Diagrama a alto nivel de la función traslación de piezas

4.9.1 Creación de ejes de traslación

El script que se encarga de la creación de los ejes de traslación es "PosicionarCilindros" (10.5). Como ya hemos visto en 4.8, cuando la forma primitiva es creada, se le añade también el prefab del eje. Por lo que en el argumento:

```
public GameObject cilindroPrefab;
```

"cilindroPrefab" siempre va a tener asignado el prefab del eje.

La función para crear los cilindros es llamada desde el script "ModoControlador" 4.12. Cuando es llamada, se destruyen todos los ejes de traslación que pudiese haber, y se crean nuevos. No debería de haber ningún eje de traslación creado cuando esta función es llamada, pero se hace por seguridad y para evitar posibles errores al expandir el código.

Para crear los ejes se llama a la función:

```
void CrearCilindroEnEje(Vector3 rotacionEjeLocal, Color color, Transform padre)
```

Esta función es llamada 3 veces, para crear los 3 ejes. El eje X es de color verde, el eje Y de color rojo y el eje Z de color azul. De esta manera se mantiene coherencia con el criterio de colores que sigue Unity.

Estos ejes se asignan como nietos de la pieza, e hijos del "emptyScale": Pieza → emptyScale → Ejes. Ello se hace por solucionar un problema en el dimensionado 4.11.3.

Para crear los ejes se utiliza la siguiente línea de código:

```
GameObject cilindro = Instantiate(cilindroPrefab, padre.position, Quaternion.identity, padre);
```

Posteriormente, se orienta cada eje en su lugar correspondiente, relativamente a la posición del padre.

Por último, se escala cada eje en función del tamaño de la pieza. Esto se hace para no tener ejes muy grandes al trabajar en una escala pequeña o viceversa. Para hacer esto, divido la escala original del eje entre un factor de escala. Este factor de escala es calculado haciendo el promedio de la escala en X, Y, y Z de la pieza original:

```
float promedio = (childScale.x + childScale.y + childScale.z) / 3;

adjustedScale.x = baseScale.x * 1 / (promedio - 0.2f * promedio);
```

Todos los ejes creados son añadidos en una lista. La función de destruir los ejes busca en esa lista y busca todos los objetos que haya en ella:

```
    public void DestruirCilindros()
    {
        foreach (GameObject cilindro in cilindrosCreados)
        {
            Destroy(cilindro);
        }
        cilindrosCreados.Clear();
    }
```

4.9.2 Movimiento de Traslación

El script encargado del movimiento de traslación es “CubeMovement” (10.6). Este script se añade a los ejes de traslación. Cuando el raycast impacta en uno de esos ejes, mientras el usuario siga manteniendo pulsado el trigger, se seguirá ejecutando la función de mover. No importa que el raycast deje de impactar en el eje, la función se seguirá ejecutando.

Esta funcionalidad puede parecer sencilla, pero resulta bastante compleja si se quiere conseguir un movimiento fluido e intuitivo. El movimiento que se quiere lograr es que la pieza se mueva

sólo en la dirección en la que está apuntando el eje. Hay que recalcar que la pieza puede estar rotada, por lo que los ejes locales no coinciden con los globales. El movimiento que se le quiere dar siempre es en función de los ejes locales de la pieza.

Para conseguirlo, hay que seguir un procedimiento geométrico y matemático e incorporarlo en el código. Es un razonamiento complejo de explicar y de visualizar. Vamos primero a suponer que los ejes locales de la pieza coinciden con los ejes globales y que queremos mover la pieza en el eje X. Cuando el raycast impacta en el eje de la dirección x, se inicializa un plano definido por un vector normal en la dirección Z. Dicho de otra manera, el plano contiene los ejes X e Y.

```
Vector3 upLocal = transform.up;

//Inicialización del plano definido por el vector normal al vector Y global
y al vector X local del eje que se esté moviendo

Vector3 normalVector = Vector3.Cross(Vector3.up, upLocal).normalized;

plane = new Plane(normalVector, transform.position);
```

Una vez creado el plano, se calcula en todo momento en qué punto del espacio está impactando el raycast con dicho plano. Esta intersección del raycast con el plano nos da un punto. Si movemos el controlador, ese punto cambiará. El propósito de esta función es detectar cuánto se ha movido el punto en ese plano sólo en la dirección X. Luego se modifica la posición de la pieza en la misma medida. De esta forma, la pieza estará "siguiendo" al raycast.

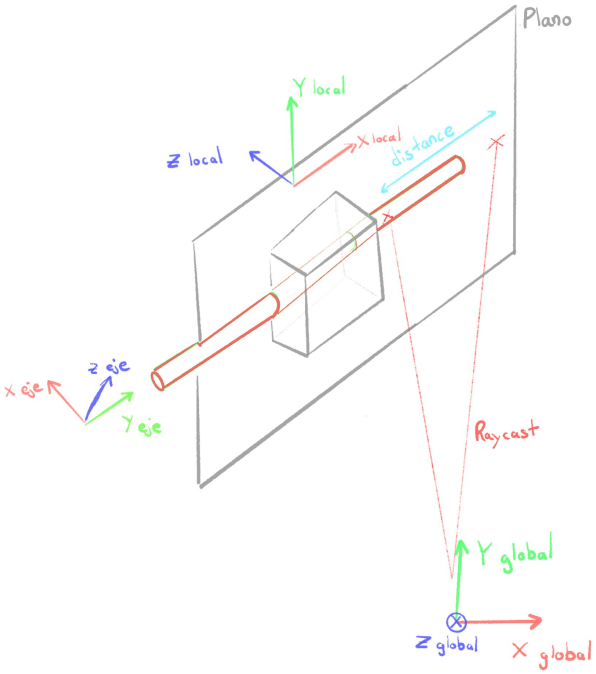


Ilustración 21: Ilustración gráfica del funcionamiento de la lógica de traslación

Poniendo un ejemplo, imaginemos que inicialmente el raycast impacta en el eje X en el punto de intersección con el plano (1, 0.5, 3). Posteriormente movemos el raycast y el nuevo punto de impacto es (2, 3.5, 9). El movimiento que se le aplicará a la pieza es (1, 0, 0), ya que solo nos interesa el movimiento en uno de los ejes.

Hasta ahora se ha explicado el movimiento, suponiendo que la pieza no está rotada, y que los ejes locales de la pieza coinciden con los globales. Ahora hay que expandir el razonamiento para tener en cuenta el caso de que la pieza esté rotada. Poniendo el mismo ejemplo de que queramos mover el eje X local de la pieza. Ese eje X ya no coincidirá con el eje X global. El plano se inicializa de la misma manera, pero ahora contiene el eje X local y el eje Y local, siendo normal al eje Z local. Se puede seguir calculando la intersección entre el raycast y el plano. Lo diferente es que ahora ya no estamos interesados en lo que se mueva el raycast en el eje X global, sino en el local. Para ello, hay que hacer una proyección, que se representa con el producto escalar.

El producto escalar se hace entre el vector que conecta los dos puntos de intersección del raycast con el plano, y con la dirección X local (hacia donde esté apuntando el eje). Con esto calculamos la distancia que hay que mover la pieza en la dirección deseada.

El último obstáculo que hay que superar es que cuando el raycast impacta con el eje, no tiene por qué impactar en el centro del eje necesariamente. Al calcular la nueva posición de la pieza, hay que tener en cuenta la distancia inicial entre el punto donde ha impactado el raycast con el plano y el centro de la pieza.

La parte del código que calcula el movimiento que se tiene que mover la pieza y la intersección entre el raycast y el plano es la siguiente:

```
// Usa raycastOrigin.position y raycastOrigin.forward para definir la dirección del rayo
Vector3 rayDirection = raycastOrigin.forward;


// Inicia el rayo en la posición del origen del rayo hacia la dirección que este apunta
Ray ray = new Ray(raycastOrigin.position, rayDirection);


// Verifica si este rayo intersecta con el plano definido en InitializePlane.
if (plane.Raycast(ray, out float enter))
{
    // Calcula el punto exacto de intersección con el plano.
```

```
    Vector3 intersectionPoint = ray.GetPoint(enter);


    // Calcula la diferencia de posición desde la posición actual del objeto
hasta el punto de intersección.
    Vector3 positionDifference = intersectionPoint - transform.position;


    // Proyecta esta diferencia en el eje up local del objeto para determinar
cuánto mover el objeto.
    float upMovementAmount = Vector3.Dot(positionDifference, transform.up);


    // Mueve el objeto en su eje up local por la cantidad calculada.
    if (transform.parent.parent != null)
    {
        float compensation = upMovementAmount - initialUpDifference;
        transform.parent.parent.position += transform.up * compensation;
    }
```

Esta función se está ejecutando constantemente hasta que el usuario suelte el gatillo, así que el movimiento que se consigue es muy fluido e intuitivo. Cuando el usuario suelta el gatillo, se dispara un evento que hace que el atributo “private bool isMoving” sea = false. De este modo, no vuelve a ejecutar la función hasta que el raycast vuelve a impactar en un eje de traslación.

4.10 ROTACIÓN DE LAS PIEZAS

De manera similar a la traslación de las piezas, se han creado unos ejes que permiten a los usuarios rotar las piezas en sus ejes locales. Para ello, el raycast debe de impactar en uno de los ejes. A medida que el usuario mueva el controlador, la pieza rotará alrededor del eje seleccionado. En la Ilustración 22 se puede observar un simple diagrama de flujo de cómo está implementada esta función.

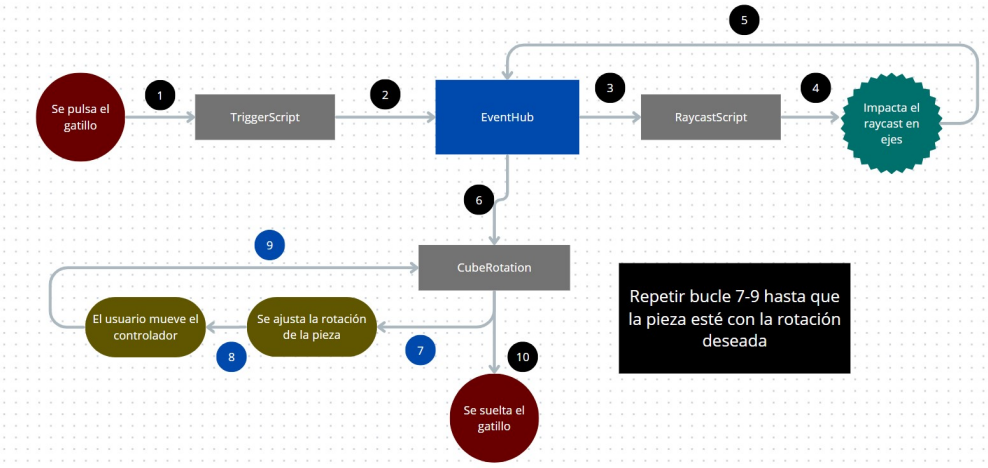


Ilustración 22: Estructura a alto nivel de la función rotación de las piezas

4.10.1 Creación de ejes de rotación

Para crear los ejes de rotación, seguí un procedimiento similar a los ejes de traslación 4.9. Esta vez, los ejes ya no son cilindros, sino toroides. El script que se encarga de la creación de los ejes de rotación se llama "RotationalAxis" (10.7). Hay un toroide que rodea la pieza para cada eje, habiendo un total de 3 toroides (Ilustración 23). El prefab de los toroides se asigna cuando se crea la pieza, en el script de "PrimitiveShapeGenerator" (4.8).

Del mismo modo que con los ejes de traslación, los de rotación se crean cuando le llega la orden de "ModoControlador". La función principal encargada de crear y posicionar los toroides es:

```
void CreateAndPositionToroid(GameObject prefab, Quaternion rotacion, float scale, Color color)
```

Un ejemplo de su uso, por ejemplo, con el eje Y sería:

```
// Calcula el tamaño del objeto padre para posicionar los toroides correctamente

Vector3 parentSize = CalculateParentSize();

float largestDimension = Mathf.Max(parentSize.x, parentSize.y, parentSize.z);

// Crea y posiciona los toroides con colores según los ejes

CreateAndPositionToroid(toroidePrefab, Quaternion.Euler(0, 0, 90), largestDimension, Color.blue); // Eje Z - Verde
```

"largestDimension" se usa para ajustar la escala del toroide en función del tamaño del padre (la pieza). El tamaño del padre "parentSize" se calcula calculando los bordes de su "mesh" o malla.

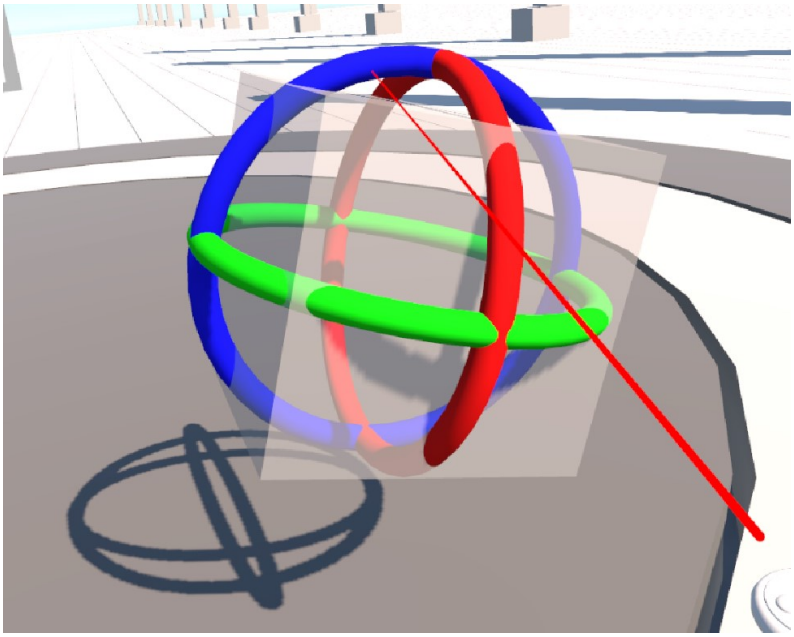


Ilustración 23: Usuario rotando la pieza usando los ejes de rotación

Posteriormente, se ajusta la escala de los ejes en función del tamaño de la pieza:

```
toroide.transform.localScale = new Vector3(scale, scale, scale) * 1f; //
Ajusta este factor según sea necesario
```

El resto del código del script para crear los ejes de rotación es análogo al de los ejes de traslación. También tiene como atributo una lista que incluye todos los ejes creados, para luego poder destruirlos todos cuando se llama a la función:

```
public void DestruirToroides()
```

4.10.2 Movimiento de rotación

El script encargado de rotar las piezas, cuando el raycast impacta en los ejes de rotación es "CubeRotation" (10.8). Hay que añadir este script a los ejes de rotación en el Inspector de Unity (al crear el prefab de los toroides ya se lo añadí).

El razonamiento que he seguido para rotar la pieza es similar al usado para trasladar la pieza 4.9. Para simplificar la explicación, vamos a llamar eje de rotación X al eje que es normal al plano que contiene al toroide. En otras palabras, es el eje que atraviesa el agujero del toroide.

La rotación de la pieza es alrededor de los ejes locales de la misma. Esto quiere decir que los ejes de la pieza no tienen por qué coincidir con los globales. La función de rotar el eje de rotación, junto a la pieza empieza cuando el raycast dispara en uno de los toroides. En ese momento, se dispara un evento que hace que se llame al método "HandleRaycastHit", que se encarga de la lógica de rotación de la pieza.

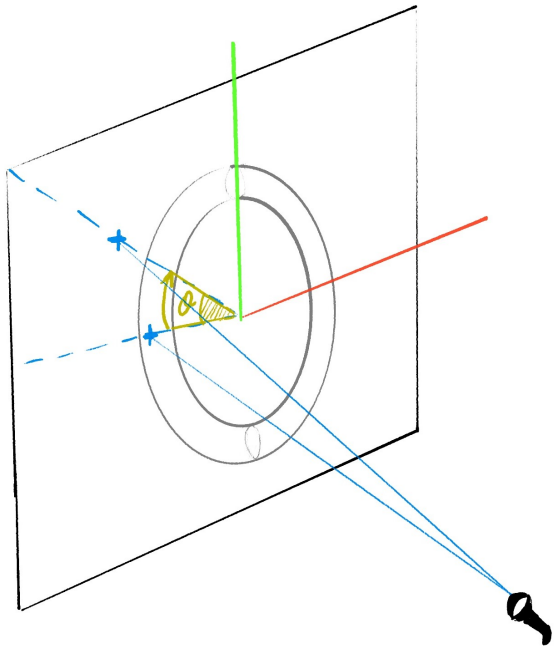


Ilustración 24: Ilustración gráfica del funcionamiento de la lógica de rotación

Cuando el raycast impacta, se inicializa un plano que contiene al eje de rotación que ha sido impactado. En el código se almacena el punto del espacio resultado de la intersección del raycast con dicho plano. Cuando el usuario mueve el raycast, el programa guarda la posición del segundo impacto del raycast con el plano. Si trazamos dos rectas, cada una entre el centro del toroide y los puntos de impacto, podemos calcular el ángulo entre dichas rectas. Ese ángulo es la cantidad que queremos rotar el eje de rotación, y por lo tanto, la pieza. Para ayuda visual, referir a la Ilustración 24: Ilustración gráfica del funcionamiento de la lógica de rotación. Los puntos de impacto se representan con la cruz azul, y el ángulo de giro en amarillo.

Mientras el usuario mantenga el gatillo pulsado, la función se está ejecutando constantemente, por lo que un pequeño movimiento del controlador del usuario, va a hacer que la pieza se ajuste a ese nuevo ángulo. Esto hace que el proceso sea intuitivo y fluido.

Los puntos de intersección entre el raycast y el plano se pasan a ejes locales. De esta manera, el vector de la recta que une el centro del toroide y el punto de intersección es el punto en sí. Me explico, la ubicación de la pieza en sus ejes locales es (0,0,0). El vector es el resultado del punto de intersección menos (0,0,0), por lo que es igual al punto de intersección. Para pasar un punto del espacio global al local se utiliza la función de Unity “InverseTransformPoint(Vector3 point)”.

Con todo esto, la parte importante del script “CubeRotation”, encargada de la rotación de la pieza es la siguiente:

```
// Calcula el punto exacto de intersección con el plano.
```

```
Vector3 intersectionPoint = ray.GetPoint(enter);


// Convertimos los puntos de intersección a coordenadas locales del toroide para calcular la diferencia de ángulo.
Vector3 lastLocalPoint = transform.InverseTransformPoint(lastHitPoint);
Vector3 currentLocalPoint = transform.InverseTransformPoint(intersectionPoint);


// Calculamos el ángulo entre los dos puntos en el espacio local del toroide.
//SignedAngle calcula el ángulo en grados entre dos vectores, alrededor de un eje específico
float angle = Vector3.SignedAngle(lastLocalPoint, currentLocalPoint, Vector3.forward);


// Rotamos el toroide alrededor del eje normal al plano creado.
transform.parent.parent.Rotate(plane.normal, angle, Space.World);


// Actualizamos el último punto de intersección.
lastHitPoint = intersectionPoint;
```

4.11 REDIMENSIONADO DE LAS PIEZAS

Otra opción de vital importancia para diseñar piezas es poder cambiar sus dimensiones. Por ello, añadí la posibilidad de que los usuarios puedan redimensionar las piezas de manera rápida e intuitiva. Cuando se activa el modo “dimensionado”, se crean unas esferas u óvalos en ciertos puntos de la pieza.

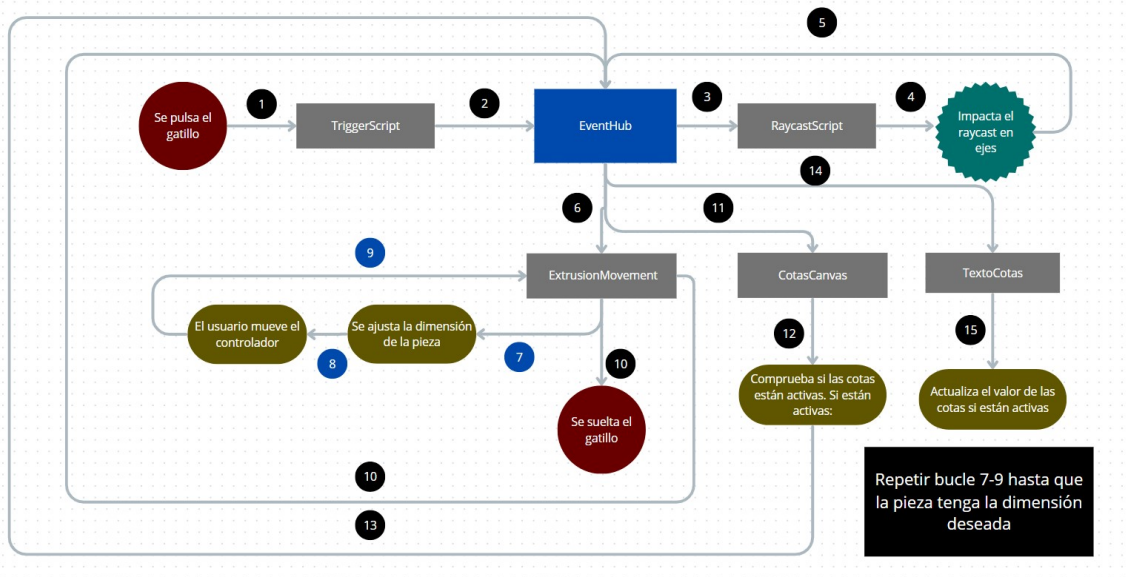


Ilustración 25: Estructura a alto nivel de la función redimensionado de las piezas

Cuando el usuario interacciona con el raycast con esos objetos, puede modificar las dimensiones de la pieza en diferentes ejes. Se puede modificar las dimensiones en X, Y, y Z locales de las piezas moviendo los óvalos que apuntan en dicha dimensión. Adicionalmente, hay se crea una esfera en una esquina de la pieza, que al moverla se modifica la escala general de la pieza. Como herramienta adicional, con los cilindros se tiene la opción de modificar directamente su radio.

4.11.1 Creación de ejes de redimensionado

El script “ExtrusionSpheres” es el encargado de la creación de las esferas de extrusión en la pieza. Posteriormente, moviendo estas esferas, se podrá cambiar las dimensiones de la pieza.

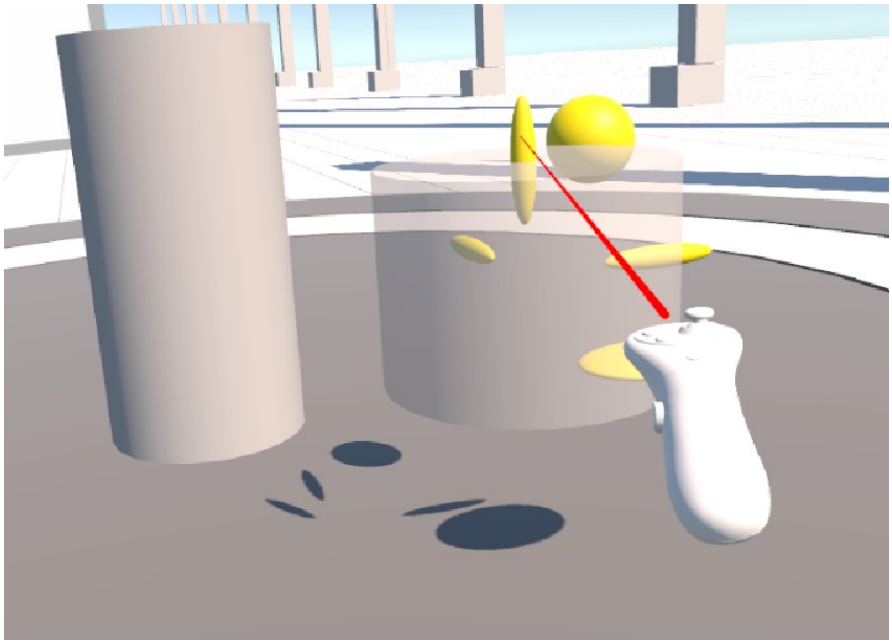


Ilustración 26: Usuario modificando la altura de un cilindro mediante los ejes de redimensionado

El prefab de las esferas es añadido cuando se crea la pieza en "PrimitiveShapeGenerator" 4.8, por lo que no hay que volver a preocuparse. Todas las piezas tienen 4 esferas de extrusión; una para cambiar las dimensiones en cada eje local y una cuarta para cambiar la escala de la pieza. Los cilindros tienen una adicional para poder cambiar el radio. Es por eso que hay que hacer una distinción en el código para saber si la pieza es un cilindro o no. El collider del cilindro es un "mesh collider", así que esto se puede arreglar añadiendo la siguiente línea de código:

```
if (collider is MeshCollider meshCollider)

{

    //Especifico la orientación y la escala de la esfera para cambiar el
radio

    Quaternion specialRotationR = Quaternion.Euler(0, 180, 90);

    Vector3 specialScaleR = new Vector3(0.1f, 0.7f, 0.5f);

    InstantiateSphere(new Vector3(colliderSize.x / 2, -colliderSize.y/2 ,
0), specialScaleR, specialRotationR, "Extr_radius");

}
```

Instancio el resto de esferas de manera similar, cambiando para cada una su orientación, escala y el tag para identificarlas. Esto es para que se orienten en los lugares adecuados alrededor de la pieza cuando se creen.

La función "InstantiateSphere" es la encargada de crear y posicionar dichas esferas:

```
void InstantiateSphere(Vector3 localPosition, Vector3 baseScale, Quaternion specificRotation, string tag)
```

Hay que tener en cuenta que la pieza puede estar rotada cuando se le instancian las esferas de redimensionado, por lo que hay que tener en cuenta esa rotación:

```
// Calcula la posición global basada en la posición local deseada y la transformación del padre
Vector3 globalPosition = transform.TransformPoint(localPosition);


// Combina la rotación del padre con la rotación específica deseada
Quaternion combinedRotation = transform.rotation * specificRotation;
```

La línea de codigo que crea las esferas en el lugar deseado es:

```
// Instancia la esfera con la posición global y la rotación combinada
GameObject sphereInstance = Instantiate(sphere, globalPosition, combinedRotation, transform.GetChild(0));
```

Por último, se ajusta la escala de las esferas, y se le añade el tag correspondiente. También hay que cambiarle el "collider" a "mesh collider". Eso es para que se ajuste mejor a su forma, y luego no haya problemas al interaccionar con el raycast. Cabe destacar que también se están almacenando todas las esferas en una lista de objetos cada vez que son creadas. Cuando se llama al método de "DestruirEsferas()", se destruyen todas las que haya en la lista.

La jerarquía de las esferas de dimensionado es la misma que la de los ejes de rotación y de extrusión. La pieza es el padre del "emptyScale", que es a su vez padre de las esferas de dimensionado y del resto de ejes. Pieza→ emptyScale →esferas de dimensionado. Se explica la razón del uso del objeto "emptyScale" en 4.11.3.

4.11.2 Redimensionado

El script que se encarga de la mecánica para redimensionar las piezas es "ExtrusionMovement" (10.10). Para hacer esta función he reutilizado parte del código usado para trasladar las piezas, en el script de "CubeMovement" 4.9. La diferencia principal es que, en lugar de mover el objeto, ajusto su escala en la dirección que corresponda. El principio del código funciona de la misma manera, por lo me voy a repetir para no alargarlo demasiado. El programa empieza cuando el raycast dispara e impacta en una de esas esferas, en ese momento se inicializa un plano que contiene la esfera. Se está midiendo constantemente si el raycast impacta en puntos distintos del plano.

La diferencia fundamental es que en este caso no se mueve la pieza para seguir al raycast, sino que cambia su dimensión en ese eje. Por lo tanto, la pieza cambia su tamaño. Queremos que las

esferas de dimensionado estén siempre a la misma posición relativa de la pieza, por lo que al cambiar el tamaño de la pieza en una dirección, hay que volver a colocar las esferas hasta que vuelan a estar en la misma posición relativa respecto a la pieza.

Pongamos el ejemplo de un cubo que tiene una esfera en una de las caras del cubo. Supongamos que esa cara es perpendicular al eje x del cubo. Cuando cambiamos la escala del cubo en su dirección x, la esfera ya no va a estar es una de sus caras, ya que la posición de esa cara es distinta. Es por ello que también hay que ajustar la posición de las esferas de extrusión. La solución que implemento para ajustar esa posición es calcular las dimensiones de la pieza una vez redimensionada en el método "CheckCollider()" y las vuelvo a orientar en su posición relativa a la pieza. Esto se hace en la última parte del método "RestoreChildrensScales(Vector3 colliderSize)", por ejemplo, para el eje Z:

```
if (child.tag == "Extr_z")
{
      child.localPosition = new Vector3(0, 0, colliderSize.z / 2 * newScale.z);
}
```

La parte del código que se encarga de recalcular la escala de la pieza es el método:

```
void AdjustParentScale(float upMovementAmount)
```

De igual manera que en el script de "CubeMovement", se calcula la distancia que se ha movido el raycast por el plano:

```
float compensation = upMovementAmount - initialUpDifference;
```

Utilizando ese factor, ya se puede saber cuánto hay que escalar la pieza. Cabe recalcar que sólo hay que extruir la pieza en la dirección deseada. De tal manera que si el raycast ha impactado en la esfera de redimensionado del eje x, solo se va a extruir la pieza en el eje x local.

He orientado las esferas de redimensionado de tal manera que el eje Y de las esferas de redimensionado siempre están apuntando hacia el eje en el que se quiere extruir la pieza. De tal manera que los ejes Y locales de la esfera y de la pieza solo coinciden en la esfera de extrusión del eje Y. En la esfera de extrusión del eje X, el eje Y local de la esfera coincide con el eje X local de la pieza. Esto puede parecer innecesario, pero tiene una buena razón. De esta manera, el cálculo del valor "compensation" (lo que se mueve el raycast por el plano) es el mismo para todas las esferas, y no hay que especificar de qué esfera se trata.

Donde sí hay que diferenciar la esfera que ha impactado con el raycast es a la hora de redimensionar la pieza. Esto se hace comprobando el tag de cada esfera. Este es el ejemplo de código de redimensionado en el eje X:

```
// Si el raycast ha impactado con la esfera de extrusión del eje X
else if (gameObject.CompareTag("Extr_x"))
{
    // El eje "up" del hijo se alinea más con el eje X del padre
    float newXScale = transform.parent.parent.localScale.x + compensation;
    newXScale = Mathf.Max(newXScale, 0.05f); // Esto evita que la escala no sea demasiado pequeña
        transform.parent.parent.localScale = new Vector3(newXScale, transform.parent.parent.localScale.y,
transform.parent.parent.localScale.z); // Ajustar según el eje relevante
}
```

Sin embargo, al hacer esto se produce un problema que se ha emitido hasta el momento. Las esferas de extrusión se añaden como “hijas” de la pieza. Esto es necesario, ya que cuando se mueve la pieza, las esferas se mueven de la misma manera. El problema es que cuando se escala la pieza en un eje, todos los hijos también se escalan en el mismo eje, dando lugar a esferas deformadas de manera indeseada.

4.11.3 Problema del escalado indeseado de los hijos

El problema parece que tiene una fácil solución inicial: devolver a todos los hijos el valor de su escala original. Esto no funciona, ya que el valor de la escala de los hijos no se cambia en sus propiedades, solo se cambia visualmente.

La siguiente alternativa puede ser calcular lo que ha incrementado la escala del padre para así revertirle ese cambio de escala a los hijos (con la inversa). El nuevo problema es que no todos los hijos tienen el mismo sistema de referencia de ejes que el padre. Esto provoca que, por ejemplo, cuando se modifica la escala en X del padre, para algunos hijos se habrá modificado en Y, para otros la Z y para otros la X. Esto hace que intentar modificar la escala de cada hijo individualmente sea muy complejo, ya que además hay que tener en cuenta un número muy elevado de combinaciones.

La solución definitiva que soluciona todos estos problemas es añadiendo un objeto vacío que sea hijo de la pieza y padre de las esferas de extrusión, se llama “emptyScale”. Ahora la jerarquía es:

Abuelo (pieza) —> padre (objeto vacío) —> hijos (ejes de traslación, rotación y esferas de dimensionado).

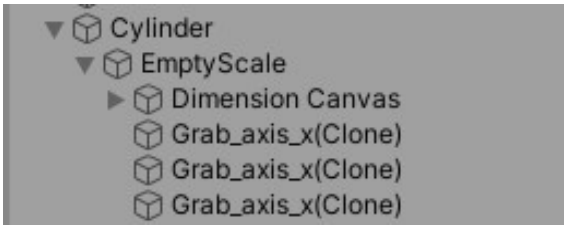


Ilustración 27: Jerarquía de los hijos de las piezas

Este objeto solo tiene el componente "Transform", que da información de posición y escalado, por lo que es invisible. Se coloca este objeto de tal manera que sus ejes de referencia coincidan con los de la pieza. Cuando se modifique la escala del abuelo, solo se tiene que invertir ese cambio de escala en el objeto vacío. Los hijos automáticamente experimentarán el mismo cambio de escala que el padre. Por otro lado, los nuevos hijos que se creen ya tendrán aplicado el cambio de escala correcto.

El "emptyScale" ya se añade a las piezas cuando se crean en "PrimitiveShapeGenerator" 4.8. En el script de "ExtrusionMovement" solo se tiene que recalcular su escala.

```
Vector3 newScale = transform.parent.parent.localScale;

Vector3 scaleFactor = new Vector3(oldScale.x / newScale.x, oldScale.y / newScale.y, oldScale.z / newScale.z);


foreach (Transform child in transform.parent.parent)

{

    // Aplica la escala a todos los hijos, que son los emptyScale y los puntos significativos

     child.localScale = new Vector3(child.localScale.x * scaleFactor.x, child.localScale.y * scaleFactor.y, child.localScale.z * scaleFactor.z);

}
```

4.12 MODOS DE EDICIÓN

En el programa se le da la opción al usuario de alternar entre mover, rotar y redimensionar las piezas. Cuando el raycast impacta en una pieza, ésta se "activa". Una vez activa, se generan los ejes del modo en el que se encuentre el programa en ese momento. Si el usuario pulsa el botón A (4.5) del controlador derecho, esos ejes se van a cambiar. El programa dispone de 3 modos, ordenados como 1º, ejes de traslación, 2º, de rotación y 3º de extrusión. Cuando se encuentra en el 3º modo y se pulsa el botón A, se vuelve al 1º.

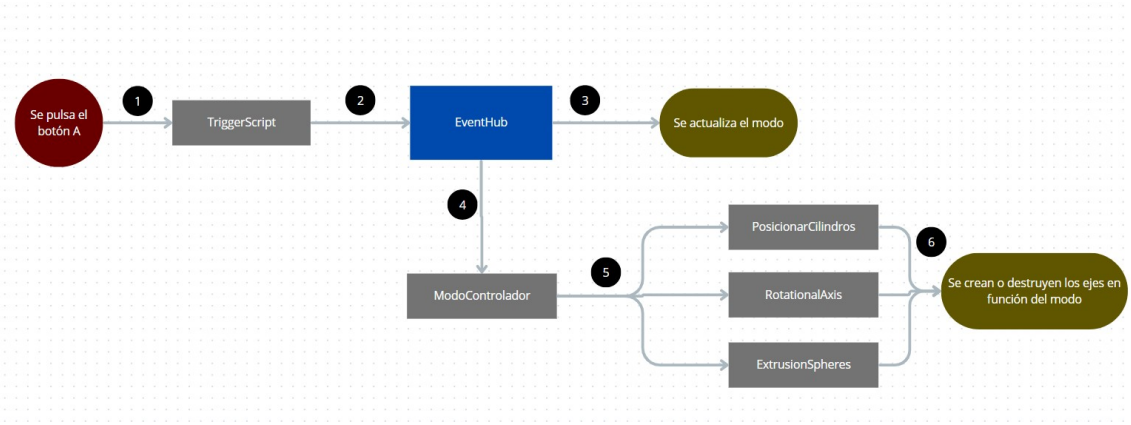
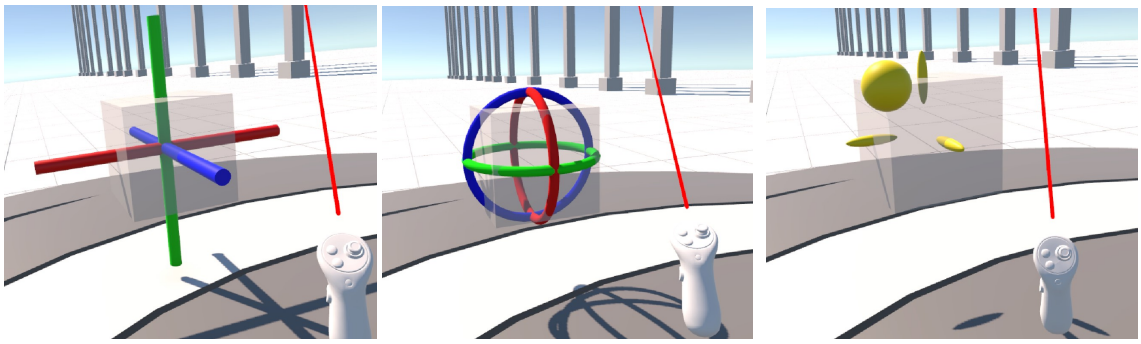


Ilustración 28: Estructura a alto nivel de los modos de edición

Toda la lógica encargada de la activación, desactivación y cambio de modo de las piezas se encuentra en el script "ModoControlador" (10.11). Este script es añadido a las formas primitivas cuando son creadas 4.8.



A) *B)* *C)*

Ilustración 29: Los 3 modos de edición. A) Traslación. B) Rotación. C) Redimensionado

Además, "ModoControlador" se encarga de dar la orden de destruir los ejes existentes y de la creación de nuevos ejes cada vez que se cambia de modo. Es por ello que tiene que tener acceso a los métodos de creación y destrucción de ejes. Estos métodos se encuentran en los scripts de "PosicionarCilindros", "RotationalAxis" y "ExtrusionSpheres". Estos scripts son asignados al inicio del programa, en "Awake":

```
private PosicionarCilindros posicionadorCilindros;

private RotationalAxis rotationalAxis;

private ExtrusionSpheres extrusionSpheres;


private void Awake()
```

```
{
    posicionadorCilindros = GetComponent<PosicionarCilindros>();
    rotationalAxis = GetComponent<RotationalAxis>();
    extrusionSpheres = GetComponent<ExtrusionSpheres>();
}
```

Cuando el usuario pulsa el botón A, se dispara un evento que ejecuta el método de "AlternarModo()":

```
private void AlternarModo()
{
    if (activated)
    {
        DestruirObjetosAnteriores();
        CrearObjetos();
    }
}
```

Este método se ejecuta en todas las piezas que estén siendo diseñadas en el escenario. Cambia el modo sólo de aquellas que se encuentren en el estado "activated". Para cambiar de modo, primero llama a un método que elimina los ejes del modo en el que se encuentre la pieza, para así luego crear los ejes del nuevo modo.

Para tener un registro de en qué modo se encuentra el programa actualmente, el modo se guarda en el "EventHub". La principal razón es que "EventHub" es la única clase estática (no hay instancias de la clase). En cambio, si lo guardase en el "ModoControlador", lo estaría guardando solo para ciertas instancias, y no para las instancias que todavía no existen, ya que es una clase dinámica. Se recuerda que se crea una instancia cada vez que se genera una forma primitiva.

Cada vez que se pulsa el botón A, se actualiza el estado del modo en el "EventHub". Para que el programa sepa qué debe destruir y cuales crear en cada cambio de modo, "Modocontrolador" accede al argumento "currentMode" del EventHub", el cual tiene la información sobre el modo actual. Posteriormente, se procede a eliminar los ejes del modo anterior y a crear los nuevos. Este es un ejemplo de código de la destrucción de los ejes del modo anterior:

```
public void DestruirObjetosAnteriores()
{
    // Lógica para destruir objetos del modo anterior
```

```
    switch (EventHub.CurrentMode)
    {
        case EventHub.ModoInteraccion.Rotar:
            posicionadorCilindros.DestruirCilindros();
            break;
        case EventHub.ModoInteraccion.Mover:
            extrusionSpheres.DestruirEsferas();
            break;
        case EventHub.ModoInteraccion.Extruir:
            rotationalAxis.DestruirToroides();
            break;
    }
}
```

El método para crear los nuevos ejes es análogo.

Cuando se intenta editar la posición, rotación o dimensión de una pieza, primero se comprueba en qué modo está el programa. Cuando se está en un modo, solo deberían aparecer los ejes de dicho modo. Por ejemplo, si está en modo rotar, el usuario solo va a poder ver los ejes de rotación, y al interactuar con ellos podrá rotar la pieza. Independientemente de eso, por medidas de generalización y seguridad, cuando se va a rotar la pieza, se comprueba que el modo sea el de rotación.

En principio siempre va a ser así, pero así se evitan posibles problemas futuros si en algún caso aparecen los ejes de otros modos en la escena. En ese caso sólo podría interaccionar con los ejes del modo en el que se encuentre. Por ejemplo, en el script “CubeRotation”, encargado de rotar el cubo:

```
void Update()
{
    // Solo ejecutar el código si el modo rotar está habilitado
    if (EventHub.CurrentMode == EventHub.ModoInteraccion.Rotar)
    {
        // Lógica específica del modo rotar
        if (isMoving)
```

```
        {
            RotateObject();
        }
    }
}
```

4.13 ACTIVAR / DESACTIVAR OBJETOS

Cuando el usuario dispara con el raycast en una pieza, ésta pasa a estar "activada". Ello sirve para tener un control de las piezas que se están editando y para cambiar su opacidad a semitransparente. La semitransparencia es útil para poder seleccionar la parte de los ejes no visibles, y para tener una mayor intuición sobre la pieza que se está editando.

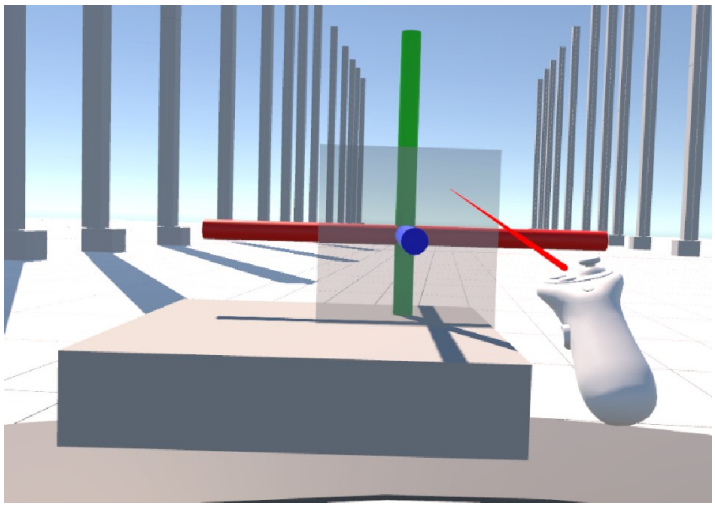


Ilustración 30: Ejemplo de pieza siendo activada al ser impactada por el raycast

Los métodos que se encargan de activar y de añadir transparencia a los objetos están definidos en "ModoControlador" (10.11). También hay definidos métodos que solo añaden semitransparencia a las piezas, sin que creen ejes. Eso es porque algunas herramientas (4.15, 4.17, 4.19) requieren que se seleccionen objetos, y no se desea que aparezcan los ejes. Por último, también hay definidos métodos que se encargan de desactivar la pieza, volviéndola opaca de nuevo.

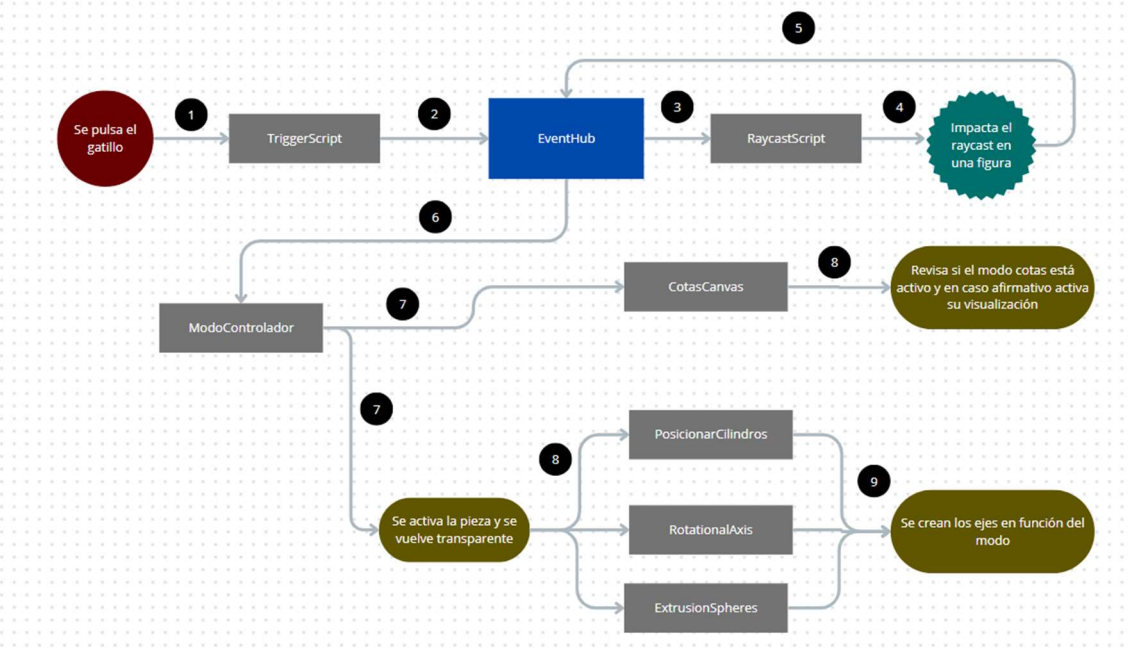


Ilustración 31: Estructura a alto nivel de la función activar objetos

Cuando el raycast impacta en una pieza, se disparan 2 eventos. El primero llama a un método que hace que se desactiven todas las piezas que estén activadas en ese momento:

```
public void DesactivateObject()
{
    activated = false;
    isTransparent = false;
    DestruirObjetos();
    RestoreOpacity();
    VisualizacionCotasOff();
}
```

El segundo evento disparado activa la pieza que ha sido impactada:

```
private void ActivateObject(RaycastHit hit)
{
                    ModoControlador          modoControlador          =
hit.collider.GetComponent<ModoControlador>();
    if (modoControlador != null)
```

```
    {
        //Debug.Log("Objeto activado");
        modoControlador.activated = true;
        modoControlador.isTransparent = true;
        modoControlador.CrearObjetos();
        modoControlador.AddTransparency();
        VisualizacionCotasOn(hit);
    }
    else
    {
        Debug.Log(" modoControlador es null");
    }
}
```

En este último método hay que recalcar un detalle importante. No se puede establecer directamente "activated = true". Al hacer eso, se estaría cambiando el valor del atributo "activated" en todas las instacias. Eso quiere decir que todas las piezas pasarían a estar activadas. En cambio, solo interesa activar la pieza que es impactada por el raycast. Es por eso que se obtiene la instancia específica del "ModoControlador" del objeto que ha sido impactado, para así solo activar ese objeto. Existen métodos similares, que omiten la parte de creación de los ejes.

Cambiar la transparencia de un material desde el Inspector de Unity es un paso sencillo, solo hay que cambiar el valor "alpha" del render. Sin embargo, para este proyecto se necesita hacer mediante código, lo cual complica la tarea. En este fórum de Unity [27] se muestra la parte del código (a más bajo nivel) para cambiar el render del material a opaco y a transparente. Este es el ejemplo de transparente:

```
private void AddTransparency()
{
    Renderer renderer = GetComponent<Renderer>();
    if (renderer != null)
    {
        foreach (Material mat in renderer.materials) // Itera sobre todos
los materiales
        {
```

```
            // Cambia el shader a transparente para cada material
                                                    mat.SetInt("_SrcBlend",
(int)UnityEngine.Rendering.BlendMode.One);
                                                    mat.SetInt("_DstBlend",
(int)UnityEngine.Rendering.BlendMode.OneMinusSrcAlpha);
            mat.SetInt("_ZWrite", 0);
            mat.DisableKeyword("_ALPHATEST_ON");
            mat.DisableKeyword("_ALPHABLEND_ON");
            mat.EnableKeyword("_ALPHAPREMULTIPLY_ON");
            mat.renderQueue = 3000;

            Color color = mat.color;
            color.a = 0.5f; // Semi-transparente
            mat.color = color;
        }
    }
    else
    {
        Debug.LogError("Renderer component not found.");
    }
}
```

4.14 ELIMINAR LOS OBJETOS CREADOS

Para mayor profesionalidad del programa, se ha añadido la opción de poder eliminar las piezas que se estén diseñando en el escenario. Para ello, el usuario deberá de seleccionar una pieza, disparándole con el raycast. Una vez la pieza está activada, si dispara con el raycast al canvas "Eliminate", se eliminará dicha pieza. El script encargado de destruir el objeto se llama "EliminateObject" (10.12). Este script se debe de añadir como componente al canvas.

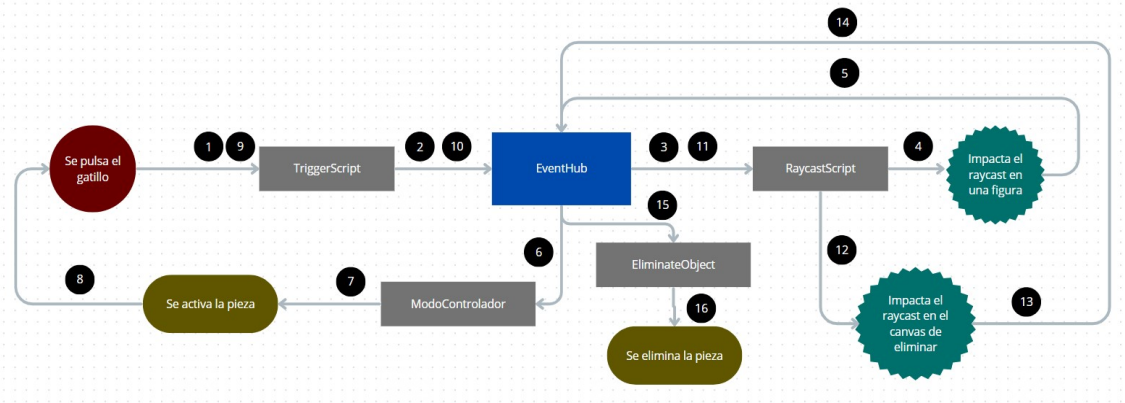
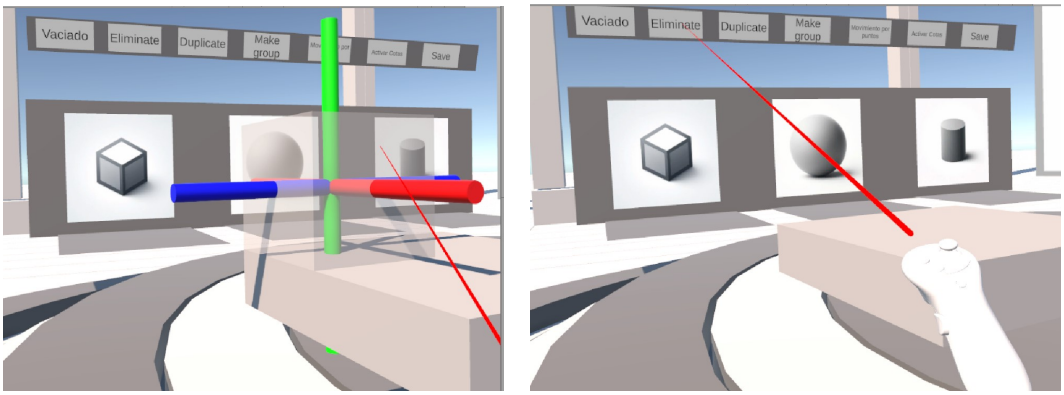


Ilustración 32: Estructura a alto nivel de la función de eliminar objetos

El canvas tiene 2 objetos adheridos como "hijos". Éstos son un panel y un texto. Su única función es visual. Además, se le añadió el componente "box collider" para que el canvas sea interactuable con el raycast. Para que el raycast sepa que se trata del canvas de "Eliminate", y no otro objeto, busca si tiene añadido el script "EliminateObject".



A) *B)*

Ilustración 33: Pasos para eliminar una pieza. A) Seleccionarla. B) Disparar al canvas "Eliminate"

Una vez que el raycast impacta en el canvas, se dispara un evento que llama al método en "EliminateObject" llamado "EliminatePart();". El funcionamiento de este método es que busca en todas las piezas su instancia de la clase "ModoControlador". En dicha instancia busca el atributo "activate". Solo elimina aquellos objetos que estén activados:

```
void EliminatePart(RaycastHit hit)
{
    // Encuentra todos los objetos en la escena que tienen el script ModoControlador
    ModoControlador[] objetos = FindObjectsOfType<ModoControlador>();
```

```
    // Revisa cada objeto
    foreach (ModoControlador objeto in objetos)
    {
        // Si la variable 'activated' es true, destruye el objeto
        if (objeto.activated)
        {
            Destroy(objeto.gameObject);
        }
    }
}
```

4.15 VACIADO

El vaciado es una de las herramientas más complejas y a la vez útiles en la industria del diseño industrial. Se trata de una operación booleana de resta entre la parte de intersección de 2 objetos. La lógica para gestionar la herramienta de vaciado está escrita en el script "Vaciado" (10.13). Este script se debe de añadir como componente al canvas.

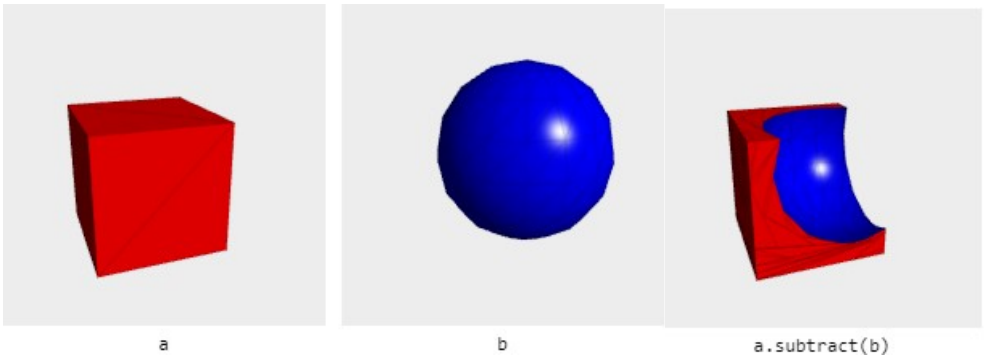


Ilustración 34: Ejemplo de funcionamiento de la operación booleana resta (Vaciado) [28].

Para utilizar esta función, el usuario tiene que disparar al canvas "vaciado" para activarla. Posteriormente tiene que disparar a los dos objetos en los que quiera realizar la operación. El primer objeto seleccionado corresponde a "a" y el segundo a "b" (Ilustración 34). Acto seguido, aparecerá el objeto resultado del vaciado de los 2 objetos anteriores. Se recuerda que el orden en el se seleccionan los objetos importa.

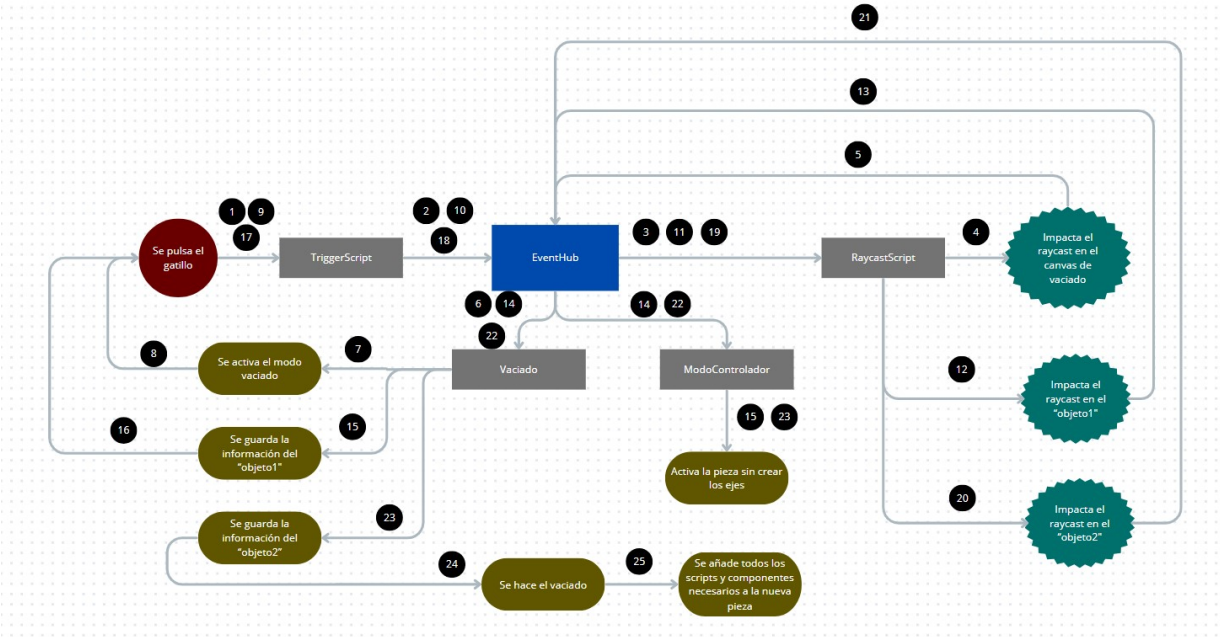
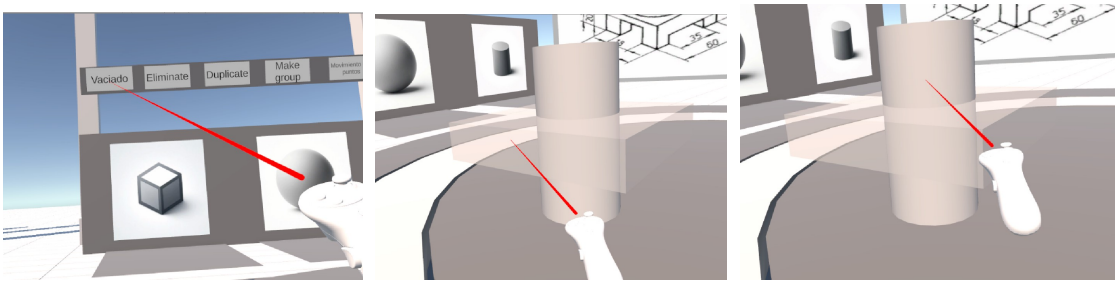


Ilustración 35: Estructura a alto nivel de la función de vaciado

El canvas funciona de la misma manera que el resto, cuando es disparado por el raycast, se comprueba si tiene como componente el script "Vaciado". En caso afirmativo, se dispara otro evento que llama al método "SelectObjects", dentro del script "Vaciado". Se encarga de gestionar la lógica de la operación. La primera vez que se llama a ese método, simplemente se establece el atributo "isVaciado" = true. Esto es para tener un control de en qué modo nos encontramos.



A) *B)* *C)*

Ilustración 36: Pasos para utilizar la herramienta vaciado. A) Disparar al canvas. B) Disparar al "objeto1". C) Disparar al "objeto2"

Lo primero que hay que habilitar es que una vez disparado el canvas, el usuario pueda disparar a dos objetos, para hacer el vaciado entre ellos. Como se ha explicado, ahora "isVaciado" = true. Esto hace que cuando se dispare con el raycast a un objeto nuevo, no se creen los ejes de traslación, rotación y dimensionado, sino que la pieza solo se vuelva semitransparente. Esto se hace para que el usuario sea consciente de las piezas que ha seleccionado. Esta lógica está escrita en "RaycastScript":

```
if (hit.collider.gameObject.tag == "PrimaryObject")
{
    if (vaciado.isVaciado)
    {
        EventHub.SelectedObjects(hit);   //Sirve para seleccionar 2 objetos en el modo de vaciado
        EventHub.ActivateOnlyObjectHit(hit);
    }
```

De la misma manera, cuando un objeto es impactado por el raycast mientras "isVaciado" = true, se vuelve a disparar el evento que llama al método "SelectObjects", contenido en "Vaciado". Este método almacena los datos de los objetos que son impactados en los atributos "objeto1" y "objeto2". El objetivo es que a "objeto1" se le reste la intersección entre "objeto1" y "objeto2", no al revés.

Una vez se tiene la información de los 2 objetos, se procede a hacer la operación de vaciado. Esto se realiza en el método "SubstractMeshes(GameObject objeto1, GameObject objeto2)".

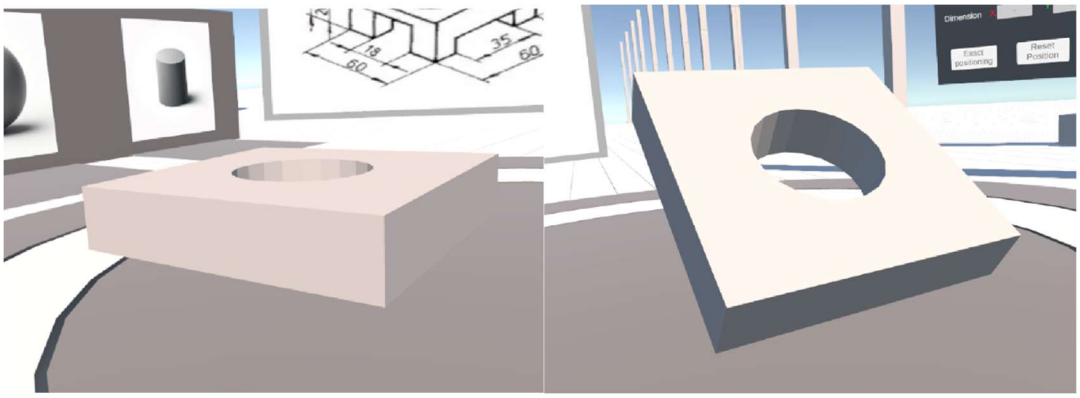


Ilustración 37: Resultado de la pieza de (Ilustración 36) tras haber aplicado el vaciado

La operación booleana de resta (vaciado) es un proceso muy complejo, que tiene que calcular la interacción entre las 2 mallas de los objetos, para así luego recalcular la malla del objeto resultante. Este es un problema que se lleva muchos años intentando resolver en la comunidad de diseñadores 3D. Intentar programar esta función desde cero, sería tema para otro proyecto entero. Es por ello que me he apoyado en la comunidad de desarrolladores para implementar algunas soluciones ya existentes. Aun así, su implementación no es tan trivial como pueda parecer.

El método utilizado proviene de la librería Parabox.CSG [28]. Su implementación sería:

```
// Operación booleana que realiza el vaciado
```

```
Model resultModel = CSG.Subtract(objeto1, objeto2);

// Crea un nuevo GameObject para mostrar el resultado

GameObject resultObj = new GameObject("Resulting Object");

MeshFilter meshFilter = resultObj.AddComponent<MeshFilter>();

MeshRenderer meshRenderer = resultObj.AddComponent<MeshRenderer>();


// Asigna la malla resultante y materiales al nuevo GameObject

meshFilter.mesh = resultModel.mesh;
```

Con estas líneas de código se genera el objeto, resultado del vaciado 2 objetos que el usuario había seleccionado previamente. Por defecto, no se destruyen “objeto1” y “objeto2”, hay que añadirlo al código.

Adicionalmente, el objeto resultado del vaciado (a partir de ahora objeto resultado) no se crea con ningún hijo (ejes de edición, emptyScale…), ni tiene añadidos los scripts necesarios para su edición. Esto se le añade a las piezas cuando se crean en “PrimitiveShapeGenerator” 4.8. Todo ello hay que volverlo a añadir de manera manual, con código. Como se considera que el “objeto1” es el principal, solo se traspasan los hijos del “objeto1” al objeto resultado.

4.15.1 Problema de origen de referencia al hacer vaciado

Si se implementase la herramienta de vaciado tal y como está explicada hasta el momento, se cometería un error poco intuitivo. Este “error” viene de cómo está construido el método de la librería Parabox.CSG [28], y de cómo se añade la malla al nuevo objeto resultado:

```
CSG.Subtract(GameObject objeto1, GameObject objeto2);
```

Como ya se ha explicado, al utilizar este método, se genera un objeto resultado del vaciado en la posición donde se encontraban el “objeto1” y el “objeto2”. El problema es que, aunque visualmente el objeto resultante esté donde antes estaban los otros 2 objetos, el origen de coordenadas para ese nuevo objeto es el (0,0,0).

En otras palabras, supongamos que los 2 objetos estaban aproximadamente en la posición (15, 15, 15). Al aplicar la función de vaciado, el nuevo objeto se crea en la misma posición visual, pero en su transformada de posición global aparece que está ubicado en el (0,0,0). Esto quiere decir que el centro de referencia del nuevo objeto está desplazado.

Este problema parece tener una solución inicial intuitiva: si se aplica la función, estando los dos objetos iniciales ya ubicados en el (0,0,0), el objeto resultante coincidirá con ellos, y su centro de referencia también estará en el (0,0,0). Simplificándolo bastante, así es como lo he resuelto. Por supuesto, después de crear el vaciado del objeto resultante, hay que devolverlo a su posición original para que el usuario no se dé cuenta de esto. En resumen, muevo los 2 objetos al (0,0,0), aplico la función para crear el objeto resultante y devuelvo dicho objeto a la posición donde estaban los 2 objetos inicialmente.

A este razonamiento solo hace falta añadirle una complicación extra. Los objetos 1 y 2 no están en la misma posición exacta. Ambos estarán ligeramente separados, y el objeto resultante se creará en una zona cercana, pero distinta. Se puede ver como que los centros de masa son distintos en los 3 objetos.

La única manera de saber dónde se va a crear el objeto resultante es haciendo la operación de vaciado y creándolo (no es el definitivo). Este objeto resultado provisional tiene su centro de referencia en el origen, no en su centro de masas. Por ello, calculo la distancia entre su centro de masas y el origen. Una vez teniendo el dato de esa distancia, se puede eliminar ese objeto resultado provisional. Aplico esa misma distancia a los objetos 1 y 2 (ahora estarán muy cerca del origen). De tal manera que al volver a hacer la operación de vaciado, el nuevo objeto resultado tendrá el centro de masas en el origen. El último paso es volver a mover el nuevo objeto resultado a la posición inicial, para que el usuario no aprecie el cambio.

Para que todo esto no afecte cuando quiera pasar los “hijos” de “objeto1” al objeto resultado, antes de hacer el vaciado guardo todos los hijos en una lista, y lo añado al nuevo padre al final.

4.15.2 Problema de bugs en la malla

Hay ocasiones donde puede la función de vaciado no funciona bien, debido a su gran complejidad a la hora de recalcular la malla del nuevo objeto resultado del vaciado. Cuando la operación de vaciado se hace pocas veces sobre la misma pieza, funciona sin problemas. El problema se da cuando se hace un número repetitivo de operaciones de vaciado sobre la misma pieza. La pieza resultante acaba con una malla muy compleja, donde se le van acumulando errores. Estos errores acaban mostrándose en forma de caras que no se renderizan.

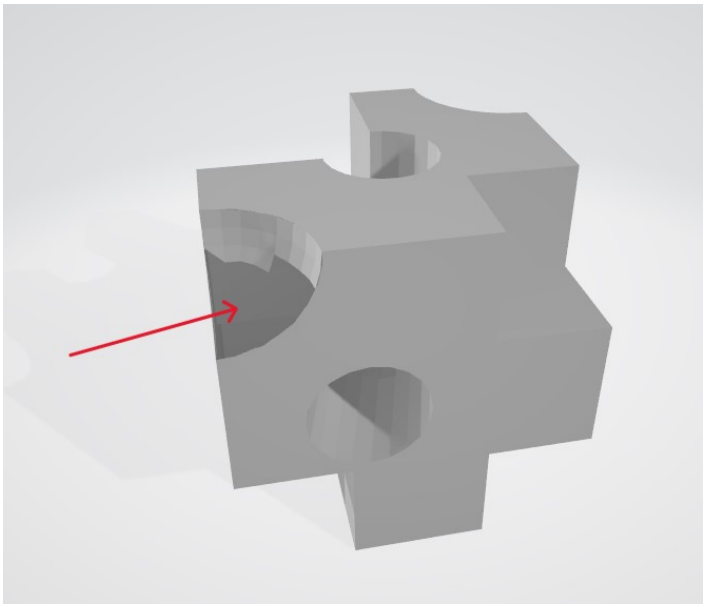


Ilustración 38: Ejemplo de error en la malla de una pieza tras haber aplicado varias veces la operación de vaciado

Se puede apreciar este error en la pieza de la Ilustración 38. La malla en la zona donde se le hizo el vaciado con una esfera, está incompleta. Cuando los errores en la malla son pequeños, se pueden solucionar con el programa Fusion 360 (4.24.1).

Después de investigar más a fondo sobre el asunto y de probar otras librerías con otras funciones, sólo hallé un método que lo mejoraba un poco. El método es recalcular las tangentes, normales y bordes de la malla del objeto resultado del vaciado:

```
resultModel.mesh.RecalculateNormals();
resultModel.mesh.RecalculateTangents();
resultModel.mesh.RecalculateBounds();
```

Estas funciones ayudan a ordenar un poco la malla, pero no solucionan del problema de verdad.

La comunidad de desarrolladores de Unity lleva mucho tiempo intentando solucionar este problema de operaciones de resta booleana en mallas. Después de toda la documentación leída hasta el momento, no he encontrado ninguna función que asegure que funciona a la perfección, al menos de manera gratis. Este enlace [29] de la comunidad de Unity puede servir como referencia para indagar más sobre este asunto. La mayoría de desarrolladores argumentan que es una función muy compleja para programar.

Después de importar la pieza de la Ilustración 38 en Fusion 360, se puede observar la complejidad de su malla después de tantas operaciones de vaciado:

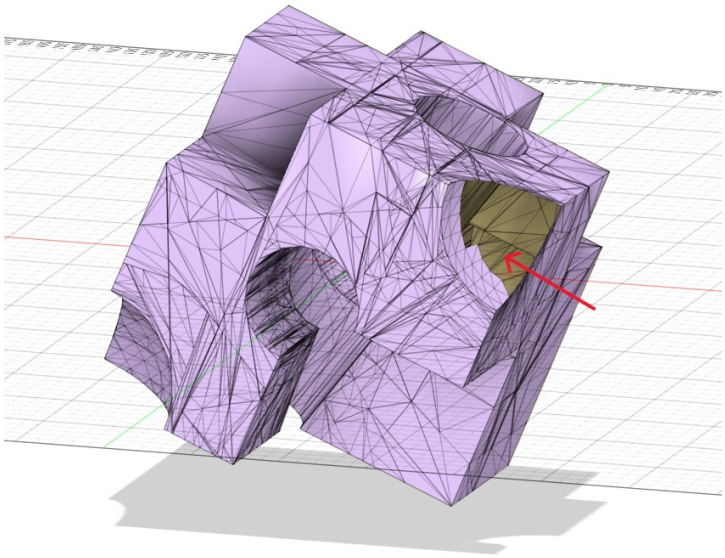


Ilustración 39: Malla de la pieza de la Ilustración 38 vista en Fusion 360

4.16 MOVIMIENTO POR PUNTOS

Esta es una herramienta muy útil para posicionar piezas, en función de la colocación y rotación de otras piezas. Sirve, por ejemplo, para especificar que dos caras planas de dos cuerpos sean paralelas, para que dos elementos sean coaxiales, o para especificar que dos puntos sean coincidentes. La inspiración de esta herramienta viene del programa RobotStudio.

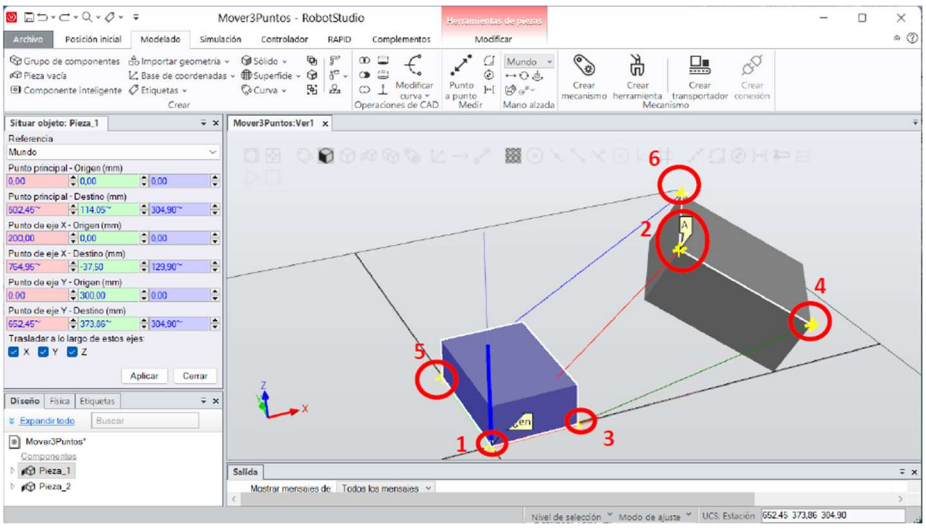


Ilustración 40: Funcionamiento de la herramienta de movimiento por puntos en RobotStudio

El funcionamiento es el siguiente: el usuario puede elegir si el movimiento es seleccionando 1, 2 o 3 puntos de cada pieza. Si selecciona 1 punto de cada pieza, esos puntos se vuelven coincidentes. Si selecciona 2 puntos de cada pieza, los primeros puntos serán coincidentes, y los otros 2 puntos estarán en la misma recta, que pasa también por los primeros puntos. Si selecciona 3 puntos de cada pieza, además, todos los puntos estarán contenidos bajo el mismo plano. Se recuerda que una recta se define por 2 puntos y un plano por 3 puntos.

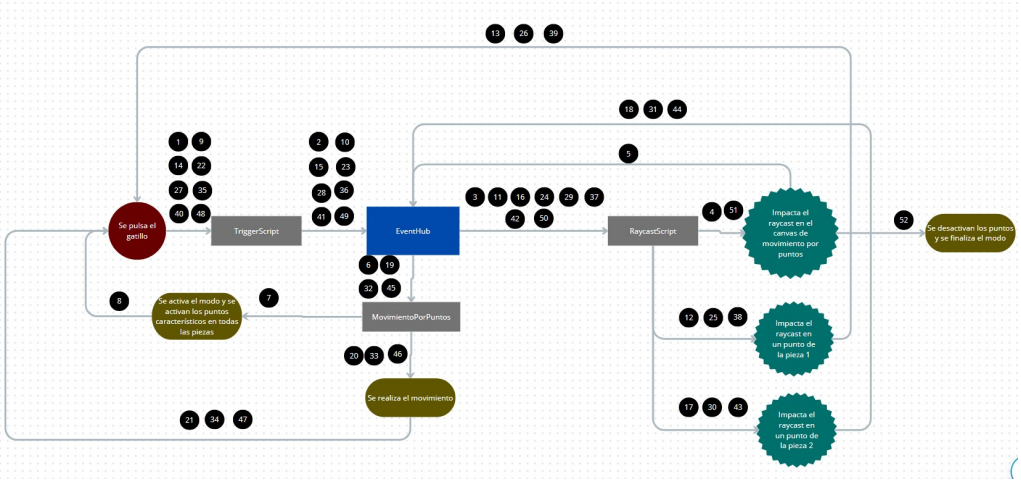


Ilustración 41: Estructura a alto nivel de la función movimiento por puntos

4.16.1 Creación de los puntos característicos

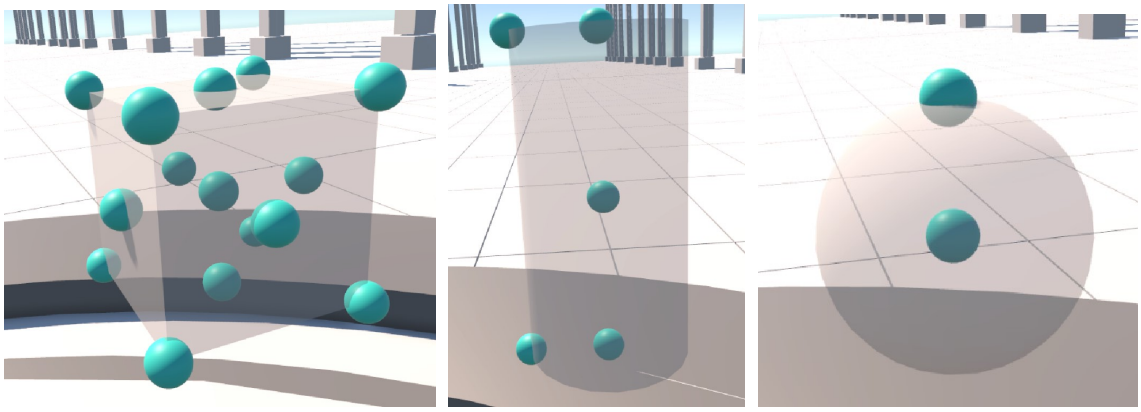
En este apartado se explica el criterio establecido y la manera en la que se añaden los puntos especiales a las piezas. El script que se encarga de ello se llama “CreateSpecialPoints” (10.14). Este script es el que tiene definido el método para crear los puntos especiales. Este método es llamado desde el script “PrimitiveShapeGenerator” cuando se generan las formas primitivas (4.8).

En la mayoría de programas CAD, se muestra un gran rango de opciones para seleccionar los puntos característicos de los objetos. En este proyecto nos hemos centrado en los que se cree que son los más útiles:

- **Cubo**: al crearse un cubo se generan puntos característicos en todos los vértices, centros de caras y centro del cubo.

- **Cilindro**: al crearse un cilindro se generan puntos característicos en los centros de las caras, en el centro del cilindro y en las tangentes.

- **Esfera**: al crearse una esfera se generan puntos característicos en el centro de la esfera y en la tangente.



A) B) C)

Ilustración 42: Puntos característicos creados en: A) Cubo. B) Cilindro. C) Esfera

Es por esta distinción por la que en el script "CreateSpecialPoints" hay 3 métodos, cada uno encargado de crear los puntos característicos de cada forma primitiva mencionada.

Los puntos característicos son esferas, y se han creado y guardado como prefabs. Al igual que con los ejes de traslación y rotación, este prefab se añade en los canvas que generan las formas primitivas, por lo que no hay que volver a preocuparse.

Cuando los puntos característicos son creados al principio, que es cuando se genera la pieza primitiva, hay que ocultar su visualización. Esto se hace para que no "molesten". Se volverá a activar su visualización mientras se esté usando la herramienta de movimiento por puntos. Para ocultarlos, se hace mediante el siguiente método:

```
public void DeActivateVisualizationSpecialPoints(GameObject point)
{
     //Hace que los puntos no estén visibles (no los elimina, solo deshabilita el renderer, los oculta)
    Renderer renderer = point.GetComponent<Renderer>();
    renderer.enabled = false;
}
```

En resumen, todos los puntos característicos se posicionan en la zona especificada anteriormente, en función de la pieza primitiva, se desactiva su visualización y se le cambia el color a azul turquesa.

4.16.2 La lógica del movimiento por puntos

El script que se encarga de orientar y recolocar las piezas en función de los puntos característicos se llama "MovimientoPorPuntos" (10.15). Esta herramienta empieza cuando el usuario dispara con el raycast en el canvas con el texto "Movimiento por puntos". En dicho canvas va añadido

el script "MovimientoPorPuntos". La primera vez que impacta el raycast en el canvas, se activa el modo. Para finalizar el modo, el usuario debe de volver a disparar al mismo canvas con el raycast.

Cuando empieza el modo, se dispara un evento que hace que comience el método "Alternate(RaycastHit hit)", definido en "MovimientoPorPuntos". Este método se encarga de volver a hacer visibles todos los puntos y de volver a cambiar su color a azul turquesa. Más adelante se muestra como se cambian algunos puntos a verde, de esta manera se reestablece su color.

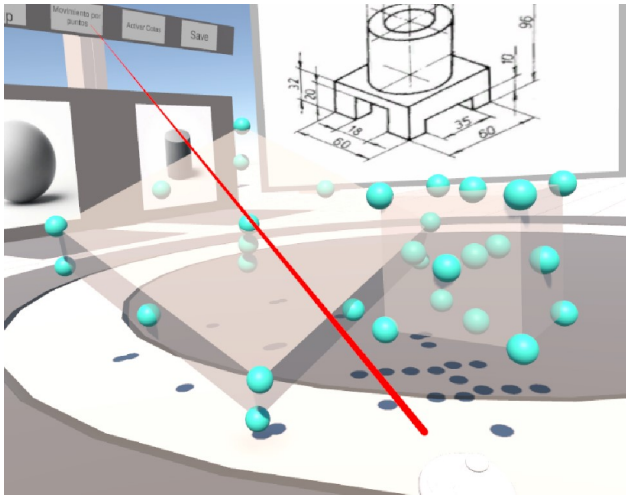


Ilustración 43: Activación del movimiento por puntos

Una vez está el modo "activo", el usuario puede empezar a disparar con el raycast a los puntos característicos de las piezas que quiera reorientar. El programa va almacenando en una lista todos los puntos característicos que el usuario va seleccionando. Cuando el usuario ha seleccionado 1 punto de cada pieza, se aplica el primer movimiento. La pieza, padre del primer punto seleccionado, es la que se mueve. La pieza se mueve hasta que los puntos característicos de las 2 piezas son coincidentes.

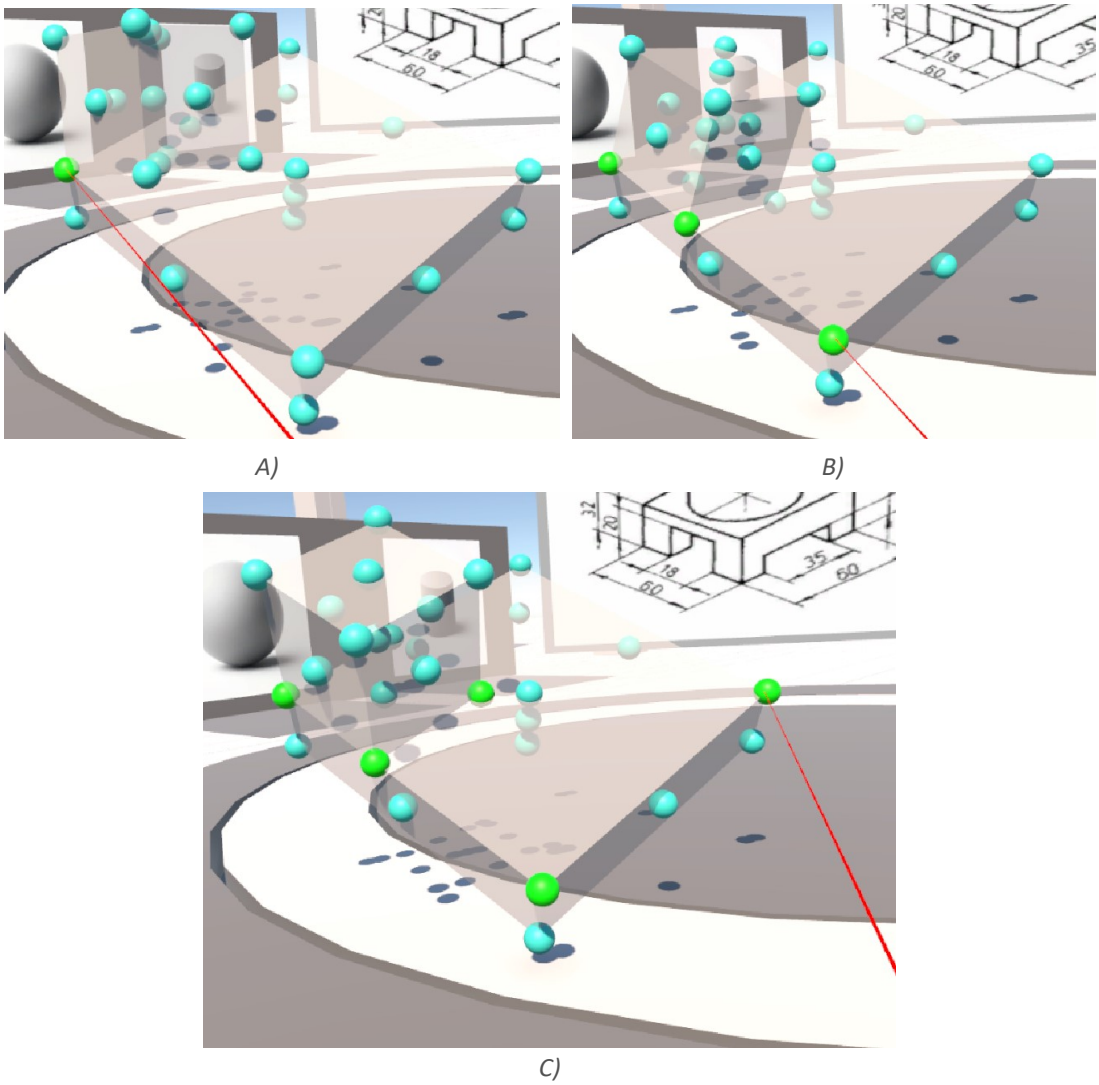
El usuario tiene la opción de seguir seleccionando puntos. Cuando ha seleccionado otro punto característico en cada pieza, estos últimos puntos se orientarán de tal manera que formen una línea recta con los primeros puntos (que son coincidentes). De esta manera, los 4 puntos seleccionados hasta el momento se contienen en una recta. Se recuerda que los segundos puntos seleccionados no tienen por qué ser coincidentes. Se muestra la parte de código encargada de dicho cálculo:

```
…

else if (count == 4) // Para pares de puntos subsiguientes, como 3 y 4, 5 y 6, etc.

{
```

```
    Transform refPoint = selectedPoints[0]; // Primer punto seleccionado como referencia

    Transform targetPoint = selectedPoints[count - 2]; // Penúltimo punto seleccionado

     Transform newPoint = selectedPoints[count - 1]; // último punto seleccionado


    RotateAroundPoint2(parentObject, refPoint.position, newPoint.position - refPoint.position, targetPoint.position - refPoint.position);

} …


void RotateAroundPoint2(GameObject obj, Vector3 point, Vector3 targetDir, Vector3 currentDir)

{

    // Normaliza los vectores de dirección

    targetDir.Normalize();

    currentDir.Normalize();


    // Calcula el eje de rotación como el perpendicular al plano formado por los vectores targetDir y currentDir

           Vector3 axisOfRotation = Vector3.Cross(currentDir, targetDir).normalized;


    // Calcula el ángulo necesario para la rotación

        float angle = Vector3.SignedAngle(currentDir, targetDir, axisOfRotation);


    // Aplica la rotación alrededor del punto especificado utilizando el eje calculado

    if (axisOfRotation != Vector3.zero) // Asegurarse de que el eje no sea cero

    {

        obj.transform.RotateAround(point, axisOfRotation, angle);
```

```
    }
}
```

Si el usuario selecciona un nuevo punto en cada pieza, se seguirán recolocando. Los puntos que ya han sido orientados no se moverán. Los nuevos puntos seleccionados se orientarán de tal manera que estén contenidos en el mismo plano que los anteriores. Se recuerda que éstos últimos puntos no tienen por qué ser coincidentes.



A) *B)*

C)

Ilustración 44: Pasos ejecutando el movimiento por puntos. A) 1 punto en cada pieza. B) 2 puntos en cada Pieza. C) 3 puntos en cada pieza

No tiene sentido añadir más puntos, ya que con éstos ya queda perfectamente definida la posición de la pieza. El usuario puede salir de este modo en cualquier momento, volviendo a disparar al canvas de movimiento por puntos. Para que sea más intuitivo, cada vez que el usuario selecciona un punto característico, éste se vuelve de color verde.

Para profesionalizar un poco el programa, se han añadido unas líneas de código que ahorran errores debidos a un mal uso de la herramienta por el usuario. Cuando el usuario selecciona el primer punto característico, se guarda el objeto al que pertenece dicho punto.

Cuando selecciona el segundo punto, se guarda también ese objeto que contiene al segundo punto. Cuando va a seleccionar los puntos 3 y 5, solo pueden pertenecer al objeto padre del punto 1. Cuando va a seleccionar los puntos 4 y 6, estos puntos deben pertenecer al objeto padre del punto 2. De igual manera, se impide que el objeto que contenga el punto característico 1 sea el mismo que el objeto que contenga al punto 2. De esta manera se garantiza un correcto uso de esta función, evitando errores inesperados.

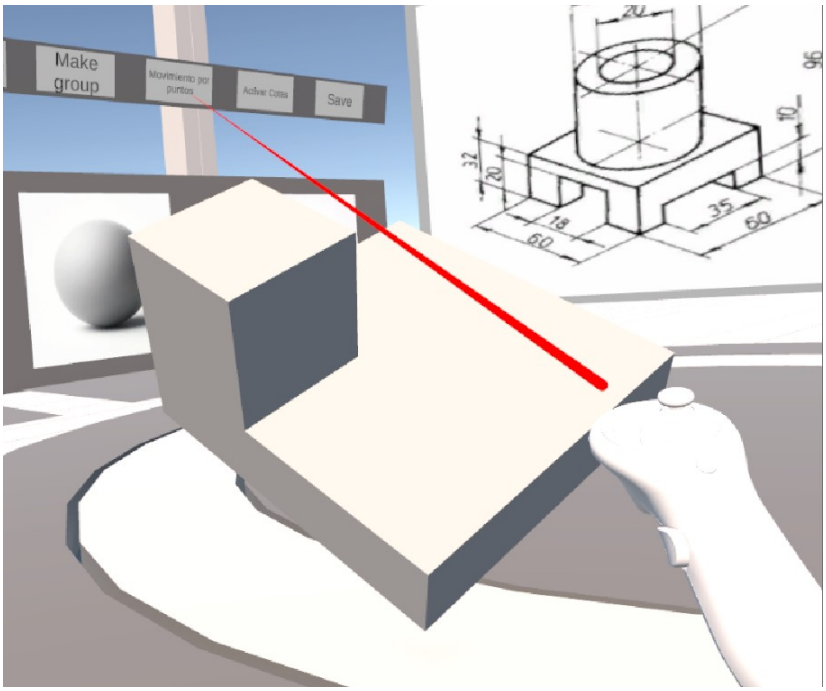


Ilustración 45: Resultado de las piezas de la Ilustración 44 después de finalizar el movimiento por puntos

Cuando se realiza la función de grupo (4.17) o vaciado (4.15), se crean nuevas piezas. Hay que establecer un criterio para añadir los nuevos puntos especiales a esas nuevas piezas creadas. El seguido en este proyecto es:

- **Vaciado**: Se mantienen los puntos característicos de ambos objetos originales: “objeto1” y “objeto2”.

- **Grupo**: Se añaden los puntos característicos de todos los objetos que forman el grupo.

4.17 CREACIÓN DE GRUPOS

En esta sección se explica el funcionamiento de la herramienta que permite al usuario hacer grupos de piezas. Los métodos para dicha función están escritos en el script "GroupCreation" (10.16). Este script debe ser añadido al canvas correspondiente al modo de creación de grupos.

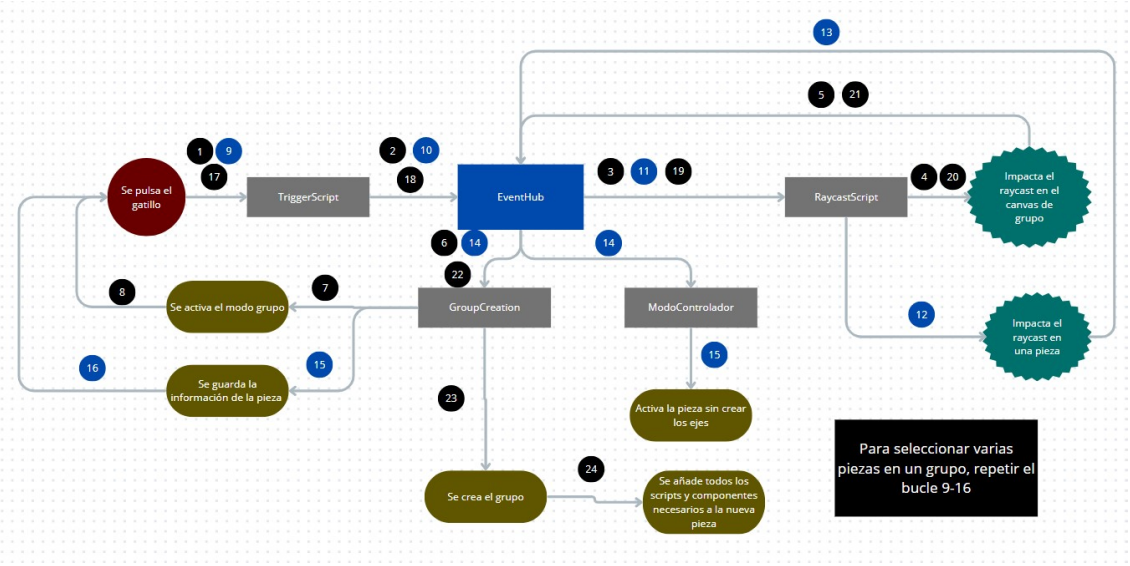
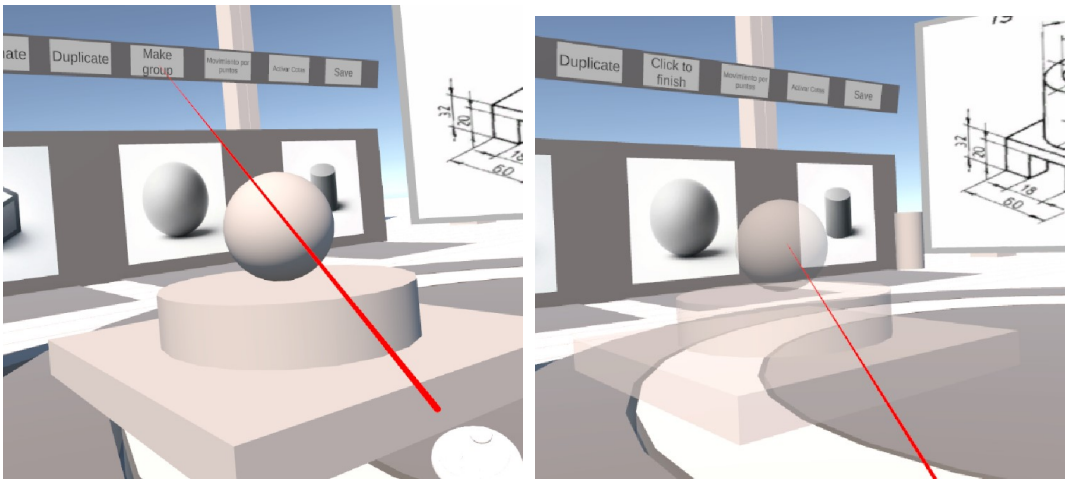


Ilustración 46: Estructura a alto nivel de la función de creación de grupos

El modo empieza cuando el usuario dispara en el canvas que tiene el texto "Make group". En ese momento, el texto se cambia a "Click to finish". El usuario puede ir seleccionando todas las piezas que quiere añadir en el grupo. Cuando haya seleccionado todas las piezas deseadas para unir, tiene que volver a disparar en el canvas para finalizar el modo y crear el grupo.



A) *B)*

Ilustración 47: Pasos para crear un grupo. A) Seleccionar en el canvas "Make group". B) Seleccionar las piezas y disparar en el canvas "Click to finish".

Los eventos funcionan de manera similar al resto de modos. Se dispara un evento cuando el raycast impacta en el canvas, se establece el modo como "activo", y el usuario puede empezar a seleccionar piezas. Cuando las selecciona, se dispara otro evento que hace ejecutar el método principal del script "GroupCreation". Se recuerda que el evento solo se dispara cuando el raycast impacta en una pieza y el modo de crear grupo está activo. El modo se activa cuando el raycast impacta en el canvas de crear grupo.

Cuando se inicia la función de crear un grupo, se crea un nuevo objeto "padre", este objeto contendrá como hijos a todas las piezas que el usuario seleccione para formar el grupo:

```
private void CreateNewGroupParent()
{
    // Incrementa el contador cada vez que se crea un nuevo grupo
    groupCounter++;

    // Crea un nuevo objeto padre para los objetos agrupados y a◆ade el
contador al nombre
    groupParent = new GameObject("GroupParent_" + groupCounter);
    groupParentNew = new GameObject("GroupParent_new" + groupCounter);
}
```

Cuando el usuario haya acabado de seleccionar las piezas del grupo y dispare de nuevo el canvas para finalizar, se ejecutará toda la lógica para crear el grupo con las piezas que ha seleccionado.

El objeto creado en (figura de arriba) es el único que quedará, los hijos se eliminarán para que el resto de funciones sigan funcionando. En principio, el nuevo objeto es un objeto vacío, hay que añadirlo componentes para que se convierta en una nueva pieza. Lo más importante es su malla. Se utiliza el método:

```
CombineChildMeshes(GameObject parent, MeshFilter parentMeshFilter)
```

Este método asigna al padre la malla resultado de la mezcla de las mallas de los hijos. Se recuerda que los hijos son las piezas que el usuario ha seleccionado. Al hacer esta operación de asignarle una nueva malla al padre surge un problema. Y es que su centro de masas y su centro de referencia de posición no coinciden. Es un error análogo al que ocurre en el vaciado (4.15.1) y se resuelve de la misma manera.

Al nuevo padre se le deben añadir todos los scripts que se les añaden a las formas primitivas cuando se crean. También se le deben de añadir hijos como el "emptyScale" (4.11.3) y los puntos característicos (4.16.1). Por último, se eliminan como hijos las piezas que el usuario ha

seleccionado. Cuando el usuario vuelva a interaccionar con el objeto padre, lo hará como si fuese una única pieza.

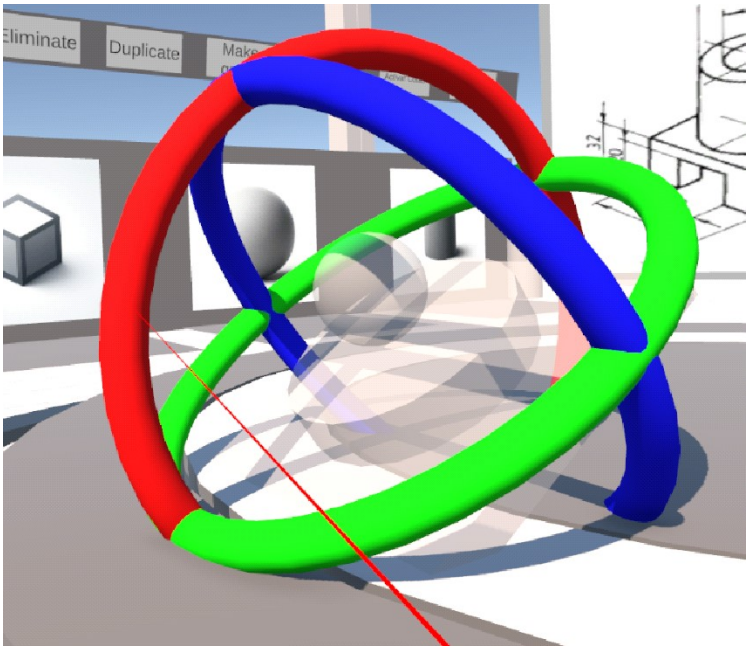


Ilustración 48: Ejemplo de rotación de un grupo. Se comporta como una pieza individual

4.18 AÑADIR COTAS

Es de vital importancia que los diseñadores puedan consultar las dimensiones de las piezas que están diseñando. Es por ello que se da la posibilidad de que se muestren las cotas de todas aquellas piezas que están "activas". El script que se encarga de añadir y quitar las cotas a las piezas se llama "TextoCotas" (10.17).

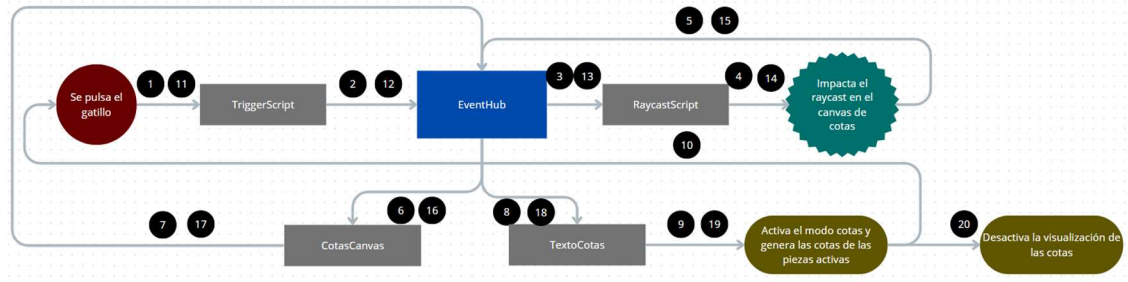


Ilustración 49: Estructura a alto nivel de la función de añadir cotas

El usuario puede elegir activar o desactivar las cotas en cualquier momento. Para activarlas, debe disparar con el raycast en el canvas que tiene el texto "Activar Cotas". Acto seguido, el texto cambiará a "Desactivar cotas". El usuario deberá volver a disparar el raycast en el canvas

para desactivarlas. El script que se encarga de la gestión de los textos en el canvas y de si las cotas están "activas" o no se llama "CotasCanvas" (10.18).

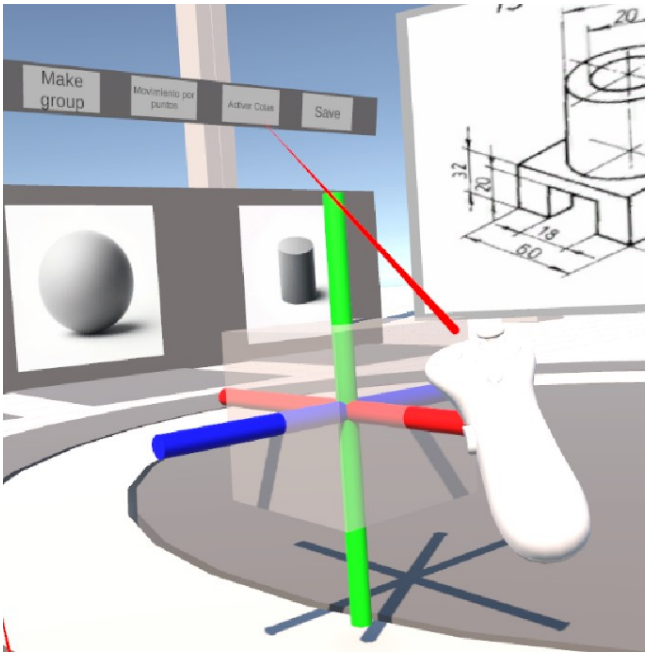


Ilustración 50: Usuario activando las cotas, disparando al raycast

En este proyecto, se muestran las dimensiones de alto, ancho y largo, al activar las cotas de las piezas. Se considera que son las más esenciales. Estas cotas están formadas por un canvas y por un texto, y son "hijos" de las piezas que se están editando. De esta manera, al mover las piezas, también se mueven las cotas.

Las cotas se crean y se asignan como "hijos" cuando se generan las formas primitivas (4.8). Existe un argumento booleano que contiene la información de si es la primera vez que se crean las cotas. En dicho caso, desactiva su visualización llamando al método "VisualizaciónCotaOff(" del script "TextoCotas".

Cuando se dispara con el raycast al canvas para activar las cotas, se dispara un evento que llama al método:

```
UpdateTexts(Transform pieza, Canvas canvas, bool firstCreation)
```

Este método se encarga de recalcular las dimensiones de la pieza por si han cambiado y cambia el texto de las cotas si es necesario. Por último, activa su visualización para que sean visibles. Solo activa la visualización de aquellas piezas que estén "activas" o seleccionadas. Para ello, busca en todas las instancias de "ModoControlador", y comprueba si el atributo "activated" = true.

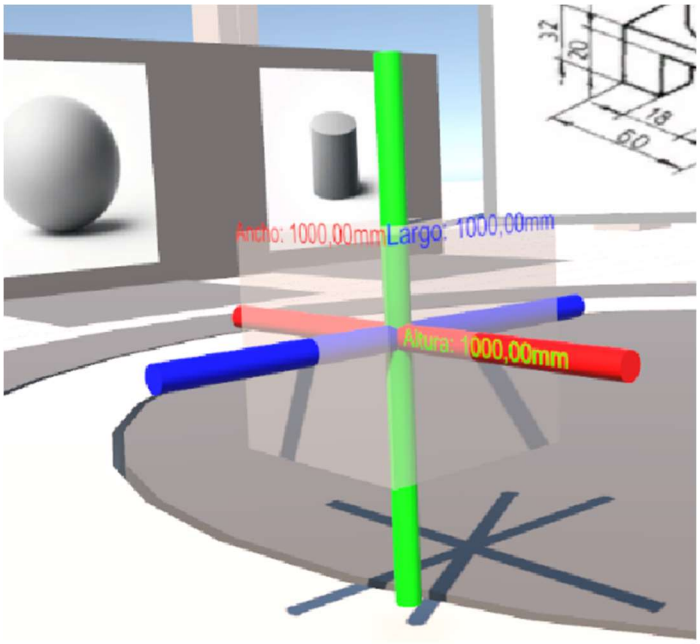


Ilustración 51: Cotas activas en un cubo

Como se puede ver en la Ilustración 51, la longitud del texto de las cotas coincide con el ancho de la pieza. Esto es así independientemente del tamaño de la pieza. Esto se debe a que se estableció el ancho de las cotas de manera proporcional a la dimensión de la pieza:

```
// Fijar el tamaño del texto a un ancho fijo proporcional a la dimensión
float targetWidth = 0.19f * dimension / 1000;  // Ancho fijo en metros
float textLength = text.text.Length * text.fontSize * 0.01f; // Estimación del largo del texto
float scale = targetWidth / textLength; // Calcula la escala necesaria para mantener el texto dentro del ancho deseado
text.transform.localScale = Vector3.one * scale;
```

Por último, hay que tener en cuenta el case en que se modifique las dimensiones de la pieza mientras las cotas estén activas. Ese es el caso de redimensionado de las piezas (4.11). Hay que actualizar el valor de las cotas constantemente, cada vez que se produce un cambio mínimo en la pieza. Es por ello que cuando se modifica el tamaño de una pieza en “ExtrusionMovement”, se dispara un evento que llama al método (escrito arriba), que actualiza el valor de las cotas.

```
GameObject cotasCanvas = GameObject.Find("Cotas_Canvas");
CotasCanvas cotasCanvasScript = cotasCanvas.GetComponent<CotasCanvas>();
// Comprueba si debe de actualizar las cotas o no, solo si el modo cotas está activo
```

```
if(cotasCanvasScript.cotasActivas)
{
    Canvas canvas = transform.parent.GetComponentInChildren<Canvas>();
    EventHub.ActualizarCotas(transform.parent.parent, canvas, false);
}
```

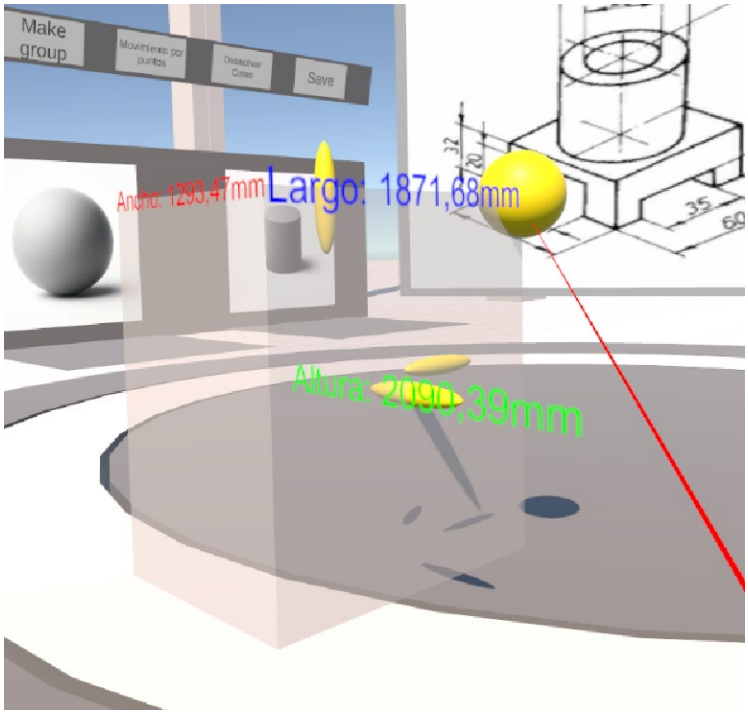


Ilustración 52: Cotas siendo modificadas al momento al redimensionar la pieza

4.19 TRASLADAR, ROTAR Y REDIMENSIONAR DE MANERA EXACTA

Las herramientas proporcionadas hasta ahora para trasladar, rotar y redimensionar son muy útiles para hacer ajustes rápidos y de manera intuitiva en las piezas. Sin embargo, el sector del diseño industrial es conocido por la exactitud de las medidas y por las ínfimas tolerancias de medidas. Es por ello que es imprescindible dar la posibilidad de que el usuario pueda añadir estas medidas de manera manual. El script que maneja toda la lógica en este modo se llama "Keyboard" (10.19). Este script debe de ir asignado a un objeto vacío en la jerarquía de Unity.

La solución propuesta a esta función es mediante la implementación de un teclado numérico. Primero, el usuario empieza el modo "exacto" (para abreviar) y en la pantalla se le muestran todos los datos de traslación, rotación y dimensiones de la pieza que seleccione. El modo exacto se activa disparando con el raycast al botón que pone "Exact positioning" Posteriormente, disparando con el raycast, va a poder seleccionar la casilla del valor que quiere cambiar.

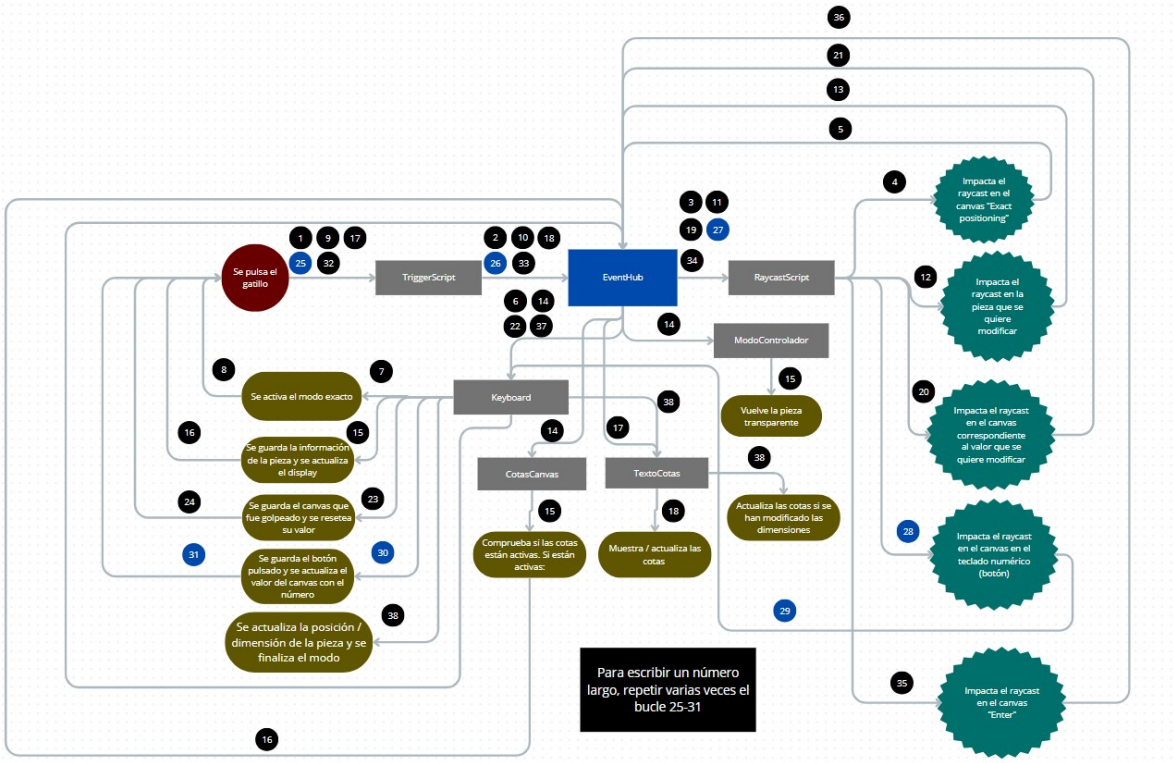
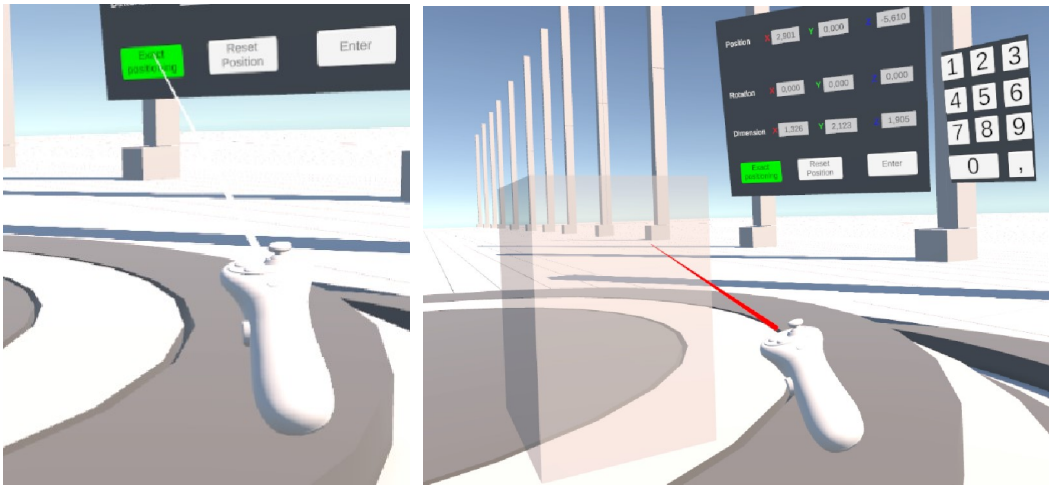
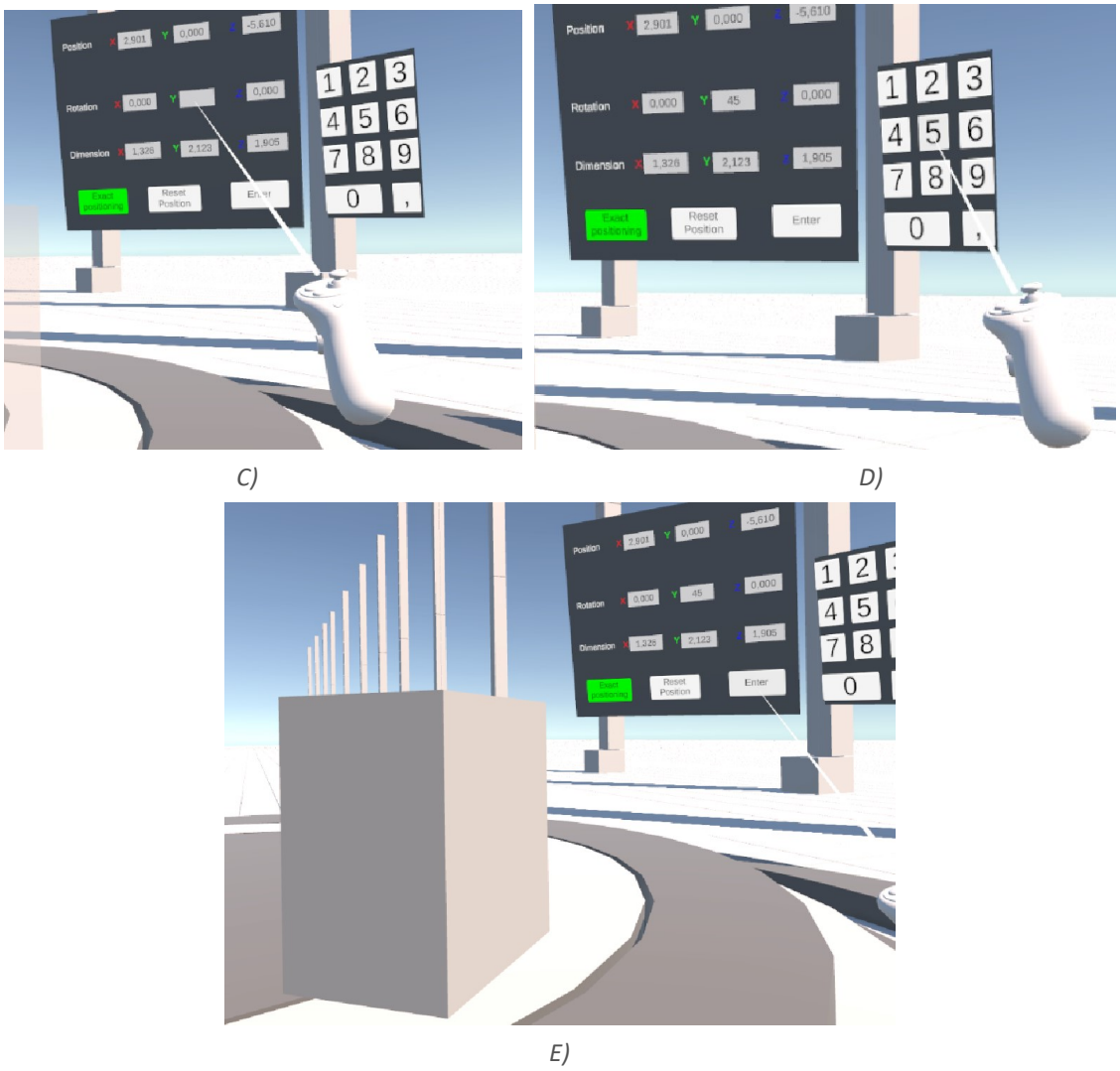


Ilustración 53: Estructura a alto nivel del modo exacto

Una vez haya seleccionado la casilla, debe introducir el valor numérico, disparando con el raycast en el teclado. Cuando esté satisfecho, debe disparar al botón "Enter". Si quiere resetear la posición y rotación de la pieza seleccionada, debe disparar al botón "Reset Position".



A) *B)*



C) *D)*

E)

Ilustración 54: Pasos para usar la "función exacta". A) Disparar al canvas "Exact positioning". B) Seleccionar la pieza. C) Disparar a la casilla con el valor que se quiere cambiar. D) Introducir el nuevo valor en el teclado numérico. E) Disparar a "Enter"

En este punto, me gustaría explicar cómo aprendí a gestionar el disparo del raycast en los canvas a un nivel más alto, sin necesidad de programación. Hasta ahora, el raycast impactaba en un canvas, que tenia un “Box collider”. Cuando el raycast impactaba en el collider, disparaba un evento. Todo ello se tenía que escribir con código. Con los botones, no hace falta hacerlo así. Los botones son unos elementos de interfaz de usuario de Unity que permiten disparar un evento, asignándolo en el inspector de Unity:



Ilustración 55: Configuración de eventos a alto nivel en el Inspector de Unity

Por ejemplo, cada vez que se dispara con el raycast al botón, se ejecuta el método "OnButtonPress", contenido en el script "Keyboard".

El modo "exacto" comienza cuando el usuario dispara con el raycast al canvas "Exact positioning". En ese momento, se dispara un evento que ejecuta el método:

```
public void StartPositioningPressed();
```

Con este método, se "activa" el modo exacto. Ahora el usuario debe de elegir la pieza que quiere modificar. Para eso debe de disparar con el raycast a esa pieza. Acto seguido, se dispara otro método, "UpdateHitObjectInfo(RaycastHit hit);" que actualiza los textos con la posición y las dimensiones de la pieza.

Ahora el usuario tienen la posibilidad de cambiar dichos valores. Para ello, debe de disparar con el raycast en el canvas con el valor que quiera cambiar. Ello hace saltar un evento que ejecuta el método "GetHitInfo(RaycastHit hit)". Este método registra en el código el parámetro que se quiere modificar.

A continuación, el usuario debe de introducir en el teclado numérico el valor que quiere cambiar. Al pulsar cada botón, se dispara un evento que activa el método "OnButtonPress()":

```
public void OnButtonPress()
{
    // Obtener el botón que disparó el evento
                            GameObject            buttonObject            =
EventSystem.current.currentSelectedGameObject;

    if (buttonObject != null)
    {
        // Obtener el texto del botón
                                                  buttonText            =
buttonObject.GetComponentInChildren<TextMeshProUGUI>().text;
```

```
        if (localText != null)
        {
            SumarNumero();
        }
    }
}
```

A medida que se van pulsando distintos botones, se van guardando los valores en una cadena de caracteres. A medida que va pulsando los botones, se va mostrando el número en el display. Cuando ha terminado de escribir el número, debe de disparar con el raycast en el botón "Enter".

Cuando pulsa dicho botón, se dispara un evento que activa el método "ApplyTextValuesToObject()". Este método actualiza todos los cambios realizados en el display, y los aplica en el objeto. Hay que tener en cuenta que se debe de pasar el texto a número. Ello se hace con el método "TryParse()".

También hay que modificar el valor de las cotas si están activadas:

```
if (cotasCanvasScript.cotasActivas)
{
                                Canvas canvas = lastObjectHit.collider.transform.GetComponentInChildren<Canvas>();
      EventHub.ActualizarCotas(lastObjectHit.collider.transform, canvas, false);
}
```

Si se ha modificado la dimensión de la pieza, hay que acordarse también de modificar la dimensión del objeto vacío "emptyScale", para que los objetos "hijos" de la pieza no sufran distorsiones (4.11.3).

4.20 DUPLICAR OBJETOS

Para ahorrar al usuario trabajo repetitivo, se ha añadido la herramienta para duplicar aquellas piezas que esté diseñando. Esta parte del código está recogida en el script "DuplicateObjects" (10.20). El script va añadido al canvas con el texto "Duplicate".

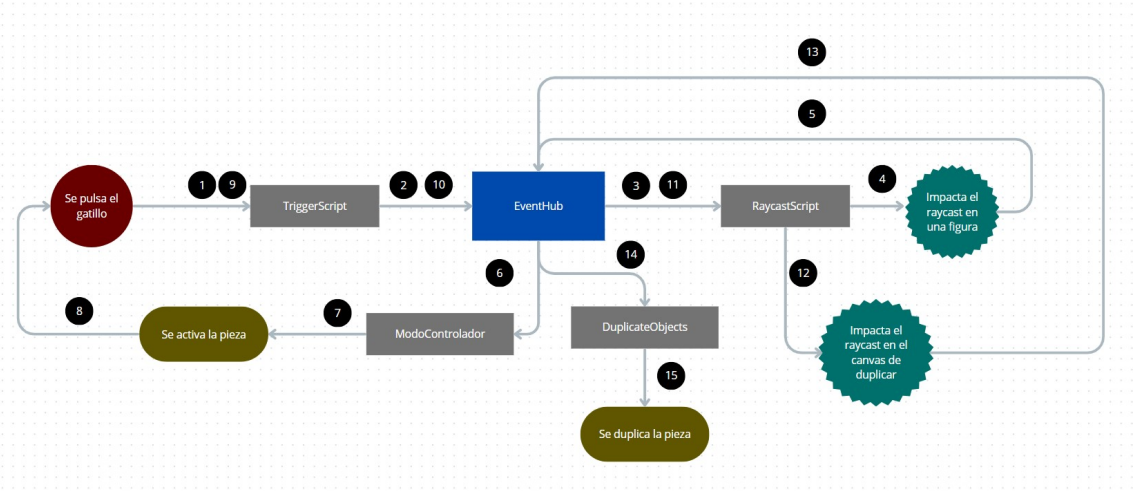


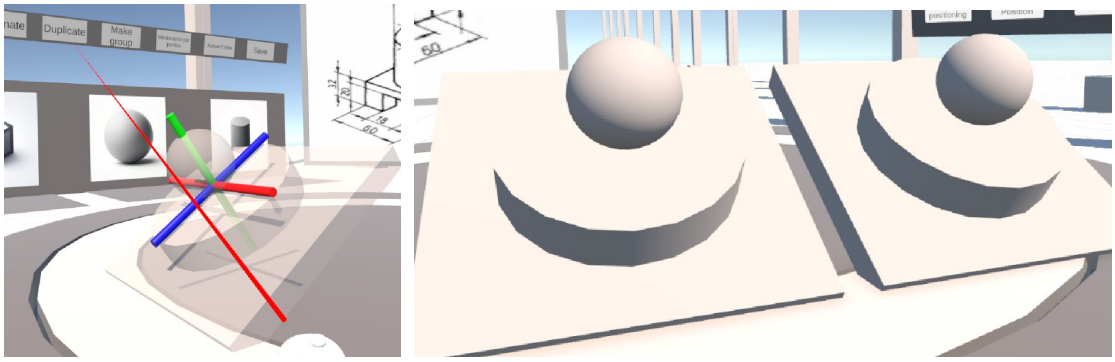
Ilustración 56: Estructura a alto nivel de la función de duplicar objetos

Cuando el usuario dispara con el raycast a dicho canvas, se dispara un evento que llama al método:

```
void DuplicateObject(RaycastHit hit)

{

     // Encuentra todos los objetos en la escena que tienen el script
ModoControlador

    ModoControlador[] objetos = FindObjectsOfType<ModoControlador>();


    // Revisa cada objeto

    foreach (ModoControlador objeto in objetos)

    {

        // Si la variable 'activated' es true, destruye el objeto

        if (objeto.activated)

        {

            // Instanciar (duplicar) el objeto en la misma posición y con
la misma rotación


             GameObject objetoDuplicado = Instantiate(objeto.gameObject,
objeto.gameObject.transform.position,
objeto.gameObject.transform.rotation);
```

```
            // Se cambia el nombre del objeto duplicado
            objetoDuplicado.name = objeto.gameObject.name + "_Duplicado";
            //Elimina los hijos de dentro de Empty Scale, porque si no luego se quedan fijos
            EliminarHijosInnecesarios(objetoDuplicado);
        }
    }
}
```

Este método busca todas aquellas piezas que están "activadas", y las duplica en la misma posición de la original. También elimina los ejes de traslación, rotación y redimensionado del objeto duplicado. Esto es porque si no, se quedan en la escena para siempre. Si se revisa los scripts encargados de la creación de esos ejes (4.9.1, 4.10.1, 4.11.1), cada vez que se crean, se guardan en una lista, para luego destruir los que estén en esa lista cuando se quiere cambiar de ejes. El problema es que, al duplicar los ejes, no figuran en esa lista, por lo que nunca se destruyen. Es por eso que se destruyen aquí de manera manual.



A) B)

Ilustración 57: Ejemplo de uso de la herramienta de duplicar objetos. A) Disparar al canvas "Duplicate". B) Resultado de los objetos duplicados

4.21 GUARDADO

Se ha habilitado la función de que los usuarios exporten las piezas que estén diseñando. El script encargado de ello se llama "SaveObject" (10.21). Cuando el usuario dispara con el raycast al canvas de "guardado", se llama al método "ExportarPiezaComoFBX(RaycastHit hit)".

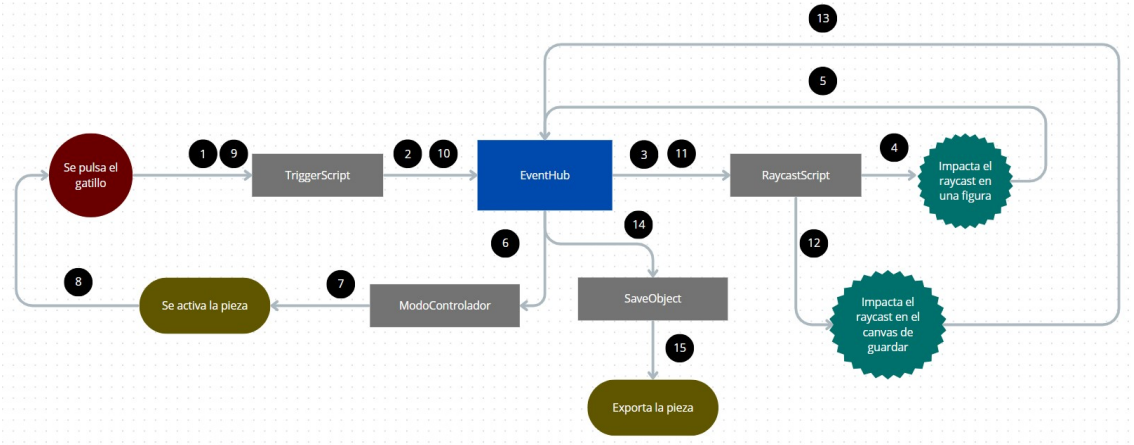


Ilustración 58: Estructura a alto nivel de la función de guardado

El método busca todas las piezas en el escenario que estén "activas" y las duplica temporalmente. Se eliminan todos los hijos de las piezas duplicadas. Posteriormente, se exporta dicha pieza duplicada y sin hijos. Las funciones para exportar están escritas por librerías de Unity. El formato de exportación es .fbx y el directorio es el que se especifique en el código. Por último, se destruye el objeto duplicado.

4.22 ANIMACIONES

Para añadir mayor intuitividad cuando el usuario dispara en los canvas para activar las funciones, se han añadido animaciones a los canvas cada vez que el raycast impacta en estos. El script se llama "CanvasClickAnimation" (10.22), y se le añade a todos los canvas. También se les debe de añadir el componente de Unity "Animator".

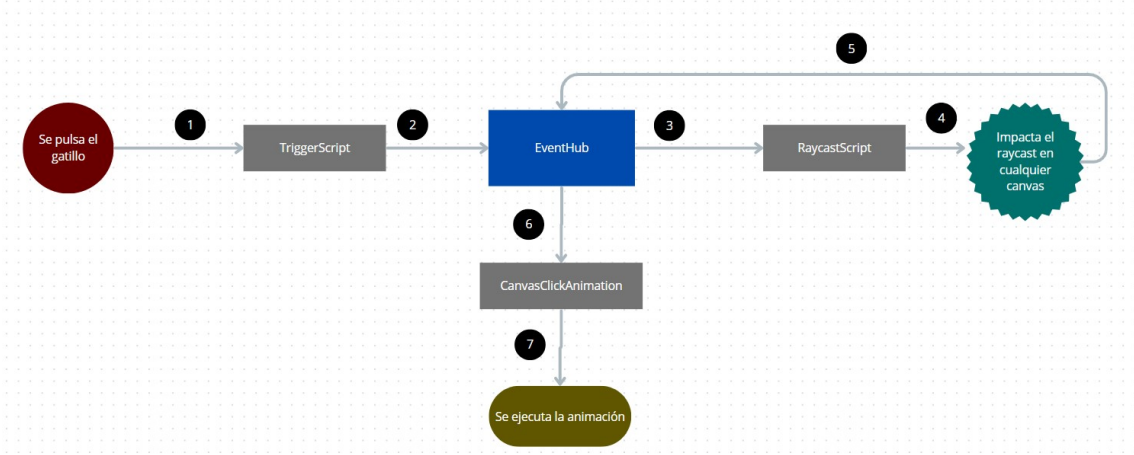


Ilustración 59: Estructura a alto nivel de la ejecución de animaciones

Parte de esta función se hace a alto nivel. La animación se hace desde el programa Unity. En este caso, se ha elegido que el canvas haga un pequeño "zoom in" y "zoom out" cuando es seleccionado. Se debe de añadir las animaciones creadas al atributo "Controller" del componente recién creado "Animator".

Con todo esto definido, solo hay que llamar a la animación desde el script cuando se desee. En el script "RaycastScript", se dispara un evento cada vez que un canvas es seleccionado. Ese evento llama al método "PlayClickAnimation(Canvas canvas)". Por último, se dispara la animación:

```
// Se obtiene el componente Animator del panel
Animator animator = panelTransform.GetComponent<Animator>();
if (animator == null)
{
    Debug.LogError("No se encontró el componente Animator en el Panel.");
    return;
}

// Se dispara el trigger de la animación
animator.SetTrigger("CanvasSelected");
```

4.23 ESCENARIO

Para una mejor experiencia del usuario, se ha construido un escenario 3D en el entorno virtual. Todos los elementos 3D creados fueron hechos con el programa Fusion 360.

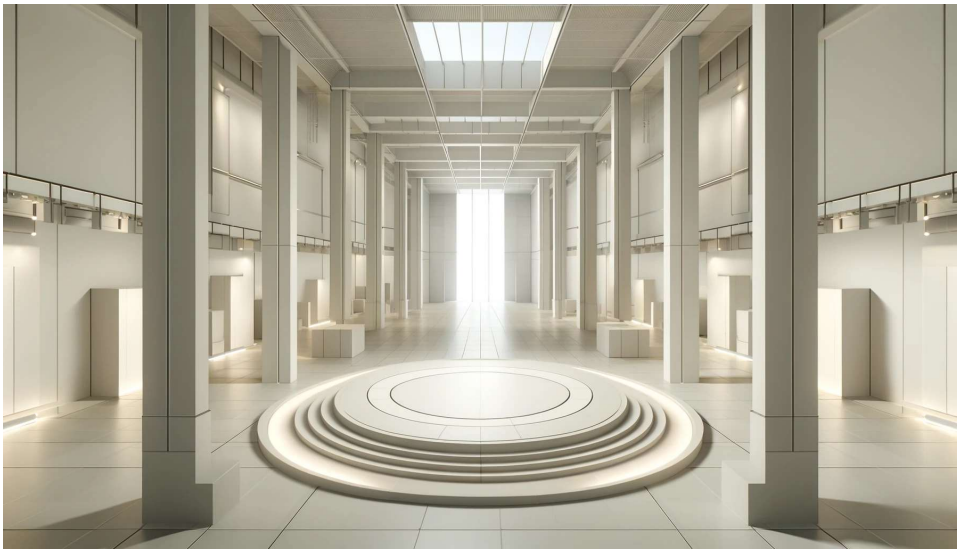
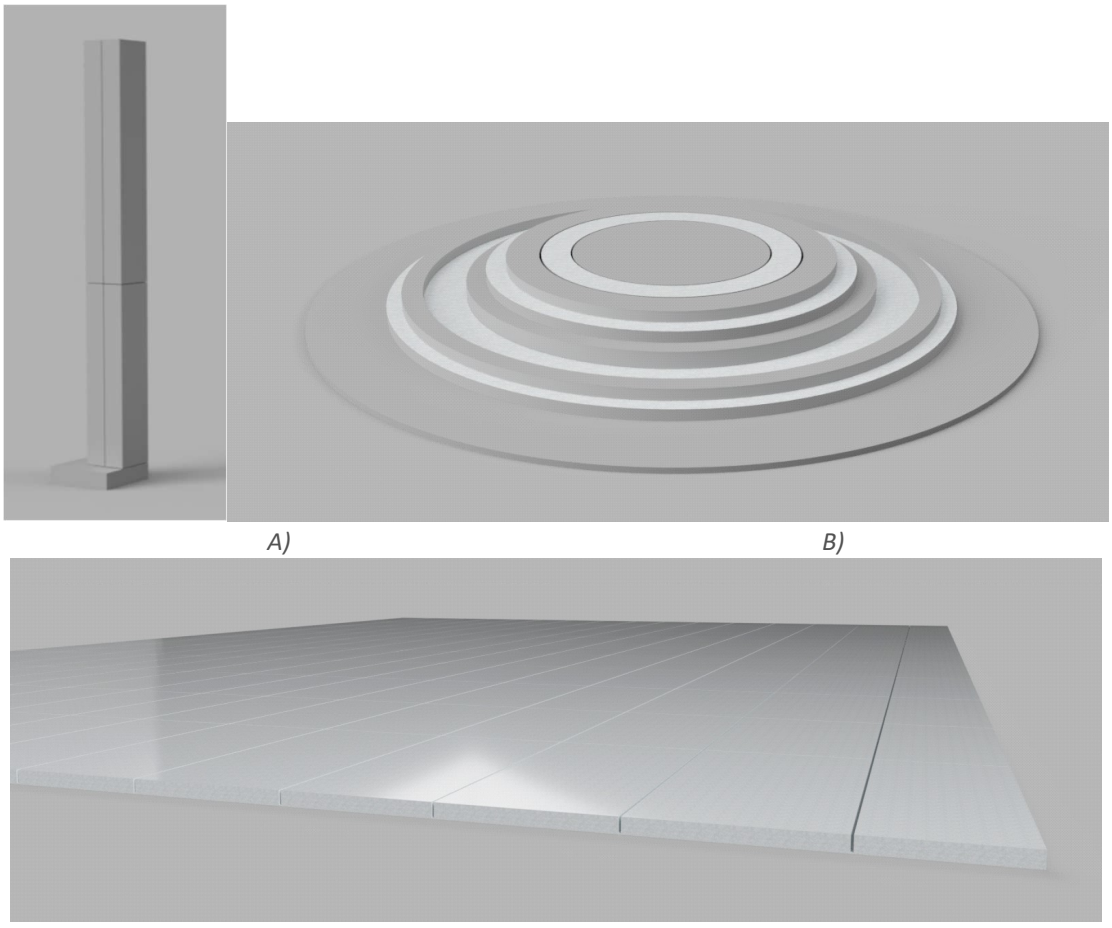


Ilustración 60: Inspiración para diseñar el escenario

La imagen de arriba fue la inspiración para el escenario. Se buscaba algo minimalista, pero al mismo tiempo colosal, ya que el usuario puede apreciar las dimensiones de los objetos dentro de la realidad virtual. Como este proyecto no tiene como objetivo en sí el diseño del escenario, se ha optado por realizar un escenario similar, pero menos complejo. Estos son los elementos que se han diseñado en Fusion 360:



A) *B)*

C)

Ilustración 61: Elementos diseñados en el escenario. A) Columna. B) Base. C) Suelo

También se han colocado los canvas con los nombres de las funciones, para que el usuario pueda utilizarlas:



Ilustración 62: Canvas con las distintas herramientas disponibles

De una manera muy visual, se le muestran al usuario imágenes de las formas primitivas que tiene la opción de crear, si dispara a dichas imágenes. La forma primitiva se creará justo enfrente, en el espacio designado para ello.

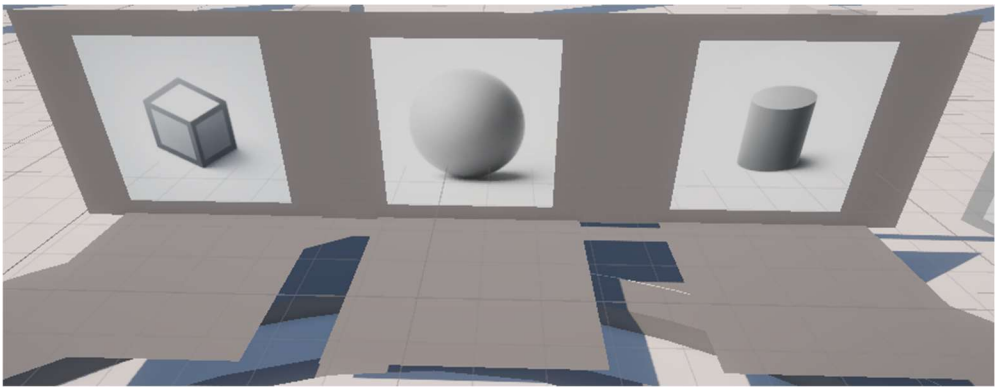


Ilustración 63: Canvas y bases para generar las formas primitivas

El display con la información de posición, rotación y dimensión de la pieza se encuentra a uno de los lados, para un fácil acceso. A su lado se encuentra también el teclado numérico.

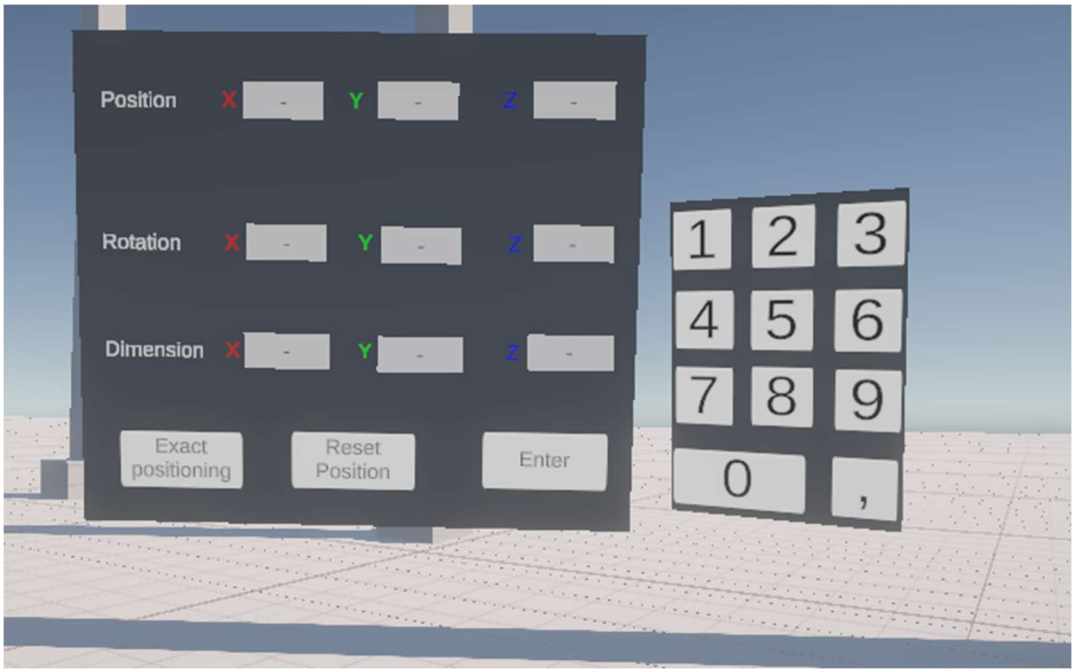


Ilustración 64: Teclado numérico y display con información de las piezas

Por último, se ha habilitado una zona para que el usuario pueda pegar imágenes del boceto de la pieza que quiere diseñar.

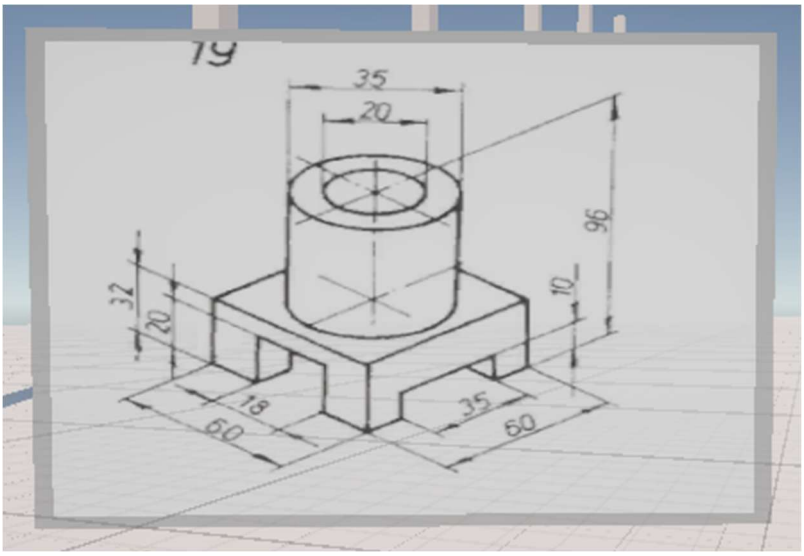


Ilustración 65: Imagen ejemplo de boceto de pieza para usar como referencia

Todo el escenario, quedaría de la siguiente forma:

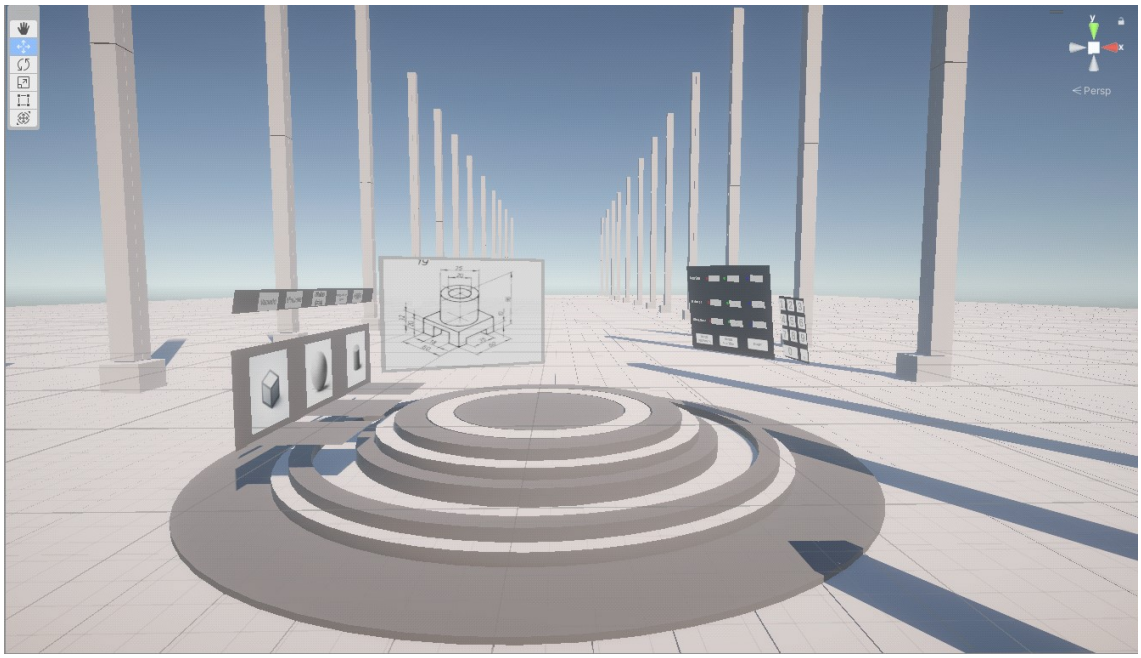


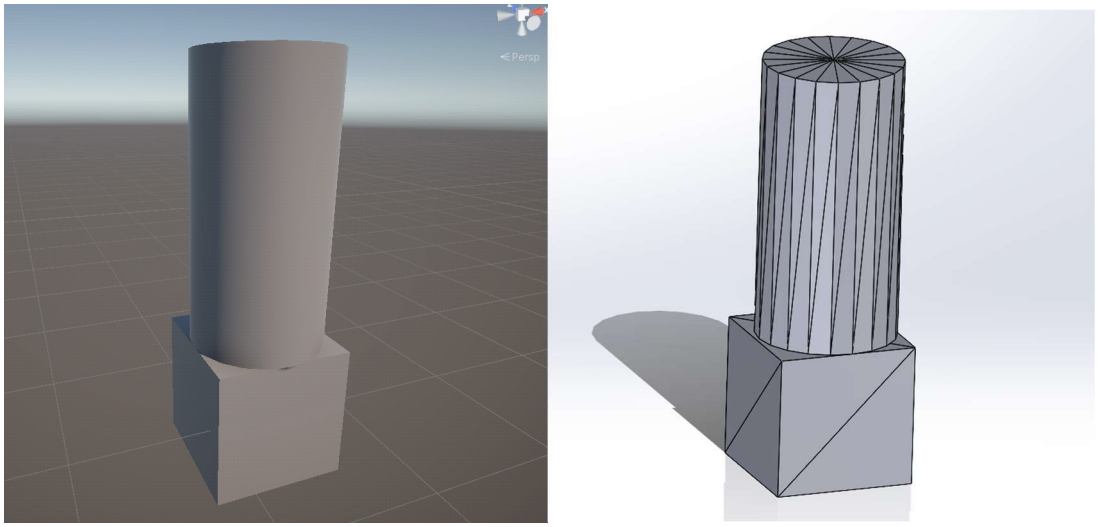
Ilustración 66: Visión general del escenario de diseño

4.24 EXPORTAR PIEZAS A SOLIDWORKS

Este proyecto tiene las funciones básicas para diseñar la mayor parte de piezas, sin embargo, programas como SolidWorks o NX tienen muchas más funciones. En algunos casos, puede ser

útil exportar las piezas diseñadas a este tipo de programas CAD para añadir los últimos detalles. En este proyecto, nos enfocamos en SolidWorks, pero se haría de manera similar en el resto.

Hay un problema fundamental al intentar exportar las piezas de Unity a SolidWorks. El problema viene de la naturaleza de los archivos. Unity trabaja con mallas, mientras que SolidWorks y el resto de programas CAD 3D con parámetros. En Unity (formato .fbx), se define un objeto 3D por su malla. En el caso de una esfera, la malla es un conjunto de triángulos que aproximan la forma de una esfera, nunca es exacto. En SolidWorks se define la esfera por un parámetro, por ejemplo, su radio. Si se intenta exportar directamente, la pieza queda de la siguiente manera:



A) B)

Ilustración 67: Pieza exportada a SolidWorks sin convertir primero a formato paramétrico. A) Pieza en Unity. B) Pieza en SolidWorks

El archivo de un objeto definido por mallas tiene menos información sobre el objeto, ya que las mallas son una aproximación. Es por ello que no hay un convertidor de archivos directo desde Unity a programas CAD 3D.

Sin embargo, he hallado una solución. Ello consiste en pasar las piezas por un programa intermedio: Fusion 360. Este programa tiene herramientas que interpretan la forma de la malla, y convierten el objeto a formato paramétrico. Posteriormente, se puede exportar a SolidWorks.

4.24.1 Implementación del programa intermediario Fusion 360

Para solucionar el problema explicado en (4.24), se requiere el uso del programa Fusion 360, como intermediario al exportar las piezas de Unity a programas CAD 3D.

El formato en el que se exportan las piezas de Unity es .fbx. Se recuerda que este formato trabaja con mallas.

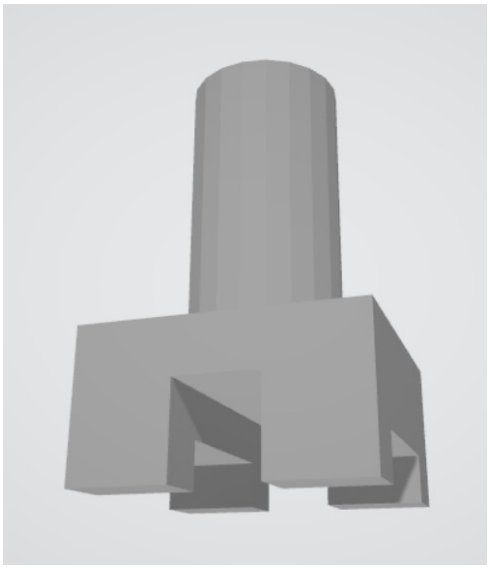


Ilustración 68: Pieza .fbx diseñada en Unity

El primer paso que se debe de realizar es limpiar la malla por si ésta tuviese imperfecciones. Si se revisa la Ilustración 39, la malla puede tener errores al hacer múltiples veces la operación de vaciado sobre la misma pieza. En gran parte de los casos se puede arreglar la malla con la herramienta "Reparar" de Fusion 360. Dentro de la función de reparar, se recomienda por regla general usar el tipo de reparación "Ajustar".

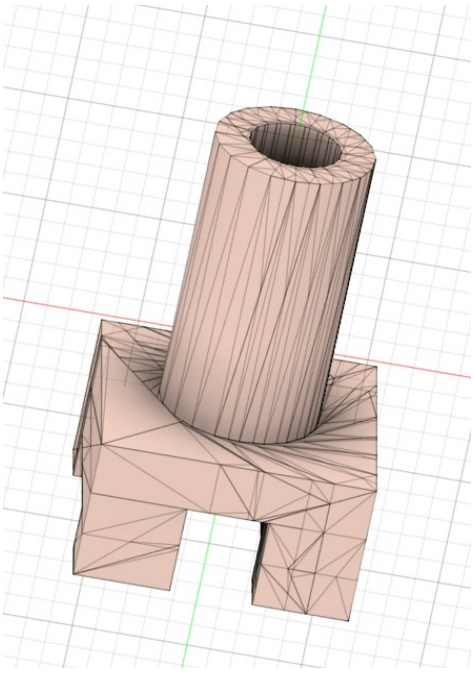


Ilustración 69: Pieza importada a Fusion 360

Una vez reparada la malla, se deben de especificar los grupos de caras de la pieza. Fusion 360 cuenta con una herramienta que genera automáticamente los grupos de caras. Se pueden ajustar parámetros como el umbral de ángulo para que las caras se generen correctamente, según la figura. La herramienta se llama "Generar grupos de cara".

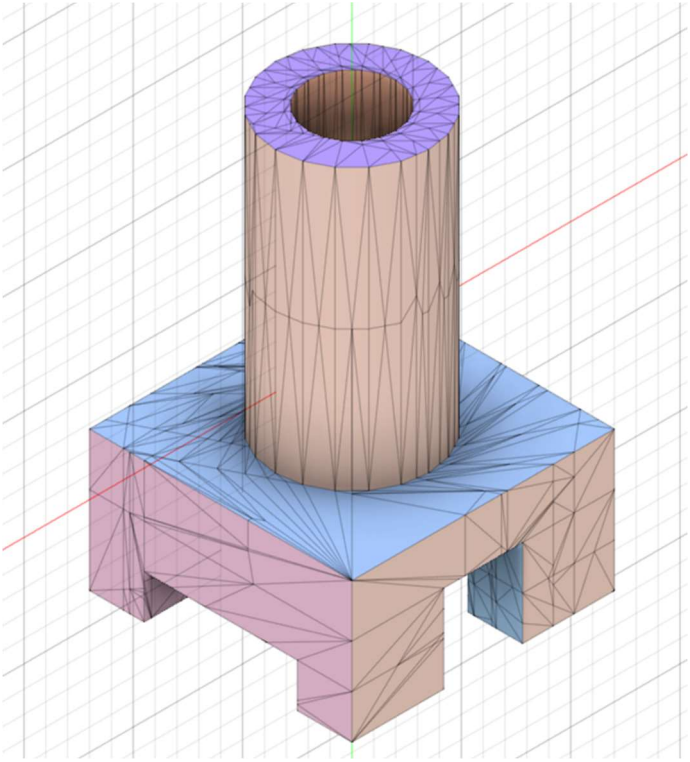


Ilustración 70: Pieza reparada y con los grupos de cara generados en Fusion 360

Con esto la pieza ya está preparada para convertirla a formato paramétrico. En Fusion 360, seguir los siguientes pasos:

Malla → Modificar → Convertir malla → Operación: Paramétrico → Aceptar

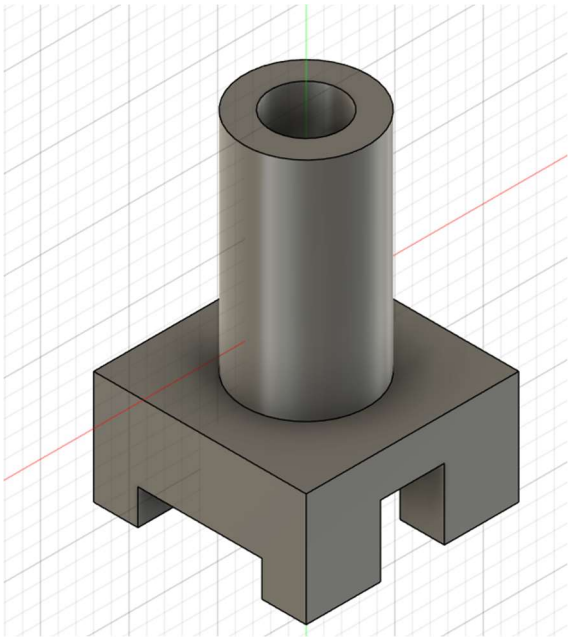


Ilustración 71: Pieza convertida a formato paramétrico en Fusion 360

Con ello, la pieza ya está definida como paramétrica. Para exportarla en SolidWorks, los tipos de archivo que mejor han funcionado son .dxf y .dwg.

4.24.2 Importar en SolidWorks

En SolidWorks importo la pieza .dxf o .dwg. Hay que seleccionar la opción Importar una nueva pieza como → Curvas o modelo 3D.

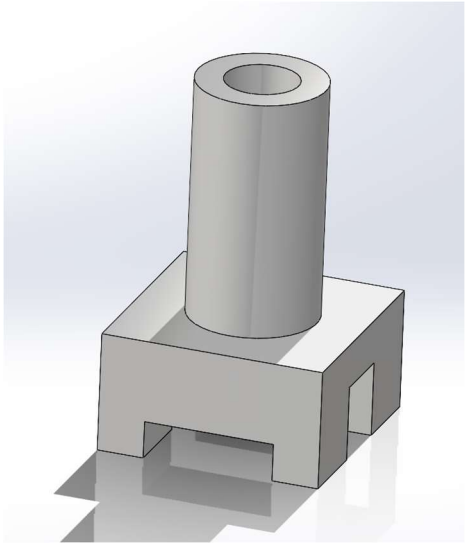


Ilustración 72: Pieza Importada en SolidWorks

5 ANÁLISIS DE RESULTADOS

El objetivo principal de este proyecto era la implementación de un programa de diseño industrial en un entorno de realidad virtual. El programa debía de proporcionar de una manera intuitiva y precisa, la posibilidad de diseñar la mayor variedad de piezas posibles. Bien se buscaba inspirarse en programas CAD 3D ya existentes como SolidWorks, nunca se planteó alcanzar tal complejidad. El programa debía de implementar las herramientas básicas y principales. En este apartado se va a valorar hasta qué punto se han conseguido cumplir los objetivos planteados desde una perspectiva crítica.

5.1 MAXIMIZAR LA CREACIÓN DE COMPONENTES

Combinando todas las herramientas y funciones implementadas se puede diseñar un gran número de piezas. Sin embargo, tiene ciertas limitaciones:

- **Redondeos**: no hay implementada ninguna herramienta que realice redondeos de manera intuitiva. Se es consciente de este hecho, por eso se ha habilitado la opción de exportar la pieza a SolidWorks. Se considera que los redondeos son "detalles", que no requieren por norma general un gran esfuerzo de visualización en pantallas 2D.

  Los usuarios que deseen añadir redondeos, deberán primero diseñar la pieza en el programa diseñado en este proyecto y posteriormente exportar la pieza a SolidWorks según se ha explicado (4.24). Por último, podrán añadir el detalle del redondeo en SolidWorks.

- **Elementos complejos como toroides o tornillos**: no se ha implementado ninguna herramienta que diseñe toroides o tornillos de una manera sencilla. Se ha dado prioridad a dar la posibilidad de diseñar un amplio espectro de piezas, en lugar de dedicar mucho tiempo a una herramienta muy específica. Como alternativa, se podría mejorar la función de creación de piezas primitivas (4.8) para que los usuarios importen sus propias piezas para que puedan trabajar con ellas como base.

Aunque no se ofrece la posibilidad de diseñar todo el abanico de piezas posibles, se puede diseñar una gran variedad. Como no se planteaba crear un programa tan completo como SolidWorks o similares, se ha dado la posibilidad de que los usuarios exporten sus piezas a dichos programas. De esta manera podrán terminar de detallar sus piezas.

5.2 AYUDA DE VISUALIZACIÓN 3D, HERRAMIENTAS INTUITIVAS

El objetivo de proporcionar una ayuda a la visualización de las piezas que presenten zonas complejas de visualizar en 3D queda satisfecho por la propia naturaleza del proyecto. Al ser un

proyecto en realidad virtual, se pueden apreciar las distancias y no hay errores de interpretación como puede pasar al visualizar 3D desde una pantalla.

Para enriquecer esta experiencia, se buscó siempre implementar herramientas que sean de uso intuitivo para los usuarios. Es por ello que se implementó el raycast. Su uso es similar al de un láser, lo que convierte la experiencia más familiar. Muchas herramientas, como la de trasladar con los ejes, se podían implementar de muchas otras maneras de manera más sencilla, pero siempre se intentó buscar la más intuitiva. También se puede apreciar la función de cambio de modo (4.12). Esta función permite cambiar entre trasladar, rotar y redimensionar simplemente pulsando un botón.

Aunque se buscaban herramientas intuitivas, el programa está orientado para la industria del diseño industrial, la que requiere de gran precisión y exactitud. Es por ello que se implementaron algunas funciones que son menos "naturales". Un ejemplo de ello es la de trasladar, mover y redimensionar de manera exacta (4.19), usando el teclado numérico.

5.3 PRECISIÓN Y EXACTITUD

Este requisito es importante para garantizar un correcto funcionamiento de las piezas diseñadas. Es por ello que se realizó un análisis para asegurar que las piezas mantienen sus dimensiones una vez convertidas a SolidWorks.

Las comprobaciones se realizaron con la figura de la Ilustración 72, cuyas medidas se pueden apreciar en el boceto de la Ilustración 73.

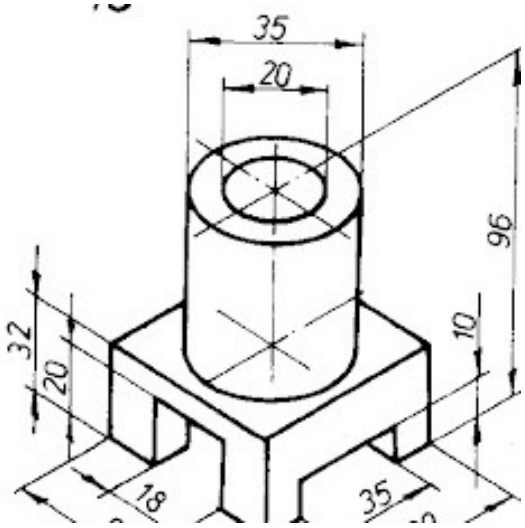


Ilustración 73: Boceto de la figura diseñada en la Ilustración 72

Tras importar la pieza en SolidWorks, he encontrado diferencias de las medidas originales en los dos diámetros de la pieza. El resto de medidas son exactamente iguales.

- **Medida diámetro exterior**: **34,9185mm**.
  - Tolerancia: $35mm - 34{,}9185mm = \mathbf{0{,}0815}\boldsymbol{mm}$

- $Tolerancia\ en\ porcentaje = \left(\frac{35mm - \ ,9185mm}{35mm}\right) \times 100 = \mathbf{0,233\%}$

- **Medida diámetro interior**: **19.8179mm**
  - Tolerancia: $20\ \text{mm} - 19{,}8179\ \text{mm} = \mathbf{0,1821\ mm}$
  - $Tolerancia\ en\ porcentaje = \left(\frac{20\ \text{mm} - 19{,}8179\ \text{mm}}{20\ \text{mm}}\right) \times 100 = \mathbf{0,911\%}$

$$Tolerancia\ en\ porcentaje = \left(\frac{Medida\ Teórica - Medida\ Real}{Medida\ Teórica}\right) \times 100 \quad \text{(Ecuación 1)}$$

Es un error relativamente pequeño, pero puede ser significativo para aplicaciones donde la tolerancia de error pueda ser menor. Sin embargo, existe una solución para reducir este error.

La causa del error es que al especificar el diámetro en Unity, estamos especificando la máxima distancia entre los 2 puntos de la circunferencia. Hay que recordar que no es una circunferencia exacta, ya que Unity trabaja con mallas.

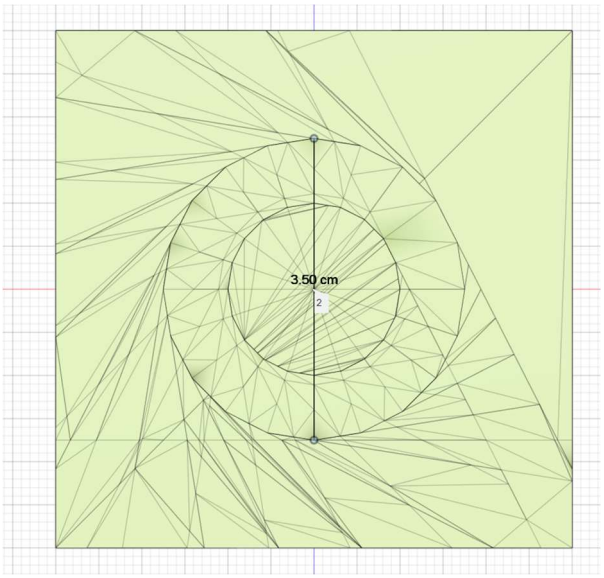


Ilustración 74: Medición de la figura en Fusion 360, vista desde arriba

Como se puede observar en la Ilustración 74, el diámetro de la figura mide correctamente en los extremos de su malla, pero si se eligen otros puntos intermedios, el resultado varía ligeramente. Se puede observar claramente que el diámetro no es una circunferencia.

Cuando se convierte la pieza de mallas a forma paramétrica, Fusion 360 interpreta la forma de cilindro y hace una media para calcular el diámetro. Es por ello que la medición final cambia ligeramente.

La solución a este problema es tener mallas más detalladas y complejas en Unity. De esta manera, las mediciones finales se aproximarán mucho más a las deseadas. Esto conlleva un modelo más complejo, que requiere de mayor potencia de procesamiento y tarjeta gráfica para diseñarlo y procesarlo.

5.4 OTROS FALLOS

Además del error descrito en el apartado 5.3, se recuerda el bug en la malla que se producía al ejecutar varias veces la función de vaciado sobre la misma pieza (10.13). Ello añade una limitación al diseñar. El número de veces que se puede usar la función de vaciado sobre la misma pieza varía según la complejidad de las piezas y sus mallas. El rango está de manera aproximada entre 3 y 8. Este error es debido al complejo cálculo de las mallas al realizar la operación.

La parte positiva es que el programa de Fusion 360 tiene herramientas para arreglar las mallas y la soluciona en gran parte de los casos (4.15.2).

5.5 PRUEBAS Y FEEDBACK

El usuario que ha probado este programa y ha hecho la pieza de la Ilustración 73 es Manuel Bodega Palomeque, ingeniero industrial. Después de un breve proceso de familiarización con las herramientas, tardó aproximadamente 5 minutos para diseñar la pieza.

Entre su feedback se destacan los siguientes puntos:

- Herramienta muy intuitiva y fácil de usar. Va todo muy fluido y funciona bien.
- El usuario echa de menos la opción de volver atrás o “control + z”.

Ha asegurado que este programa es más fácil de usar que los tradicionales como “SolidWorks”. Esto puede ser en parte a su simpleza, pero también a su intuitividad.

Se deja la implementación de la herramienta para volver atrás y adelante para futuras mejoras del programa.

5.6 REFLEXIONES FINALES Y FUTURAS IMPLEMENTACIONES

El programa funciona de manera general como se esperaba en los objetivos. Sin embargo, se han dado algunos problemas que han hecho replantearse las bases del proyecto. Las mallas son un elemento muy complejo de manejar. En este proyecto se han intentado atenuar algunos problemas con programas como Fusion 360, pero no se piensa que esta sea una solución viable si se piensa elevar este proyecto al nivel profesional.

Una de las posibles opciones para futuras implementaciones sería diseñar de manera paramétrica directamente, en lugar de trabajar con mallas. De esta manera, las piezas estarían definidas de manera exacta, por ejemplo, una esfera se define por su radio. Queda como estudio

de investigación explorar las vías para diseñar de manera paramétrica en un entorno de realidad virtual. Hay que tener en cuenta que Unity trabaja con mallas.

Además, se espera que en futuras versiones este proyecto implemente un mayor número herramientas, para igualar las posibilidades que ofrecen programas CAD 3D profesionales. De esta manera, el diseño en realidad virtual se podría convertir en el nuevo método preferido por los diseñadores.

6 Presupuesto

En este apartado se va a hacer un análisis de los costos de este proyecto o similares. Como herramienta fundamental de trabajo, se necesita un dispositivo de realidad virtual. En este caso se han elegido las Meta Quest 2 por su gran versatilidad y compatibilidad con los programas. Este dispositivo de RV se puede encontrar por un valor de entre 250€ - 350€ [10].

Para poder ejecutar una aplicación de realidad virtual, se necesita un ordenador con una tarjeta gráfica equivalente o superior a la a NVIDIA GeForce RTX 20-series. Se recomienda usar procesador Intel i7 o AMD Ryzen 7 o superior, y 8GB de RAM o más [30]. El valor de un ordenador con estas características puede variar entre 1200€ - 3000€.

Los programas Unity, Fusion 360 y SolidWorks cuentan con licencia gratuita para estudiantes, por lo que no suponen un costo adicional.

En resumen, el costo se reduce al equipo de realidad virtual y al ordenador personal. Haciendo una media de los valores aproximados:

Equipo	Cantidad (uds.)	Coste unitario (€/ud.)	Coste total (€)
Ordenador personal compatible	1	2100€	2100€
Dispositivo de realidad virtual	1	300€	300€
		Total:	**2400€**

7 ALINEAMIENTO CON LOS OBJETIVOS DE DESARROLLO SOSTENIBLE (ODS)

4.- Educación de Calidad: este programa también podría ser utilizado en el ámbito de la educación. Visualizar las piezas en un entorno 3D es mucho más intuitivo, asemejándose más a la vida real. También podría ayudar como suplemento en la educación tradicional de programas CAD enseñados en un ordenador.

8.- Trabajo decente y crecimiento económico: Aunque este programa es una base, se demuestra que esta metodología de trabajo inmersivo tiene mucho potencial para ser aplicado en las industrias en el futuro. Si se desarrolla un programa profesional y completo, similar a los programas CAD 3D, se agilizarán los procesos y se obtendrá mejores resultados de los productos diseñados. Todo ello conllevará a un mayor crecimiento económico, teniendo más alternativas para el mismo proceso.

9.- Industria, innovación e infraestructura: El proyecto contribuye a la innovación en la industria del diseño en 3D, al ofrecerse como una alternativa a los métodos utilizados hasta la fecha.

8 CONCLUSIONES Y RECOMENDACIONES

Este proyecto se posiciona como un referente en el del uso de la realidad virtual en el sector del diseño industrial. En él se facilita un entorno inmersivo en el que los usuarios pueden diseñar directamente en 3D para así eliminar las limitaciones de las pantallas 2D. Como herramientas para implantar el programa de realidad virtual se ha hecho uso de la aplicación Unity, así como de las gafas Meta Quest 2.

El objetivo del proyecto es la creación de un programa de RV con las herramientas básicas para que el usuario pueda diseñar piezas de ingeniería en un entorno 3D. De esta manera el proceso de diseño es mucho más intuitivo, con un proceso de creación más natural. Además, se proporciona un método para exportar los modelos diseñados desde Unity a programas CAD más avanzados como SolidWorks.

El proyecto supuso algunos desafíos, la mayor parte de ellos relacionados con el trabajo de mallas tridimensionales. Esto provocó errores en la herramienta de vaciado cuando se usa repetidamente sobre la misma pieza. También puede suponer una pequeña variación en las dimensiones finales del modelo cuando la malla inicial es poco compleja. Estas dificultades sugieren otras maneras de aproximarse al proyecto en futuras implementaciones, como trabajar directamente con formatos paramétricos.

Con este proyecto, se da el primer paso en la implementación de la realidad virtual en el diseño industrial. Con el incremento de la accesibilidad de las tecnologías RV, se espera que más proyectos como este impulsen la transición del diseño industrial a la realidad virtual.

9 BIBLIOGRAFÍA

[1] Unity, «Unity, main page,» 2024. [En línea]. Available: https://unity.com/es. [Último acceso: 15 07 2024].

[2] Meta, «Meta Quest 2,» 2024. [En línea]. Available: https://www.meta.com/es/quest/products/quest-2/. [Último acceso: 15 07 2024].

[3] Autodesk, «Autodesk AutoCAD,» 2024. [En línea]. Available: https://www.autodesk.com/es/products/autocad/overview?term=1-YEAR&tab=subscription. [Último acceso: 15 07 2024].

[4] SolidWorks, «SolidWorks,» 2024. [En línea]. Available: https://www.solidworks.com/es. [Último acceso: 15 07 2024].

[5] Siemens, «Software NX,» 2024. [En línea]. Available: https://plm.sw.siemens.com/es-ES/nx/. [Último acceso: 15 07 2024].

[6] The Business Research Company, Virtual Realitiy Software Global Market Report 2024, 2024.

[7] GravitySketch, «Gravity Sketch,» 2024. [En línea]. Available: https://www.gravitysketch.com/. [Último acceso: 15 07 2024].

[8] Logitech, «Unleash Creativity in True 3D, with Logitech VR Ink Pilot Edition,» Logitech, 21 04 2022. [En línea]. Available: https://youtu.be/MduBtmijXZc?si=O3aeAjT5VSYxuQ92. [Último acceso: 2024 07 15].

[9] Logitech, «Introducing MX Ink,» 2024. [En línea]. Available: https://www.logitech.com/en-us/products/vr/mx-ink.html. [Último acceso: 15 07 2024].

[10] Meta, «Meta Quest 2,» 2024. [En línea]. Available: https://www.meta.com/es/quest/products/quest-2/. [Último acceso: 15 07 2024].

[11] McKinsey analysis, «https://www.mckinsey.com/spContent/bespoke/tech-trends/pdfs/mckinsey-tech-trends-outlook-2022-immersive-reality.pdf,» 2022.

[12] RobotStudio, 2024. [En línea]. Available: https://new.abb.com/products/robotics/es/robotstudio. [Último acceso: 15 07 2024].

[13] Shapelab, 2024. [En línea]. Available: https://shapelabvr.com/. [Último acceso: 15 07 2024].

[14] Quill, 2024. [En línea]. Available: https://www.quill.org/. [Último acceso: 15 07 2024].

Bibliografía

[15] Blender, 2024. [En línea]. Available: https://www.blender.org/. [Último acceso: 15 07 2024].

[16] Archimatix, «Archimatix, Node-based ParametricModeling in Unity,» Archimatix, 2024. [En línea]. Available: https://www.archimatix.com/. [Último acceso: 15 07 2024].

[17] Unreal Engine, 2024. [En línea]. Available: https://www.unrealengine.com/es-ES. [Último acceso: 15 07 2024].

[18] Udemy, 2024. [En línea]. Available: https://www.udemy.com/. [Último acceso: 15 07 2024].

[19] D. Stringer, «Learn Unity Games Engine & C# By Creating A VR Space Shooter,» Udemy, 2024. [En línea]. Available: https://www.udemy.com/course/unityspaceshooter/?couponCode=ST18MT62524. [Último acceso: 15 07 2024].

[20] Visual Studio, 2024. [En línea]. Available: https://visualstudio.microsoft.com/es/. [Último acceso: 15 07 2024].

[21] LuisCanary, «Unity Solucion AUTOCOMPLETAR en Visual Studio 2024/23/22/21... ✓,» Youtube, 21 04 2022. [En línea]. Available: https://youtu.be/O23-mcVhOeY?si=VpByKssd_AHOLz6j. [Último acceso: 15 07 2024].

[22] Notion, 2024. [En línea]. Available: https://www.notion.so/. [Último acceso: 07 07 2024].

[23] M. Quest, «Meta Quest Developer Center,» [En línea]. Available: https://developer.oculus.com/. [Último acceso: 15 07 2024].

[24] Andrew, «Custom Input Actions for Unity XR,» Youtube, 21 04 2021. [En línea]. Available: https://youtu.be/jOn0YWoNFVY?si=rp8rCbief6axjHIP. [Último acceso: 2024 07 15].

[25] DocFx, «Enum InputActionType,» Unity, 18 10 2023. [En línea]. Available: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/api/UnityEngine.InputSystem.InputActionType.html. [Último acceso: 2024 07 10].

[26] DocFX, «Namespace UnityEngine.InputSystem.Controls,» Unity, 18 10 2023. [En línea]. Available: https://docs.unity3d.com/Packages/com.unity.inputsystem@0.9/api/UnityEngine.InputSystem.Controls.html. [Último acceso: 2024 07 15].

[27] berkhulagu, «Change Rendering Mode via Script,» Unity, 2023. [En línea]. Available: https://forum.unity.com/threads/change-rendering-mode-via-script.476437/. [Último acceso: 15 07 2024].

[28] E. Wallace, «csg.js,» Github, 2011. [En línea]. Available: https://github.com/evanw/csg.js/. [Último acceso: 15 07 2024].

Bibliografía

[29] Unity, «Boolean Subtraction Operations on Mesh,» Unity, 2023. [En línea]. Available: https://forum.unity.com/threads/boolean-subtraction-operations-on-mesh.85713/. [Último acceso: 15 07 2024].

[30] Meta, «Requisitos para usar Meta Quest Link,» Meta, 2024. [En línea]. Available: https://www.meta.com/es-es/help/quest/articles/headsets-and-accessories/oculus-link/requirements-quest-link/. [Último acceso: 15 07 2024].

[31] Autodesk, «Autodesk Fusion,» 2024. [En línea]. Available: https://www.autodesk.com/es/products/fusion-360/overview?term=1-YEAR&tab=subscription. [Último acceso: 15 07 2024].

10 ANEXOS

10.1 EVENTHUB

```
using UnityEngine;

public static class EventHub
{
    //Definición de los distintos delegados d elos eventos (en ellos se especifica el tipo de argumentos)
    public delegate void RaycastEvent(RaycastHit hitInfo); //Se define el delegado
    public delegate void RaycastEvent2(); //En este no hace falta pasar argumento
      public delegate void CotasEvent(Transform hitTransform, Canvas hitCanvas, bool firstCreation);
    public delegate void CanvasAnimator(Canvas canvas);

    public static event RaycastEvent OnRaycastHit;
    public delegate void ModeChangeEvent();
    public static event ModeChangeEvent OnModeChange;
    public delegate void TriggerAction();
    public static event TriggerAction OnTriggerPressed;
    public static event TriggerAction OnTriggerReleased;
    public static event RaycastEvent OnShapeGeneratorHit;
    public static event RaycastEvent OnEliminateObjectHit;
    public static event RaycastEvent OnVaciadoHit;
    public static event RaycastEvent OnActivateObject;
    public static event RaycastEvent OnActivateOnlyObject;
    public static event RaycastEvent OnSelectedObjects;
    public static event RaycastEvent OnGroupObjectsCanvas;
    public static event RaycastEvent OnSelectedGroupObjects;
    public static event RaycastEvent OnSpecialPointsCanvas;
    public static event RaycastEvent OnSelectedSpecialPoints;
    public static event RaycastEvent OnCotasCanvas;
    public static event RaycastEvent OnPositionCanvas;
    public static event RaycastEvent OnGetObjectPosition;
    public static event RaycastEvent OnDuplicateObjectsHit;
    public static event RaycastEvent OnSaveObjectsHit;
    public static event RaycastEvent2 OnResetActivation;
    public static event RaycastEvent2 OnActivateOnlyObjectAll;
    public static event CotasEvent OnActualizarCotas;
    public static event CanvasAnimator OnCanvasAnimation;
```

```
    // Atributo que puede tener el valor de los 3 modos
    public enum ModoInteraccion { Mover, Rotar, Extruir }
     //Permite acceder públicamente al argumento de forma segura. Para
establecerlo, solo se puede de forma privada desde este script
     public static ModoInteraccion CurrentMode { get; private set; } =
ModoInteraccion.Mover;

    //Se le asigna cada evento para que dispare diferentes banderas
       public static void TriggerRaycastHit(RaycastHit hitInfo) =>
OnRaycastHit?.Invoke(hitInfo);
       public static void ShapeGeneratorHit(RaycastHit hitInfo) =>
OnShapeGeneratorHit?.Invoke(hitInfo);
       public static void EliminateObjectHit(RaycastHit hitInfo) =>
OnEliminateObjectHit?.Invoke(hitInfo);
         public static void VaciadoHit(RaycastHit hitInfo) =>
OnVaciadoHit?.Invoke(hitInfo);
       public static void SelectedObjects(RaycastHit hitInfo) =>
OnSelectedObjects?.Invoke(hitInfo);
    public static void TriggerPressed() => OnTriggerPressed?.Invoke();
    public static void TriggerReleased() => OnTriggerReleased?.Invoke();
       public static void ActivateObjectHit(RaycastHit hitInfo) =>
OnActivateObject?.Invoke(hitInfo);
      public static void ActivateOnlyObjectHit(RaycastHit hitInfo) =>
OnActivateOnlyObject?.Invoke(hitInfo);
       public static void GroupObjectsCanvas(RaycastHit hitInfo) =>
OnGroupObjectsCanvas?.Invoke(hitInfo);
      public static void SelectedGroupObjects(RaycastHit hitInfo) =>
OnSelectedGroupObjects?.Invoke(hitInfo);
       public static void SpecialPointsCanvas(RaycastHit hitInfo) =>
OnSpecialPointsCanvas?.Invoke(hitInfo);
      public static void SelectedSpecialPoints(RaycastHit hitInfo) =>
OnSelectedSpecialPoints?.Invoke(hitInfo);
         public static void CotasCanvas(RaycastHit hitInfo) =>
OnCotasCanvas?.Invoke(hitInfo);
       public static void PositionCanvas(RaycastHit hitInfo) =>
OnPositionCanvas?.Invoke(hitInfo);
       public static void GetObjectPosition(RaycastHit hitInfo) =>
OnGetObjectPosition?.Invoke(hitInfo);
      public static void DuplicateObjectsHit(RaycastHit hitInfo) =>
OnDuplicateObjectsHit?.Invoke(hitInfo);
       public static void SaveObjectsHit(RaycastHit hitInfo) =>
OnSaveObjectsHit?.Invoke(hitInfo);
    public static void ResetActivation() => OnResetActivation?.Invoke();
            public static void ActivateOnlyObjectAll() =>
OnActivateOnlyObjectAll?.Invoke();
```

```
    public static void ActualizarCotas(Transform hitTransform, Canvas
hitCanvas, bool firstCreation) => OnActualizarCotas?.Invoke(hitTransform,
hitCanvas, firstCreation);
        public static void CanvasAnimation(Canvas canvas) =>
OnCanvasAnimation?.Invoke(canvas);


    public static void TriggerModeChange()
    {
        // Método que se encarga de ciclar entre los 3 modos
        // Se cambia de modo cada vez que se pulsa el botón A
        // Cada vez que se cambia de modo se llama al modo controlador
        switch (CurrentMode)
        {
            case ModoInteraccion.Mover:
                CurrentMode = ModoInteraccion.Rotar;
                break;
            case ModoInteraccion.Rotar:
                CurrentMode = ModoInteraccion.Extruir;
                break;
            case ModoInteraccion.Extruir:
                CurrentMode = ModoInteraccion.Mover;
                break;
            default:
                CurrentMode = ModoInteraccion.Mover;
                break;
        }
        //Debug.Log("El modo es: " + CurrentMode);
        OnModeChange?.Invoke();
    }
}
```

10.2 TRIGGERSCRIPT

```
using UnityEngine;
using UnityEngine.InputSystem;

public class TriggerScript : MonoBehaviour
{
    public InputActionReference triggerActionReference; //Lectura del
gatillo
    public InputActionReference sButtonActionReference; //Lectura del botón
```

```
    private bool isTriggerPressed = false;                //Variable de
lectura para confirmar si el gatillo está pulsado


    private void OnEnable()
    {
        triggerActionReference.action.started += OnTriggerPressed;
        triggerActionReference.action.canceled += OnTriggerReleased;
        triggerActionReference.action.Enable();

        sButtonActionReference.action.started += OnButtonPressed;
        sButtonActionReference.action.Enable();
    }

    private void OnDisable()
    {
        triggerActionReference.action.started -= OnTriggerPressed;
        triggerActionReference.action.canceled -= OnTriggerReleased;
        triggerActionReference.action.Disable();

        sButtonActionReference.action.started -= OnButtonPressed;
        sButtonActionReference.action.Disable();
    }

    private void OnTriggerPressed(InputAction.CallbackContext context)
    {
        isTriggerPressed = true;
        EventHub.TriggerPressed(); //Dispara el evento de pulsar gatillo
    }

    private void OnTriggerReleased(InputAction.CallbackContext context)
    {
        isTriggerPressed = false;
        EventHub.TriggerReleased(); //Dispara el evento de soltar gatillo
    }

    private void OnButtonPressed(InputAction.CallbackContext context)
    {
         EventHub.TriggerModeChange(); //Dispara el evento de pulsar el
botón y cambia de modo
    }
}
```

10.3 RAYCASTSCRIPT

```
using System.Linq;
using UnityEditor.SearchService;
using UnityEngine;

public class Raycast : MonoBehaviour
{
     [SerializeField] private Transform raycastOrigin; //Objeto con la posición y dirección del raycast

    void OnEnable()
    {
        EventHub.OnTriggerPressed += HandleTriggerPressed;  //Se suscribe al evento de cuando se presiona el gatillo
    }

    void OnDisable()
    {
        EventHub.OnTriggerPressed -= HandleTriggerPressed;
    }

    private void HandleTriggerPressed()
    {
        //Guardo todos los objetos en los que ha impactado el raycast
          RaycastHit[] hits = Physics.RaycastAll(raycastOrigin.position, raycastOrigin.forward, 800); //Guarda en hits todos los objetos que han sido golpeados por el raycast
        //Ordena por distancia para ejecutar el primero que encuentre
        var sortedHits = hits.OrderBy(hit => hit.distance).ToArray();
        bool processedPriorityObject = false;
        Vaciado vaciado = FindObjectOfType<Vaciado>();
        GroupCreation group = FindObjectOfType<GroupCreation>();
                            MovimientoPorPuntos specialPoints = FindObjectOfType<MovimientoPorPuntos>();
        Keyboard keyboard = FindObjectOfType<Keyboard>();

        //Si no choca con nada, desactiva los objetos (tiene en cuenta de que si está en un modo especial no desactiva los objetos)
         if (hits.Length == 0 && !vaciado.isVaciado && !group.isGrouping && !specialPoints.isActive && !keyboard.exactPositioningActive)
        {
            EventHub.ResetActivation();
        }
        //Revisa si ha impactado en algún objeto prioritario
```

```
        foreach (RaycastHit hit in sortedHits)
        {
            // Busca por CubeMovement y procesa el primero que encuentra
                                    CubeMovement     hitCubeMovement     =
hit.collider.GetComponent<CubeMovement>();
            if (hitCubeMovement != null)
            {
                EventHub.TriggerRaycastHit(hit);
                processedPriorityObject = true;
                break; // Detiene el bucle después de procesar este objeto
            }

            // Busca por CubeRotation y procesa el primero que encuentra
                                    CubeRotation     hitCubeRotation     =
hit.collider.GetComponent<CubeRotation>();
            if (hitCubeRotation != null)
            {
                EventHub.TriggerRaycastHit(hit);
                processedPriorityObject = true;
                break; // Detiene el bucle después de procesar este objeto
            }

           //Busca por ExtrusionSpheres y procesa el primero que encuentra
                                ExtrusionMovement    hitExtrusionSpheres    =
hit.collider.GetComponent<ExtrusionMovement>();
            if (hitExtrusionSpheres != null)
            {
                EventHub.TriggerRaycastHit(hit);
                processedPriorityObject = true;
                break; // Detiene el bucle después de procesar este objeto
            }
           //Detecta si ha impactado en el canvas de movimiento por puntos
                                 MovimientoPorPuntos    hitMovimientopp    =
hit.collider.GetComponent<MovimientoPorPuntos>();
            if (hitMovimientopp != null)
            {
                 EventHub.CanvasAnimation(hit.collider.GetComponent<Canvas
>());
                EventHub.SpecialPointsCanvas(hit);
                processedPriorityObject = true;
                break;
            }
            //Detecta si ha impactado en un punto característico
                                    if     (specialPoints.isActive     &&
hit.transform.CompareTag("SpecialPoint"))
            {
```

```
                EventHub.SelectedSpecialPoints(hit);
                processedPriorityObject = true;
                break; // Detiene el bucle despu□s de procesar este objeto
            }
             //Lee el valor que el usuario quiere cambiar en la "función
exacta" (la función del teclado)
           if (keyboard.exactPositioningActive && keyboard.objectSelected)
            {
                if (hit.collider.gameObject.name == "PosX_B" ||
                hit.collider.gameObject.name == "PosY_B" ||
                hit.collider.gameObject.name == "PosZ_B" ||
                hit.collider.gameObject.name == "RotX_B" ||
                hit.collider.gameObject.name == "RotY_B" ||
                hit.collider.gameObject.name == "RotZ_B" ||
                hit.collider.gameObject.name == "DimX_B" ||
                hit.collider.gameObject.name == "DimY_B" ||
                hit.collider.gameObject.name == "DimZ_B")
                {
                    EventHub.PositionCanvas(hit);
                    break;
                }
            }
        }

        // Si no se ha procesado ningún objeto prioritario, entonces busca
por PrimaryObject
        if (!processedPriorityObject && !specialPoints.isActive)
        {
            foreach (RaycastHit hit in sortedHits)
            {
                            PrimitiveShapeGenerator   hitShapeGenerator   =
hit.collider.GetComponent<PrimitiveShapeGenerator>();
                                EliminateObject   hitEliminateObject   =
hit.collider.GetComponent<EliminateObject>();
                Vaciado hitVaciado = hit.collider.GetComponent<Vaciado>();
                                        GroupCreation    hitgroup    =
hit.collider.GetComponent<GroupCreation>();
                                    CotasCanvas    hitCotasCanvas    =
hit.collider.GetComponent<CotasCanvas>();
                                DuplicateObjects   hitDuplicateObjects   =
hit.collider.GetComponent<DuplicateObjects>();
                                      SaveObject    hitSaveObject    =
hit.collider.GetComponent<SaveObject>();

                //Si el objeto golpeado es una pieza, tiene que buscar qué
modo está activo
```

```
                //En función de la función que esté activa disparará unos
eventos u otros
                if (hit.collider.gameObject.tag == "PrimaryObject")
                {
                    //Si el modo vaciado está activo
                    if (vaciado.isVaciado)
                    {
                            EventHub.SelectedObjects(hit);    //Sirve para
seleccionar 2 objetos en el modo de vaciado
                        EventHub.ActivateOnlyObjectHit(hit);
                    }
                    //Si el modo grupo está activo
                    else if (group.isGrouping)
                    {
                        EventHub.SelectedGroupObjects(hit);
                        EventHub.ActivateOnlyObjectHit(hit);
                    }
                    //Si la función exacta está activa
                    else if (keyboard.exactPositioningActive)
                    {
                        if (!keyboard.objectSelected)
                        {
                            EventHub.GetObjectPosition(hit);
                            EventHub.ActivateOnlyObjectHit(hit);
                        }
                    }
                      //Si no hay ninguna herramienta activa, simplemente
activa la pieza y genera los ejes que correspondan
                    else
                    {
                         EventHub.ResetActivation(); //Desactiva todas las
piezas
                       EventHub.ActivateObjectHit(hit); //Activa la pieza
impactada por el raycast
                    }
                     break; // Salir del bucle si se cumple alguna de las
condiciones
                }
                //Si golpea en el canvas de la generación de la pieza
                else if (hitShapeGenerator != null)
                {
                    EventHub.ShapeGeneratorHit(hit);
                    break; // Salir del bucle si se cumple esta condición
                }
                //Si golpea en el canvas de eliminar objetos
                else if (hitEliminateObject != null)
```

```
                {
                    EventHub.CanvasAnimation(hit.collider.GetComponent<Ca
nvas>());
                    EventHub.EliminateObjectHit(hit);
                    break; // Salir del bucle si se cumple esta condición
                }
                //Si golpea en el canvas de duplicar objetos
                else if (hitDuplicateObjects != null)
                {
                    EventHub.CanvasAnimation(hit.collider.GetComponent<Ca
nvas>());
                    EventHub.DuplicateObjectsHit(hit);
                    EventHub.ResetActivation();
                    break; // Salir del bucle si se cumple esta condición
                }
                //Si golpea en el canvas de guardar / exportar objetos
                else if (hitSaveObject != null)
                {
                    EventHub.CanvasAnimation(hit.collider.GetComponent<Ca
nvas>());
                    EventHub.SaveObjectsHit(hit);
                    break; // Salir del bucle si se cumple esta condición
                }
                //Si golpea en el canvas de vaciado
                else if (hitVaciado != null)
                {
                    EventHub.CanvasAnimation(hit.collider.GetComponent<Ca
nvas>());
                    EventHub.VaciadoHit(hit);
                    break; // Salir del bucle si se cumple esta condición
                }
                //Si golpea en el canvas de hacer grupo
                else if (hitgroup != null)
                {
                    EventHub.CanvasAnimation(hit.collider.GetComponent<Ca
nvas>());
                    EventHub.GroupObjectsCanvas(hit);
                    break; // Salir del bucle si se cumple esta condición
                }
                //Si golpea en el canvas de activar las cotas
                else if (hitCotasCanvas != null)
                {
                    EventHub.CanvasAnimation(hit.collider.GetComponent<Ca
nvas>());
                    EventHub.CotasCanvas(hit);
                    break; // Salir del bucle si se cumple esta condición
```

```
                }
            }

        }
    }
}
```

10.4 PRIMITIVESHAPEGENERATOR

```
using Sabresaurus.SabreCSG;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
//using UnityEngine.ProBuilder;
//using UnityEngine.ProBuilder.MeshOperations;


public class PrimitiveShapeGenerator : MonoBehaviour
{
    public GameObject toroidePrefab;        // Ejes de rotación
    public GameObject grabAxisXPrefab;      // Ejes de traslación
    public GameObject extrusionSphere;      // Ejes de dimensionado
    public CreateSpecialPoints specialPointsScript; //Puntos significativos
para la función de movimiento por puntos
    public Material transparentMaterial; //Material ligeramente modificado
para poder cambiarle la transparencia


    void OnEnable()
    {
        EventHub.OnShapeGeneratorHit += HandleCanvasHit;
    }

    void OnDisable()
    {
        EventHub.OnShapeGeneratorHit -= HandleCanvasHit;
    }

    void HandleCanvasHit(RaycastHit hit)
    {
        // Gestiona el impacto del raycast en el canvas de generación de
formas primitivas
```

```
        // Obtiene el nombre del objeto golpeado
        string hitObjectName = hit.collider.gameObject.name;

        // Decide qué tipo de figura primitiva crear basado en el nombre
del objeto
        if (hitObjectName == "Cube_icon" && gameObject.name == "Cube_icon")
        {
            CreatePrimitiveShape(PrimitiveType.Cube);
        }
         else if (hitObjectName == "Sphere_icon" && gameObject.name ==
"Sphere_icon")
        {
            CreatePrimitiveShape(PrimitiveType.Sphere);
        }
         else if (hitObjectName == "Cylinder_icon" && gameObject.name ==
"Cylinder_icon")
        {
            CreatePrimitiveShape(PrimitiveType.Cylinder);
        }
    }

    void CreatePrimitiveShape(PrimitiveType type)
    {
        // Crea la figura primitiva
        GameObject shape = GameObject.CreatePrimitive(type);

        if (transparentMaterial != null)
        {
            shape.GetComponent<Renderer>().material = transparentMaterial;
        }
        else
        {
            Debug.LogWarning("Transparent Material is not assigned. Please
assign it in the inspector.");
        }

        //Añade los scripts necesarios para su correcto funcionamiento
                                    var       posicionadorCilindros       =
shape.AddComponent<PosicionarCilindros>(); //Añade el script para crear los
ejes de traslación
        posicionadorCilindros.cilindroPrefab = grabAxisXPrefab; // Asigna
el prefab de eje de traslación

        var rotationalAxis = shape.AddComponent<RotationalAxis>();
        rotationalAxis.toroidePrefab = toroidePrefab; // Asigna el prefab
de eje de rotación
```

```
        var ExtrusionSpheres = shape.AddComponent<ExtrusionSpheres>();
        ExtrusionSpheres.sphere = extrusionSphere; // Asigna el prefab de
eje de dimensionado

       var modoControlador = shape.AddComponent<ModoControlador>(); //Esta
línea importa

       //ProBuilderMesh probuilder = shape.AddComponent<ProBuilderMesh>();
//Se puede eliminar, era para intentar corregir lo de vaciado

        var cotas = shape.AddComponent<TextoCotas>();


        shape.tag = "PrimaryObject"; // Asigna el tag al objeto

        // Se añade un GameObject vacío como hijo llamado 'EmptyScale'
        //Esto es para resolver el problema de la escala al modificar las
dimensiones del cubo
        GameObject emptyScale = new GameObject("EmptyScale");
        emptyScale.transform.SetParent(shape.transform); // Establece la
pieza como padre de 'EmptyScale'
                                                         // La transformada
del emptyScale tiene qeu ser la misma que la del padre
        emptyScale.transform.localPosition = Vector3.zero;
        emptyScale.transform.localRotation = Quaternion.identity;
        emptyScale.transform.localScale = Vector3.one; // La escala tiene
que ser neutra


       cotas.CreateCanvasText(shape, emptyScale);  //Añade las cotas y el
texto por primera vez

        // Específico para cilindros
        if (type == PrimitiveType.Cylinder)
        {
            // Posiciona la figura en frente del jugador o en una ubicación
deseada
              shape.transform.position = new Vector3(-11f, 0, 7.47f); //
Ejemplo de posición

            specialPointsScript.CreateCylinderPoints(shape);
            DestroyImmediate(shape.GetComponent<CapsuleCollider>());
            // Añade un MeshCollider y activa la opción convex
            //Convex es útil para que el collider se ajuste a la forma del
objeto y sea detectado correctamente por el raycast
```

```
            //ChangeMesh(shape, "Prefabs/Mallas/CylindroMalla v1");
           MeshCollider meshCollider = shape.AddComponent<MeshCollider>();
            meshCollider.convex = true;
        }else if (type == PrimitiveType.Cube)
        {
            specialPointsScript.CreateCubePoints(shape);
           // Posiciona la figura en frente del jugador o en una ubicación
deseada
             shape.transform.position = new Vector3(-11f, 0, -5.61f); //
Ejemplo de posición
            //ChangeMesh(shape, "Prefabs/Mallas/CuboMalla v1");
           MeshCollider meshcollyder = shape.GetComponent<MeshCollider>();
            meshcollyder.convex = true;
        }
        else
        {
            specialPointsScript.CreateSpherePoints(shape);
           // Posiciona la figura en frente del jugador o en una ubicación
deseada
            //ChangeMesh(shape, "Prefabs/Mallas/EsferaMalla v1");
              shape.transform.position = new Vector3(-11f, 0, 0.43f); //
Ejemplo de posición
                                    //MeshCollider     meshcollyder     =
shape.GetComponent<MeshCollider>();
            //meshcollyder.convex = true;
        }
    }
}
```

10.5 PosicionarCilindros

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

public class PosicionarCilindros : MonoBehaviour
{
    public GameObject cilindroPrefab; // El prefab del eje ya está asignado
en PrimitiveShapeGenerator
    private float alturaCilindro = 2.5f; //altura inicial del eje
    private float radioCilindro = 0.05f; // radio inicial del eje
```

```
    private List<GameObject> cilindrosCreados = new List<GameObject>();
//Para saber qué cilindros he creado para luego destruirlos al cambiar de
modo


    void PosicionarCilindrosEnEjes()
    {
            CrearCilindroEnEje(new Vector3(0, 0, 0), Color.green,
transform.GetChild(0)); // Eje X
            CrearCilindroEnEje(new Vector3(0, 0, -90), Color.red,
transform.GetChild(0)); // Eje Y
            CrearCilindroEnEje(new Vector3(90, 0, 0), Color.blue,
transform.GetChild(0)); // Eje Z
    }

    public void CrearCilindros()
    {
        DestruirCilindros(); //Solo por si acaso no se han destruido
        PosicionarCilindrosEnEjes();
    }

      void CrearCilindroEnEje(Vector3 rotacionEjeLocal, Color color,
Transform padre)
    {
        GameObject cilindro = Instantiate(cilindroPrefab, padre.position,
Quaternion.identity, padre);
        cilindrosCreados.Add(cilindro);

        Vector3 baseScale = new Vector3(radioCilindro * 2, alturaCilindro
/ 2, radioCilindro * 2); // Ajuste en la escala si es necesario

        // Asume que el hijo del que queremos basarnos es el primer hijo
del GameObject
        Transform childTransform = transform.GetChild(0);

        // Obtiene la escala del hijo
        Vector3 childScale = childTransform.localScale;

        // Calcula la proporción de la escala de los cilindros en relación
a la escala del hijo
        float promedio = (childScale.x + childScale.y + childScale.z) / 3;
            float maxChildScaleDimension = Mathf.Min(childScale.x,
childScale.y, childScale.z);
        Vector3 adjustedScale = new Vector3(
            baseScale.x * 1 / (promedio - 0.2f * promedio),
            baseScale.y * 1 / maxChildScaleDimension,
```

```
            baseScale.z * 1 / (promedio - 0.2f * promedio)
        );

        cilindro.transform.localScale = adjustedScale;

        // Aplica primero la rotación del padre para alinear el cilindro con este
        cilindro.transform.localRotation = Quaternion.identity; // Resetea la rotación local para aplicar la rotación del padre
                                                                // Luego aplica la rotación específica del eje para orientar correctamente el cilindro
                                    cilindro.transform.localRotation       = Quaternion.Euler(rotacionEjeLocal);

        cilindro.GetComponent<Renderer>().material.color = color;
    }

    public void DestruirCilindros()
    {
        //Método que destruye los cilindros que estén en la lista
        foreach (GameObject cilindro in cilindrosCreados)
        {
            Destroy(cilindro);
        }
        cilindrosCreados.Clear();
    }
}
```

10.6 CUBEMOVEMENT

```
using Unity.VisualScripting;
using UnityEngine;

public class CubeMovement : MonoBehaviour
{
    private Plane plane;
    private Transform raycastOrigin;   //Transform del objeto raycast
    private bool isMoving = false;       //Variable que dice si el modo mover está activado. Se desactiva cuando se suelta el gatillo
    private Vector3 lastHitPoint;       //Último punto en el que el raycast impacta con el plano
    private float initialUpDifference; // Almacena la diferencia inicial en el eje up
```

```
    void Start()
    {
        raycastOrigin = GameObject.Find("rightRaycastOrigin").transform;
        InitializePlane();
    }

    void Update()
    {
        // Solo ejecuta el código si modo mover está habilitado
        if (EventHub.CurrentMode == EventHub.ModoInteraccion.Mover)
        {
            // Lógica específica del modo Mover
            if (isMoving)
            {
                MoveObject();
            }
        }
    }


    void OnEnable()
    {
        EventHub.OnRaycastHit += HandleRaycastHit;
        EventHub.OnTriggerReleased += StopMoving; // Suscribirse al nuevo
evento para parar de moverse
    }

    void OnDisable()
    {
        EventHub.OnRaycastHit -= HandleRaycastHit;
         EventHub.OnTriggerReleased -= StopMoving; // Desuscribirse del
evento
    }

    void InitializePlane()
    {
        //Inicializa el plano la primera vez
        Vector3 upLocal = transform.up;
        if (upLocal != Vector3.up)
        {
                    Vector3 normalVector = Vector3.Cross(Vector3.up,
upLocal).normalized;
            plane = new Plane(normalVector, transform.position);
        }
```

```
        else
        {
            //Este es el caso en el que el eje Y local de la pieza coincida
con el eje Y global.
            //Si coinciden, son paralelos, y no se podría definir un plano
con dos vectores paralelos
            Vector3 normalVector = raycastOrigin.forward;
            plane = new Plane(normalVector, transform.position);
        }

    }

    void HandleRaycastHit(RaycastHit hit)
    {
        //Cuando el raycast impacta con el eje, ejecuta este método
        if (hit.transform == this.transform)
        {
             // Actualiza el plano con la posición y orientación actuales
antes de mover el objeto.
            UpdatePlaneToCurrentPosition();

             // Solo actualiza el punto de impacto si se está actualmente
moviendo el objeto.
            if (!isMoving)
            {
                 lastHitPoint = hit.point; // Guarda el punto de impacto
inicial.
              initialUpDifference = Vector3.Dot(transform.up, lastHitPoint
- transform.position);
            }
            isMoving = true; // Inicia o continua el movimiento.
        }
    }

     // Actualiza el plano según la posiciún y orientación actuales del
objeto.
    void UpdatePlaneToCurrentPosition()
    {
        //Este método vuelve a calcular el plano
         //Hay ocasiones donde es necesario volver a crear el plano, por
ejemplo, cuando se ha rotado la pieza
        Vector3 upLocal = transform.up;
        Vector3 normalVector;
        if (upLocal != Vector3.up)
        {
            normalVector = Vector3.Cross(Vector3.up, upLocal).normalized;
```

```
        }
        else
        {
            normalVector = raycastOrigin.forward;
        }
        plane = new Plane(normalVector, transform.position);
    }

    private void MoveObject()
    {
         //Método que se encarga de calcular cuánto se debe de mover el
objeto
        if (raycastOrigin == null)
        {
                    Debug.LogError("Raycast  Origin  no  está  asignado  en
CubeMovement.");
            return;
        }

          // raycastOrigin.position y raycastOrigin.forward defininen la
dirección del rayo
        Vector3 rayDirection = raycastOrigin.forward;
          // Inicia el rayo en la posición del origen del rayo hacia la
dirección que este apunta
        Ray ray = new Ray(raycastOrigin.position, rayDirection);

          // Verifica si este rayo intersecta con el plano definido en
InitializePlane.
        if (plane.Raycast(ray, out float enter))
        {
            // Calcula el punto exacto de intersección con el plano.
            Vector3 intersectionPoint = ray.GetPoint(enter);

             // Calcula la diferencia de posición desde la posición actual
del objeto hasta el punto de intersección.
                    Vector3  positionDifference  =  intersectionPoint  -
transform.position;

            // Proyecta esta diferencia en el eje up local del objeto para
determinar cuánto mover el objeto.
                float  upMovementAmount  =  Vector3.Dot(positionDifference,
transform.up);


           // Mueve el objeto en su eje up local por la cantidad calculada.
            if (transform.parent.parent != null)
```

```
            {
                    //float finalUpDifference = Vector3.Dot(transform.up,
lastHitPoint - transform.position);
                float compensation = upMovementAmount - initialUpDifference;
                       transform.parent.parent.position += transform.up *
compensation;
            }
        }
    }
    // Llamada para detener el movimiento cuando el trigger se dispara.
    public void StopMoving()
    {
        isMoving = false;
    }

}
```

10.7 ROTATIONALAXIS

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

public class RotationalAxis : MonoBehaviour
{
    public GameObject toroidePrefab; // Asigna el prefab del toroide en el
inspector de Unity
     private List<GameObject> toroidesCreados = new List<GameObject>();
//Guarda los ejes de rotación creados en una lista


    public void CrearToroides()
    {
        DestruirToroides();

        // Calcula el tamaño del objeto padre para posicionar los toroides
correctamente
        Vector3 parentSize = CalculateParentSize();
          float largestDimension = Mathf.Max(parentSize.x, parentSize.y,
parentSize.z);

        // Crea y posiciona los toroides con colores según los ejes
```

```
        CreateAndPositionToroid(toroidePrefab, Quaternion.Euler(0, 0, 90),
largestDimension, Color.blue); // Eje Y - Azul
        CreateAndPositionToroid(toroidePrefab, Quaternion.Euler(90, 0, 0),
largestDimension, Color.green); // Eje Z - Verde
        CreateAndPositionToroid(toroidePrefab, Quaternion.Euler(0, 90, 0),
largestDimension, Color.red); // Eje X - Rojo
    }

    void CreateAndPositionToroid(GameObject prefab, Quaternion rotacion,
float scale, Color color)
    {
        GameObject toroide = Instantiate(prefab, transform.GetChild(0));
        toroidesCreados.Add(toroide);
        toroide.transform.localRotation = rotacion;

        // Escala el toroide basado en la mayor dimensión del padre para
que esté 'por fuera'
        toroide.transform.localScale = new Vector3(scale, scale, scale) *
1f; // Ajustar este factor según sea necesario

        // Centra el toroide en el objeto padre
        toroide.transform.localPosition = Vector3.zero;

        toroide.GetComponent<Renderer>().material.color = color;

    }

    Vector3 CalculateParentSize()
    {
        //Calcula las dimensiones del objeto padre según su malla
        MeshRenderer meshRenderer = GetComponent<MeshRenderer>();
        if (meshRenderer != null)
        {
            return meshRenderer.bounds.size;
        }

        // Si el objeto no tiene un MeshRenderer, busca uno en los hijos
        meshRenderer = GetComponentInChildren<MeshRenderer>();
        if (meshRenderer != null)
        {
            return meshRenderer.bounds.size;
        }
        return new Vector3(1, 1, 1); // Tamaño predeterminado. Con este
valor funciona bien la escala por ahora
    }
```

```
    public void DestruirToroides()
    {
        //Destruye los toroides guardados en la lista
        foreach (GameObject toroide in toroidesCreados)
        {
            Destroy(toroide);
        }
        toroidesCreados.Clear();
    }
}
```

10.8 CUBEROTATION

```
using UnityEngine;

public class CubeRotation : MonoBehaviour
{
    private Plane plane;
    private Vector3 lastHitPoint;
    private Transform raycastOrigin;
    private bool isMoving = false;


    void OnEnable()
    {
        EventHub.OnRaycastHit += HandleRaycastHit;
        EventHub.OnTriggerReleased += StopMoving; // Suscribirse al nuevo
evento para dejar de rotar, cuando se suelta el gatillo
    }

    void OnDisable()
    {
        EventHub.OnRaycastHit -= HandleRaycastHit;
        EventHub.OnTriggerReleased -= StopMoving; // Desuscribirse del
evento
    }

    // Start se llama antes del primer frame en Update, cuando se llama al
script
    void Start()
    {
        raycastOrigin = GameObject.Find("rightRaycastOrigin").transform;
        InitializePlane();
```

```
    }

    // Update se llama todos los frames.
    void Update()
    {
        // Solo ejecutar el código si modo rotar está habilitado
        if (EventHub.CurrentMode == EventHub.ModoInteraccion.Rotar)
        {
            // Lógica específica del modo Rotar
            if (isMoving)
            {
                RotateObject();
            }
        }
    }

    void InitializePlane()
    {
        //Inicializa el plano la primera vez
        Vector3 upLocal = transform.up;
        Vector3 rightLocal = transform.right;

                    Vector3    normalVector    =    Vector3.Cross(rightLocal,
upLocal).normalized;
        plane = new Plane(normalVector, transform.position);
    }


    void HandleRaycastHit(RaycastHit hit)
    {
         //Ejecuta el método cuando los ejes han sido impactados por el
raycast
        if (hit.transform == this.transform)
        {
             // Actualiza el plano con la posición y orientación actuales
antes de rotar el objeto.
            InitializePlane();

             // Solo actualiza el punto de impacto si no está actualmente
rotando el objeto.
            if (!isMoving)
            {
                 lastHitPoint = hit.point; // Guarda el punto de impacto
inicial.
            }
            isMoving = true; // Inicia o continua el movimiento.
```

```
        }
    }

    private void RotateObject()
    {
         //Método que se encarga de calcular cuánto se debe de rotar el
objeto
        if (raycastOrigin == null)
        {
                    Debug.LogError("Raycast  Origin  no  está  asignado  en
CubeRotation.");
            return;
        }
        // Usa raycastOrigin.position y raycastOrigin.forward para definir
la dirección del rayo
        Vector3 rayDirection = raycastOrigin.forward;
         // Inicia el rayo en la posición del origen del rayo hacia la
dirección que este apunta
        Ray ray = new Ray(raycastOrigin.position, rayDirection);

          // Verifica si este rayo intersecta con el plano definido en
InitializePlane.
        if (plane.Raycast(ray, out float enter))
        {
            // Calcula el punto exacto de intersección con el plano.
            Vector3 intersectionPoint = ray.GetPoint(enter);

             // Convierte los puntos de intersección a coordenadas locales
del toroide para calcular la diferencia de ángulo.
                                        Vector3      lastLocalPoint      =
transform.InverseTransformPoint(lastHitPoint);
                                     Vector3     currentLocalPoint     =
transform.InverseTransformPoint(intersectionPoint);

            // Calcula el ángulo entre los dos puntos en el espacio local
del toroide.
            //SignedAngle calcula el ángulo en grados entre dos vectores,
alrededor de un eje específico
                    float   angle   =   Vector3.SignedAngle(lastLocalPoint,
currentLocalPoint, Vector3.forward);

           // Rotamos el toroide alrededor del eje normal al plano creado.
                    transform.parent.parent.Rotate(plane.normal,   angle,
Space.World);
```

```
            // Actualizamos el último punto de intersección para volver a
empezar.
            lastHitPoint = intersectionPoint;
        }
    }

    public void StopMoving()
    {
        isMoving = false;
    }
}
```

10.9 EXTRUSIONSPHERES

```
using AX.Generators;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Rendering.Universal;

public class ExtrusionSpheres : MonoBehaviour
{
     public GameObject sphere; // Arrastrar aquí el prefab esfera de
extrusión desde el Inspector
    private List<GameObject> esferasCreadas = new List<GameObject>(); //
Para saber qué esferas he creado para luego destruirlas


    void checkCollider()
    {
         // Intenta obtener diferentes tipos de colliders para sacar las
dimensiones del objeto
        Collider collider = GetComponent<Collider>();

        if (collider != null && sphere != null)
        {
            if (collider is BoxCollider box)
            {
                CreateSpheres(box.size, collider);
            }
            else if (collider is SphereCollider sphere)
            {
                float diameter = sphere.radius * 2;
                Vector3 size = new Vector3(diameter, diameter, diameter);
                CreateSpheres(size, collider);
```

```
            }
            else if (collider is MeshCollider meshCollider)
            {
                // Obtiene los límites del mesh asociado al MeshCollider
                Bounds bounds = meshCollider.sharedMesh.bounds;

                // El tamaño (size) del Bounds representa las dimensiones
del mesh en el espacio local del objeto
                Vector3 localSize = bounds.size;

                 // Escala el tamaño local por la escala del objeto para
obtener el tamaño en el mundo
                        Vector3  worldSize  =  Vector3.Scale(localSize,
transform.lossyScale);

                 // Asumiendo una orientación vertical estándar de Unity
(eje Y como altura)
                float width = worldSize.x; // Ancho, que corresponde con
la dimension X o Z
                float length = worldSize.z; // Largo igual al ancho para
un cilindro (al inicio es igual)
               float height = worldSize.y; // Altura basada en la dimensión
Y

                // Guarda las dimensiones calculadas en 'size'
                Vector3 size = new Vector3(1, 2, 1);
                Vector3 center = transform.TransformPoint(bounds.center);

                //Llama al método para crear las esferas de extrusión
                CreateSpheres(size, collider);
            }
            else
            {
                 Debug.LogError("Collider no soportado o el GameObject de
la esfera.");
            }
        }
        else
        {
            Debug.LogError("No se ha asignado el collider o el GameObject
de la esfera.");
        }
    }

    void CreateSpheres(Vector3 colliderSize, Collider collider)
    {
```

```
        // Método que se encarga de colocar las esferas de extrusión
          // Para simplificar, utilizo las mismas rotaciones y escalas
independientemente del tipo de collider
        Debug.Log(colliderSize);

        Quaternion defaultRotation = Quaternion.identity; // Sin rotación
(identidad)

        //Se definen posiciones y escalas para las distintas esferas
        Quaternion specialRotationX = Quaternion.Euler(180, 0, -90);
        Quaternion specialRotationZ = Quaternion.Euler(90, 0, 0);
        Quaternion specialRotationXYZ = Quaternion.Euler(45, 45, 0);

        Vector3 defaultScale = new Vector3(0.4f, 0.4f, 0.4f);
        Vector3 specialScale = new Vector3(0.1f, 0.5f, 0.1f);

            InstantiateSphere(new Vector3(0, 0, colliderSize.z / 2),
specialScale, specialRotationZ, "Extr_z");
           InstantiateSphere(new Vector3(colliderSize.x / 2 , 0, 0),
specialScale, specialRotationX, "Extr_x");
            InstantiateSphere(new Vector3(0, colliderSize.y / 2, 0),
specialScale, defaultRotation, "Extr_y");
       InstantiateSphere(new Vector3(colliderSize.x / 2, colliderSize.y /
2, colliderSize.z / 2), defaultScale, specialRotationXYZ, "Extr_scale");

        if (collider is MeshCollider meshCollider)
        {
             //Esfera adicional para añadir a los cilindros o a piezas
complejas (grupo, vaciado)
             //Especifico la orientación y la escala de la esfera para
cambiar el radio
            Quaternion specialRotationR = Quaternion.Euler(0, 180, 90);
            Vector3 specialScaleR = new Vector3(0.1f, 0.7f, 0.5f);
                 InstantiateSphere(new Vector3(colliderSize.x / 2, -
colliderSize.y/2 , 0), specialScaleR, specialRotationR, "Extr_radius");
        }
    }

     void InstantiateSphere(Vector3 localPosition, Vector3 baseScale,
Quaternion specificRotation, string tag)
    {
        // Asume que el hijo en el que queremos basarnos es el primer hijo
del GameObject
        Transform childTransform = transform.GetChild(0);

        // Obtiene la escala del hijo
```

```
        Vector3 childScale = childTransform.localScale;

        // Calcula la proporción de la escala de las esferas en relación a
la escala del hijo
        float promedio = (childScale.x + childScale.y + childScale.z) / 3;
              float   maxChildScaleDimension   =   Mathf.Min(childScale.x,
childScale.y, childScale.z);
        Vector3 adjustedScale = new Vector3(
            baseScale.x * 1 / promedio,
            baseScale.y * 1 / promedio,
            baseScale.z * 1 / promedio
        );

        // Calcula la posición global basada en la posición local deseada
y la transformación del padre
        Vector3 globalPosition = transform.TransformPoint(localPosition);

       // Combina la rotación del padre con la rotación específica deseada
               Quaternion   combinedRotation   =   transform.rotation   *
specificRotation;

       // Instancia la esfera con la posición global y la rotación combinada
         GameObject sphereInstance = Instantiate(sphere, globalPosition,
combinedRotation, transform.GetChild(0));

         //Usa 'adjustedScale' en lugar de 'scale' para la escala de la
esfera
        sphereInstance.transform.localScale = adjustedScale;
        sphereInstance.tag = tag;
        esferasCreadas.Add(sphereInstance); // Añade la esfera a la lista
de esferas creadas para luego poder eliminarlas si quiero

        // Cambia el color de la esfera a amarillo
        Renderer sphereRenderer = sphereInstance.GetComponent<Renderer>();
        if (sphereRenderer != null)
        {
            sphereRenderer.material.color = Color.yellow;
        }
        DestroyImmediate(sphereInstance.GetComponent<SphereCollider>());
        // Añade un MeshCollider y activa la opción convex (para que sea
más preciso y el raycast las detecte bien)
                                    MeshCollider       meshCollider       =
sphereInstance.AddComponent<MeshCollider>();
        meshCollider.convex = true;
    }
```

```
    public void CrearEsferas()
    {
        DestruirEsferas();
        checkCollider();
    }

    public void DestruirEsferas()
    {
        foreach (GameObject sphereInstance in esferasCreadas)
        {
            Destroy(sphereInstance);
        }
        esferasCreadas.Clear();
    }
}
```

10.10 EXTRUSIONMOVEMENT

```
using LibNoise.Unity.Generator;
using System;
using System.Collections.Generic;
using System.Xml.Serialization;
using UnityEngine;

public class ExtrusionMovement : MonoBehaviour
{
    private Plane plane;
    private Transform raycastOrigin;
    private bool isExtruding = false;    //Atributo que dice si está en el
modo redimensionar o no
    private Vector3 lastHitPoint;
     private float initialUpDifference; // Almacena la diferencia inicial
en el eje up
      private Dictionary<Transform, Vector3> originalChildScales = new
Dictionary<Transform, Vector3>(); // Para almacenar las escalas originales
de los hijos
    Vector3 oldScale;

    void OnEnable()
    {
        EventHub.OnRaycastHit += HandleRaycastHit;
          EventHub.OnTriggerReleased += StopExtruding; // Suscribirse al
nuevo evento
```

```
    }

    void OnDisable()
    {
        EventHub.OnRaycastHit -= HandleRaycastHit;
        EventHub.OnTriggerReleased -= StopExtruding; // Desuscribirse del
evento
    }

    void Start()
    {
        raycastOrigin = GameObject.Find("rightRaycastOrigin").transform;
        InitializePlane();
    }


    void Update()
    {
        // Solo ejecuta el código si modo redimensionar está habilitado
        if (EventHub.CurrentMode == EventHub.ModoInteraccion.Extruir)
        {
            // Lógica específica del modo redimensionar
            if (isExtruding)
            {
                ExtrudeObject();
            }
        }
    }

    void InitializePlane()
    {
        //Crea el plano por primera vez y lo actualiza cuando sea necesario
        Vector3 upLocal = transform.up;
        Vector3 normalVector;
        if (upLocal != Vector3.up)
        {
            normalVector = Vector3.Cross(Vector3.up, upLocal).normalized;
        }
        else
        {
            //Evita casos donde no se pueda crear el plano
            normalVector = raycastOrigin.forward;
        }
        plane = new Plane(normalVector, transform.position);
    }
```

```
    void HandleRaycastHit(RaycastHit hit)
    {
        //Método que se encarga de gestionar el impacto del raycast en los
ejes
        if (hit.transform == this.transform)
        {

            // Actualiza el plano con la posición y orientación actuales
antes de redimensionar el objeto.
            InitializePlane();

            // Verifica si es la primera vez que se redimensona y guarda
el punto de impacto
            if (!isExtruding)
            {
                lastHitPoint = hit.point; // Guarda el punto de impacto
inicial.
              initialUpDifference = Vector3.Dot(transform.up, lastHitPoint
- transform.position);
            }
            isExtruding = true; // Inicia o continua el movimiento.
        }
    }

    private void ExtrudeObject()
    {
        //Lógica que se encarga de redimensionar el objeto
        Debug.Log("Iniciando Extrusión");
        if (raycastOrigin == null)
        {
                Debug.LogError("Raycast Origin no está asignado en
CubeMovement.");
            return;
        }
        // Usa raycastOrigin.position y raycastOrigin.forward para definir
la dirección del rayo
        Vector3 rayDirection = raycastOrigin.forward;
         // Inicia el rayo en la posición del origen del rayo hacia la
dirección que este apunta
        Ray ray = new Ray(raycastOrigin.position, rayDirection);

         // Verifica si este rayo intersecta con el plano definido en
InitializePlane.
        if (plane.Raycast(ray, out float enter))
        {
            // Calcula el punto exacto de intersección con el plano.
```

```
            Vector3 intersectionPoint = ray.GetPoint(enter);
            Debug.Log($"Punto de Intersección: {intersectionPoint}");

            // Calcula la diferencia de posición desde la posición actual
del objeto hasta el punto de intersección.
                    Vector3 positionDifference = intersectionPoint -
transform.position;

            // Proyecta esta diferencia en el eje up local del objeto para
determinar cuánto redimensionar el objeto.
                float upMovementAmount = Vector3.Dot(positionDifference,
transform.up);
                    Debug.Log($"Cantidad de Movimiento hacia Arriba:
{upMovementAmount}");


             // Redimensiona el objeto en su eje up local (local de los
ejes) por la cantidad calculada.
            if (transform.parent.parent != null)
            {
                AdjustParentScale(upMovementAmount);
            }
        }

        void AdjustParentScale(float upMovementAmount)
        {
            float compensation = upMovementAmount - initialUpDifference;
                //Debug.Log($"Compensación: {compensation}, Nueva Escala
Calculada: {transform.parent.parent.localScale}");
               oldScale = transform.parent.parent.localScale; // Escala
original del padre antes del cambio
            //Debug.Log($"Escala Antes: {oldScale}");
            // Transforma el "up" del hijo al espacio del mundo y luego al
espacio local del padre para determinar la correspondencia del eje
                                        Vector3 upInParentSpace =
transform.parent.parent.InverseTransformDirection(transform.up).normalize
d;

             //Redimensiona el objeto en función del eje de extrusión que
haya sido impactado por el raycast
            if (gameObject.CompareTag("Extr_scale"))
            {
                 float newScaleX = transform.parent.parent.localScale.x +
compensation;
                 float newScaleY = transform.parent.parent.localScale.y +
compensation;
```

```
                float newScaleZ = transform.parent.parent.localScale.z +
compensation;
                newScaleX = Mathf.Max(newScaleX, 0.05f); // Asegura que la
nueva escala no sea menor que 0.05
                newScaleY = Mathf.Max(newScaleY, 0.05f); // Asegura que la
nueva escala no sea menor que 0.05
                newScaleZ = Mathf.Max(newScaleZ, 0.05f); // Asegura que la
nueva escala no sea menor que 0.05
               transform.parent.parent.localScale = new Vector3(newScaleX,
newScaleY , newScaleZ ); // Ajusta según el eje relevante
            }
             // Si el raycast ha impactado con la esfera de extrusión del
eje X
            else if (gameObject.CompareTag("Extr_x"))
            {
                // El eje "up" del hijo se alinea con el eje X del padre
                 float newXScale = transform.parent.parent.localScale.x +
compensation;
                newXScale = Mathf.Max(newXScale, 0.05f); // Asegura que la
nueva escala no sea menor que 0.05
               transform.parent.parent.localScale = new Vector3(newXScale,
transform.parent.parent.localScale.y,
transform.parent.parent.localScale.z); // Ajusta seg□n el eje relevante
            }
            else if (gameObject.CompareTag("Extr_y"))
            {
                // El eje "up" del hijo se alinea con el eje Y del padre
                 float newYScale = transform.parent.parent.localScale.y +
compensation;
                newYScale = Mathf.Max(newYScale, 0.05f); // Asegura que la
nueva escala no sea menor que 0.05
                                transform.parent.parent.localScale   =   new
Vector3(transform.parent.parent.localScale.x,                   newYScale,
transform.parent.parent.localScale.z); // Ajusta seg□n el eje relevante
            }
            else if (gameObject.CompareTag("Extr_z"))
            {
                // El eje "up" del hijo se alinea con el eje Z del padre
                 float newZScale = transform.parent.parent.localScale.z +
compensation;
                newZScale = Mathf.Max(newZScale, 0.05f); // Asegura que la
nueva escala no sea menor que 0.05
                                transform.parent.parent.localScale   =   new
Vector3(transform.parent.parent.localScale.x,
transform.parent.parent.localScale.y, newZScale); // Ajusta seg□n el eje
relevante
```

```
            }
            else if (gameObject.CompareTag("Extr_radius"))
            {
                // El eje "up" del hijo se alinea con el eje Z del padre
                 float newScaleX = transform.parent.parent.localScale.x +
compensation;
                 float newScaleZ = transform.parent.parent.localScale.z +
compensation;
                newScaleX = Mathf.Max(newScaleX, 0.05f); // Asegura que la
nueva escala no sea menor que 0.05
                newScaleZ = Mathf.Max(newScaleZ, 0.05f); // Asegura que la
nueva escala no sea menor que 0.05
               transform.parent.parent.localScale = new Vector3(newScaleX,
transform.parent.parent.localScale.y, newScaleZ); // Ajusta seg□n el eje
relevante
            }

            CheckCollider();

            //Se asegura de actializar el valor y tamaño de las cotas, si
están activas
            GameObject cotasCanvas = GameObject.Find("Cotas_Canvas");
                                    CotasCanvas     cotasCanvasScript     =
cotasCanvas.GetComponent<CotasCanvas>();

            if(cotasCanvasScript.cotasActivas)
            {
                                                  Canvas      canvas      =
transform.parent.GetComponentInChildren<Canvas>();
                EventHub.ActualizarCotas(transform.parent.parent, canvas,
false);
            }
                                        //Debug.Log($"Escala      Despu□s:
{transform.parent.parent.localScale}");
        }
    }


    void RestoreChildrensScales(Vector3 colliderSize)
    {
        //Reestablece la dimensión de todos los hijos para que no sufran
deformaciones
        Vector3 newScale = transform.parent.parent.localScale;
           Vector3  scaleFactor  =  new  Vector3(oldScale.x  /  newScale.x,
oldScale.y / newScale.y, oldScale.z / newScale.z);
```

```
        foreach (Transform child in transform.parent.parent)
        {
            // Aplica la escala a todos los hijos, que son los emptyScale
y los puntos significativos
           // No hace falta cambiar la dimension de los hijos de emptyScale,
ya que al modificar emptyScale ellos también se modifican
                child.localScale = new Vector3(child.localScale.x *
scaleFactor.x, child.localScale.y * scaleFactor.y, child.localScale.z *
scaleFactor.z);
        }

        // Busca y ajusta la posición de los hijos específicos basado en
sus tags
        foreach (Transform child in transform.parent)
        {
            if (child.tag == "Extr_z")
            {
                child.localPosition = new Vector3(0, 0, colliderSize.z / 2
* newScale.z);
            }
            else if (child.tag == "Extr_x")
            {
                  child.localPosition = new Vector3(colliderSize.x / 2 *
newScale.x, 0, 0);
            }
            else if (child.tag == "Extr_y")
            {
                 child.localPosition = new Vector3(0, colliderSize.y / 2 *
newScale.y, 0);
            }
            else if (child.tag == "Extr_scale")
            {
                  child.localPosition = new Vector3(colliderSize.x / 2 *
newScale.x, colliderSize.y / 2 * newScale.y, colliderSize.z / 2 *
newScale.z);
            }
            else if (child.tag == "Extr_radius")
            {
                  child.localPosition = new Vector3(colliderSize.x / 2 *
newScale.x, -colliderSize.y / 2 * newScale.y, 0);
            }
        }
    }


    void CheckCollider()
```

```
    {
                                        Collider        collider        =
transform.parent.parent.GetComponent<Collider>();
        // Intenta obtener diferentes tipos de colliders, solo se usa para
saber la dimensión de cada pieza en los distintos ejes
         //En realidad el colider no va a cambiar al cambiar la escala,
podría pasar simplemente las dimensiones iniciales. Lo mantengo porque hace
el código más general. Si se modifican las dimensiones iniciales sigue
funcionando
        if (collider != null)
        {
            if (collider is BoxCollider box)
            {
                RestoreChildrensScales(box.size);
            }
            else if (collider is SphereCollider sphere)
            {
                float diameter = sphere.radius * 2;
                Vector3 size = new Vector3(diameter, diameter, diameter);
                RestoreChildrensScales(size);
            }
            else if (collider is MeshCollider meshCollider)
            {
                // Obtiene los límites del mesh asociado al MeshCollider
                Bounds bounds = meshCollider.sharedMesh.bounds;

                // El tamaño (size) del Bounds representa las dimensiones
del mesh en el espacio local del objeto
                Vector3 localSize = bounds.size;

                 // Escala el tamaño local por la escala del objeto para
obtener el tamaño en el mundo
                        Vector3  worldSize  =  Vector3.Scale(localSize,
transform.lossyScale);

                 // Asumiendo una orientación vertical estándar de Unity
(eje Y como altura)
                float width = worldSize.x; // Ancho basado en la dimensión
más grande entre X y Z
                 float length = worldSize.z; // Largo igual al ancho para
un cilindro
               float height = worldSize.y; // Altura basada en la dimensión
Y

                // Guarda las dimensiones calculadas en 'size'
                Vector3 size = new Vector3(1, 2, 1);
```

```
                // Ahora 'size' contiene las dimensiones de ancho, alto y
largo del objeto en el mundo
                // Esta información está lista para ser usada para colocar
esferas las esferas de extrusión y para redimensionar el objeto
                RestoreChildrensScales(size);
            }
            else
            {
                Debug.LogError("Collider no soportado o el GameObject de
la esfera.");
            }
        }
        else
        {
            Debug.LogError("No se ha asignado el collider o el GameObject
de la esfera.");
        }
    }

    public void StopExtruding()
    {
        isExtruding = false;
    }
}
```

10.11 MODOCONTROLADOR

```
using UnityEngine;

public class ModoControlador : MonoBehaviour
{
    private PosicionarCilindros posicionadorCilindros;  //Script que crea
los ejes de traslación
    private RotationalAxis rotationalAxis;      //Script que crea los ejes
de rotación
    private ExtrusionSpheres extrusionSpheres;    //Script que crea las
esferas de redimensionado
    public bool activated = false;   //Variable que controla si las piezas
están activas (seleccionadas) o no
    public bool isTransparent = false;  //Variable que controla si las
piezas son transparentes o no
```

```
    private void Awake()
    {
        //Busca los scripts y los asigna
        posicionadorCilindros = GetComponent<PosicionarCilindros>();
        rotationalAxis = GetComponent<RotationalAxis>();
        extrusionSpheres = GetComponent<ExtrusionSpheres>();
    }

    private void ActivateObject(RaycastHit hit)
    {
       //Método que se encarga de activar el objeto y volverlo transparente
                            ModoControlador      modoControlador      =
hit.collider.GetComponent<ModoControlador>();
        if (modoControlador != null)
        {
            //Debug.Log("Objeto activado");
            modoControlador.activated = true;
            modoControlador.isTransparent = true;
            modoControlador.CrearObjetos();
            modoControlador.AddTransparency();
            VisualizacionCotasOn(hit);
        }
        else
        {
            Debug.Log(" modoControlador es null");
        }
    }

    private void ActivateOnlyObject(RaycastHit hit) //En este no creo los
ejes
    {
        //Este método convierte la pieza seleccionada en transparente. No
se crean los ejes
                            ModoControlador      modoControlador      =
hit.collider.GetComponent<ModoControlador>();
        if (modoControlador != null)
        {
            //Debug.Log("Objeto activado");
            //modoControlador.activated = true;
            modoControlador.AddTransparency();
            modoControlador.isTransparent = true;
        }
        else
        {
            Debug.Log(" modoControlador es null");
        }
```

```
    }

    private void ActivateOnlyObjectAll()
    {
        //Este método convierte todas las piezas en transparentes. No se
crean los ejes
        //activated = true;
        AddTransparency();
        isTransparent = true;
    }

    private void AddTransparency()
    {
        //Método encargado de cambiar el render y de añadir transparencia
al material
        Renderer renderer = GetComponent<Renderer>();
        if (renderer != null)
        {
            foreach (Material mat in renderer.materials) // Itera sobre
todos los materiales
            {
                // Cambia el shader a transparente para cada material
                                                    mat.SetInt("_SrcBlend",
(int)UnityEngine.Rendering.BlendMode.One);
                                                    mat.SetInt("_DstBlend",
(int)UnityEngine.Rendering.BlendMode.OneMinusSrcAlpha);
                mat.SetInt("_ZWrite", 0);
                mat.DisableKeyword("_ALPHATEST_ON");
                mat.DisableKeyword("_ALPHABLEND_ON");
                mat.EnableKeyword("_ALPHAPREMULTIPLY_ON");
                    mat.renderQueue = 3000; // RenderQueue de objetos
transparentes

                Color color = mat.color;
                color.a = 0.5f; // Semi-transparente
                mat.color = color;
            }
        }
        else
        {
            Debug.LogError("Renderer component not found.");
        }
    }

    private void RestoreOpacity()
    {
```

```
        //Método encargado de cambiar el render y de devolver la opacidad
al material
        Renderer renderer = GetComponent<Renderer>();
        if (renderer != null)
        {
             foreach (Material mat in renderer.materials) // Itera sobre
todos los materiales
            {
                Color color = mat.color;
                color.a = 1f; // Opaco
                mat.color = color;

                // Cambia el shader a opaco para cada material
                                                    mat.SetInt("_SrcBlend",
(int)UnityEngine.Rendering.BlendMode.One);
                                                    mat.SetInt("_DstBlend",
(int)UnityEngine.Rendering.BlendMode.Zero);
                mat.SetInt("_ZWrite", 1);
                mat.DisableKeyword("_ALPHATEST_ON");
                mat.DisableKeyword("_ALPHABLEND_ON");
                mat.DisableKeyword("_ALPHAPREMULTIPLY_ON");
                mat.renderQueue = -1; // Restaura el valor por defecto de
RenderQueue
            }
        }
        else
        {
            Debug.LogError("Renderer component not found.");
        }
    }

    public void DesactivateObject()
    {
         //Método encargado de desactivar el objeto. Lo vuelve opaco y
destruye los ejes.
        activated = false;
        isTransparent = false;
        DestruirObjetos();
        RestoreOpacity();
        VisualizacionCotasOff();
    }

    public void DestruirObjetos()
    {
        //Destruye cualquier eje que haya
        posicionadorCilindros.DestruirCilindros();
```

```
        extrusionSpheres.DestruirEsferas();
        rotationalAxis.DestruirToroides();
    }

    public void CrearObjetos()
    {
        // Crea los ejes, en función del modo en el que se esté
        switch (EventHub.CurrentMode)
        {
            case EventHub.ModoInteraccion.Rotar:
                rotationalAxis.CrearToroides();
                break;
            case EventHub.ModoInteraccion.Mover:
                posicionadorCilindros.CrearCilindros();
                break;
            case EventHub.ModoInteraccion.Extruir:
                extrusionSpheres.CrearEsferas();
                break;
        }
    }

    public void DestruirObjetosAnteriores()
    {
        // Destruye los ejes del método anterior al actual
        switch (EventHub.CurrentMode)
        {
            case EventHub.ModoInteraccion.Rotar:
                posicionadorCilindros.DestruirCilindros();
                break;
            case EventHub.ModoInteraccion.Mover:
                extrusionSpheres.DestruirEsferas();
                break;
            case EventHub.ModoInteraccion.Extruir:
                rotationalAxis.DestruirToroides();
                break;
        }
    }

    private void OnEnable()
    {
          EventHub.OnModeChange += AlternarModo; //Evento que dispara el
cambio de modo
        EventHub.OnActivateObject += ActivateObject;    //Evento que activa
el objeto y crea los ejes
         EventHub.OnActivateOnlyObject += ActivateOnlyObject;    //Evento
que activa el objeto, pero no crea los ejes (solo lo vuelve transparente)
```

```
        EventHub.OnResetActivation += DesactivateObject;     //Evento que
desactiva el objeto
       EventHub.OnActivateOnlyObjectAll += ActivateOnlyObjectAll; //Evento
que vuelve transparentes todos los objetos, sin crear los ejes
    }

    private void OnDisable()
    {
        EventHub.OnModeChange -= AlternarModo;
        EventHub.OnActivateObject -= ActivateObject;
        EventHub.OnActivateOnlyObject -= ActivateOnlyObject;
        EventHub.OnResetActivation -= DesactivateObject;
        EventHub.OnActivateOnlyObjectAll -= ActivateOnlyObjectAll;
    }


    private void AlternarModo()
    {
        //Método que se encarga de alternar el modo
        if (activated)
        {
            DestruirObjetosAnteriores();
            CrearObjetos();
        }
    }

    void VisualizacionCotasOn(RaycastHit hit)
    {
        //Método que se encarga de activar la visualización de las cotas
        CotasCanvas cotas = FindObjectOfType<CotasCanvas>();
        //Para la visualizacion del texto de las cotas
        foreach (Transform child in hit.transform)
        {
            foreach (Transform grandChild in child)
            {
                // Verifica si el hijo tiene el tag deseado
                if (grandChild.CompareTag("CanvasCotas"))
                {
                    // Si el hijo tiene el tag, desactiva sus hijos (nietos
del objeto original)
                    foreach (Transform greatgrandChild in grandChild)
                    {
                        greatgrandChild.gameObject.SetActive(cotas.cotasA
ctivas);
                    }
                }
```

```
            }
        }
    }

    void VisualizacionCotasOff()
    {
        //Desactiva la visualización de las cotas
        // Busca todos los GameObjects con la etiqueta "CanvasCotas"
                                GameObject[]       canvasObjects       =
GameObject.FindGameObjectsWithTag("CanvasCotas");

        // Comprobar si se encontraron objetos con la etiqueta
        if (canvasObjects.Length == 0)
        {
        }
        else
        {
             // Iterar sobre cada GameObject encontrado con la etiqueta
"CanvasCotas"
            foreach (GameObject canvasObject in canvasObjects)
            {
                // Desactivar cada hijo del GameObject
                foreach (Transform child in canvasObject.transform)
                {
                    child.gameObject.SetActive(false);
                }
            }
        }
    }
}
```

10.12 ELIMINATEOBJECT

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

public class EliminateObject : MonoBehaviour
{
    void OnEnable()
    {
        EventHub.OnEliminateObjectHit += EliminatePart;
    }
```

```
    void OnDisable()
    {
        EventHub.OnEliminateObjectHit -= EliminatePart;
    }

    void EliminatePart(RaycastHit hit)
    {
        // Encuentra todos los objetos en la escena que tienen el script
ModoControlador
        ModoControlador[] objetos = FindObjectsOfType<ModoControlador>();

        // Revisa cada objeto
        foreach (ModoControlador objeto in objetos)
        {
            // Si la variable 'activated' es true, destruye el objeto
            if (objeto.activated)
            {
                Destroy(objeto.gameObject);
            }
        }
    }
}
```

10.13 VACIADO

```
using UnityEngine;
using Parabox.CSG;
using AX.Generators;
using static UnityEngine.GraphicsBuffer;
using System.Collections.Generic;
using MIConvexHull;
using System.Linq;
using Sabresaurus.SabreCSG;
using Sabresaurus.SabreCSG.ShapeEditor;
//using UnityEngine.ProBuilder;
//using UnityEngine.ProBuilder.MeshOperations;


public class Vaciado : MonoBehaviour
{
    private GameObject objeto1 = null;      //Objeto sobre el que se realiza
el vaciado
```

```
    private GameObject objeto2 = null;      //Objeto que realiza el vaciado
     public bool isVaciado = false;            //Atributo que dice si la
función vaciado está activa
    public GameObject toroidePrefab;        //Prefab de los ejes de rotación
      public GameObject grabAxisXPrefab;        //Prefab de los ejes de
traslación
     public GameObject extrusionSphere;       //Prefab de las esferas de
extrusión
    public Vector3 objectPos = Vector3.zero;
    public Vector3 objectRot = Vector3.zero;



    private void OnEnable()
    {
        EventHub.OnVaciadoHit += SelectObjects;
        EventHub.OnSelectedObjects += SelectObjects;
    }

    private void OnDisable()
    {
        EventHub.OnVaciadoHit -= SelectObjects;
        EventHub.OnSelectedObjects -= SelectObjects;
    }


    void SelectObjects(RaycastHit hit)
    {
         //Método que actua mientras se seleccionan las piezas que van a
formar parte del vaciado
        isVaciado = true;
        if(objeto1 == null && objeto2 == null)
        {
            EventHub.ResetActivation();
        }
        // Comprueba si el objeto está activado y aún no está asignado
        if (hit.collider.gameObject.tag == "PrimaryObject")
        {
            if (objeto1 == null)
            {
                objeto1 = hit.collider.gameObject;
                     //Debug.Log("Primer objeto activado asignado: " +
controlador.gameObject.name);
            }
           else if (objeto2 == null && hit.collider.gameObject != objeto1)
            {
```

```
                objeto2 = hit.collider.gameObject;
                    //Debug.Log("Segundo objeto activado asignado: " +
controlador.gameObject.name);
            }
        }

       // Si ambos objetos están activos y asignados, llama a SubtractMeshes
y desactiva la funcion de vaciado
        if (objeto1 != null && objeto2 != null)
        {
            SubtractMeshes(objeto1, objeto2);
            EventHub.ResetActivation();
            objeto1 = null;
            objeto2 = null;
            isVaciado = false;
        }

    }


    void SubtractMeshes(GameObject objeto1, GameObject objeto2)
    {
        //Método que se encarga de implementar la función de vaciado
       Vector3 oldScale2 = objeto2.transform.localScale; //Para usarlo más
tarde al pasar las esferas especiales

        MeshFilter meshObjeto1 = objeto1.GetComponent<MeshFilter>();
        MeshFilter meshObjeto2 = objeto2.GetComponent<MeshFilter>();

        //Funciones para limpiar las mallas antes de hacer el vaciado
        meshObjeto1.mesh.RecalculateNormals();
        meshObjeto1.mesh.RecalculateTangents();
        meshObjeto1.mesh.RecalculateBounds();

        if (objeto1 == null || objeto2 == null)
        {
            Debug.LogError("Objetos no asignados.");
            return;
        }

        // Realiza la operación de vaciado

         //Me guardo los hijos de objeto1 (Para luego pasar los puntos
especiales a objeto 2)
        List<Transform> children1 = new List<Transform>();
        List<Transform> children2 = new List<Transform>();
```

```
        Vector3 scaleObjeto1 = objeto1.transform.localScale;
        Vector3 scaleObjeto2 = objeto2.transform.localScale;

        // Llena la lista con los hijos apropiados
        foreach (Transform child in objeto1.transform)
        {
            if (child.name != "EmptyScale") // Excluir "EmptyScale"
            {
                children1.Add(child);
                  child.localScale = new Vector3(1 / scaleObjeto1.x, 1 /
scaleObjeto1.y, 1 / scaleObjeto1.z);
            }
        }

        foreach (Transform child in objeto2.transform)
        {
            if (child.name != "EmptyScale") // Excluir "EmptyScale"
            {
                children2.Add(child);
            }
        }

       TrucoPosicion(objeto1, objeto2); //Truco que me arregla el problema
de que el nuevo objeto creado tiene coordenadas 0.

        objeto1.transform.position -= objectPos;
        //objeto2.transform.position -= objectPos;

        Model resultModel = CSG.Subtract(objeto1, objeto2);

        //Vuelvo a limpiar la malla por si acaso
        resultModel.mesh.RecalculateNormals();
        resultModel.mesh.RecalculateTangents();
        resultModel.mesh.RecalculateBounds();

        // Crea un nuevo GameObject para mostrar el resultado
        GameObject resultObj = new GameObject("Resulting Object");
        resultObj.transform.position = objectPos;
        resultObj.transform.eulerAngles = objectRot;
        objectPos = Vector3.zero;
        objectRot = Vector3.zero;
        MeshFilter meshFilter = resultObj.AddComponent<MeshFilter>();
       MeshRenderer meshRenderer = resultObj.AddComponent<MeshRenderer>();
```

```
        // Asigna la malla resultante y materiales al nuevo GameObject
        meshFilter.mesh = resultModel.mesh;

        //Y vuelvo a limpiar la malla
        meshFilter.mesh.RecalculateNormals();
        meshFilter.mesh.RecalculateTangents();
        meshFilter.mesh.RecalculateBounds();

        //RecalculateMesh(meshFilter); //Este método no lo voy a usar, no
mejoraba el error

        //Le añado una malla y un collider a la nueva pieza
        meshRenderer.materials = resultModel.materials.ToArray();
       MeshCollider meshCollider = resultObj.AddComponent<MeshCollider>();
        meshCollider.sharedMesh = resultModel.mesh;
        meshCollider.convex = true; // Establece el collider como convexo.
El collider se ajusta a la malla
        meshCollider.isTrigger = true; // El collider es detectable por el
raycast

        resultObj.tag = "PrimaryObject"; // Asigna el tag al objeto
                                    var       posicionadorCilindros       =
resultObj.AddComponent<PosicionarCilindros>();
        posicionadorCilindros.cilindroPrefab = grabAxisXPrefab; // Asigna
el prefab de grabAxisX

        var rotationalAxis = resultObj.AddComponent<RotationalAxis>();
        rotationalAxis.toroidePrefab = toroidePrefab; // Asigna el prefab
de toroide

        var ExtrusionSpheres = resultObj.AddComponent<ExtrusionSpheres>();
        ExtrusionSpheres.sphere = extrusionSphere; // Asigna el prefab de
las esferas de extrusión

        var modoControlador = resultObj.AddComponent<ModoControlador>();

        GameObject emptyScale = new GameObject("EmptyScale");
       emptyScale.transform.SetParent(resultObj.transform); //Empty scale
se vuelve hijo de la pieza

        emptyScale.transform.localPosition = Vector3.zero;
        emptyScale.transform.localRotation = Quaternion.identity;
        emptyScale.transform.localScale = Vector3.one;

        Vector3 oldScale = objeto1.transform.localScale;
        // Mover los hijos de la lista temporal al nuevo objeto padre
```

```
        foreach (Transform child in children1)
        {
              child.SetParent(resultObj.transform, false); // false para
mantener la transformación local respecto al nuevo padre
            child.localScale = new Vector3(0.2f, 0.2f, 0.2f);
               child.localPosition = new Vector3(child.localPosition.x *
oldScale.x, child.localPosition.y * oldScale.y, child.localPosition.z *
oldScale.z);
        }

        // Mover los hijos de la lista temporal al nuevo objeto padre
        foreach (Transform child in children2)
        {
              child.SetParent(resultObj.transform, false); // false para
mantener la transformación local respecto al nuevo padre
            child.localScale = new Vector3(0.2f, 0.2f, 0.2f);
               child.localPosition = new Vector3(child.localPosition.x *
oldScale2.x, child.localPosition.y * oldScale2.y, child.localPosition.z *
oldScale2.z);
        }

        var cotas = resultObj.AddComponent<TextoCotas>();
         cotas.CreateCanvasText(resultObj, emptyScale);  //Añade las cotas
y el texto por primera vez

        Destroy(objeto1);
        //Destroy(objeto2);
    }

    void TrucoPosicion(GameObject objeto1, GameObject objeto2)
    {
        //Método que se aencarga de solucionar el problema de que el origen
de la pieza no es su centro de masas
        objeto2.transform.parent = objeto1.transform;
        objectRot = objeto1.transform.eulerAngles;
        objeto1.transform.eulerAngles -= objectRot;

        Model resultModel = CSG.Subtract(objeto1, objeto2);
        // Crea un nuevo GameObject para mostrar el resultado
        GameObject resultObj = new GameObject("Resulting Object");
        MeshFilter meshFilter = resultObj.AddComponent<MeshFilter>();
       MeshRenderer meshRenderer = resultObj.AddComponent<MeshRenderer>();

        // Asigna la malla resultante y materiales al nuevo GameObject
        meshFilter.mesh = resultModel.mesh;
        Mesh mesh = meshFilter.mesh; // Accede al Mesh
```

```
        mesh.RecalculateNormals();
        mesh.RecalculateTangents();
        mesh.RecalculateBounds();
        objectPos = mesh.bounds.center;
        Destroy(resultObj);
    }

}
```

10.14 CREATESPECIALPOINTS

```
using UnityEngine;

public class CreateSpecialPoints : MonoBehaviour
{

    public GameObject specialPoint; // Asigna el prefab del cilindro en el
Inspector
     public GameObject specialAxe; // Asigna el prefab del cilindro en el
Inspector

    public void CreateCubePoints(GameObject cube)
    {
        // Método que crea los puntos característicos en el cubo
        // Se asume que el cubo es regular y está alineado con los ejes
        // Obtiene el tamaño del cubo
        Vector3 size = cube.GetComponent<Renderer>().bounds.size;

        // Calcula el half size para posicionar correctamente los puntos
        float halfX = size.x / 2;
        float halfY = size.y / 2;
        float halfZ = size.z / 2;

        // Vector con las esquinas del cubo
        Vector3[] corners = new Vector3[]
        {
        new Vector3(halfX, halfY, halfZ),
        new Vector3(halfX, halfY, -halfZ),
        new Vector3(halfX, -halfY, halfZ),
        new Vector3(halfX, -halfY, -halfZ),
        new Vector3(-halfX, halfY, halfZ),
        new Vector3(-halfX, halfY, -halfZ),
        new Vector3(-halfX, -halfY, halfZ),
        new Vector3(-halfX, -halfY, -halfZ)
```

```
        };

        // vector con los centros de las caras del cubo
        Vector3[] faceCenters = new Vector3[]
        {
        new Vector3(halfX, 0, 0),   // Centro de la cara positiva en X
        new Vector3(-halfX, 0, 0),  // Centro de la cara negativa en X
        new Vector3(0, halfY, 0),   // Centro de la cara positiva en Y
        new Vector3(0, -halfY, 0),  // Centro de la cara negativa en Y
        new Vector3(0, 0, halfZ),   // Centro de la cara positiva en Z
        new Vector3(0, 0, -halfZ)   // Centro de la cara negativa en Z
        };

        // Coloca los puntos en cada esquina
        foreach (Vector3 corner in corners)
        {
                        GameObject   point   =   Instantiate(specialPoint,
cube.transform.position + corner, Quaternion.identity);
            point.transform.parent = cube.transform; // Opcional: hacer que
el punto sea hijo del cubo
            point.tag = "SpecialPoint"; // Asigna el tag al objeto
            DeActivateVisualizationSpecialPoints(point);
            ChangeColorToTurquoise(point); // Cambia el color a turquesa
        }

        // Coloca los puntos en el centro de cada cara
        foreach (Vector3 faceCenter in faceCenters)
        {
                        GameObject   point   =   Instantiate(specialPoint,
cube.transform.position + faceCenter, Quaternion.identity);
            point.transform.parent = cube.transform; // Opcional: hacer que
el punto sea hijo del cubo
            point.tag = "SpecialPoint"; // Asigna el tag al objeto
            DeActivateVisualizationSpecialPoints(point);
            ChangeColorToTurquoise(point); // Cambia el color a turquesa
        }

        // Punto en el centro del cubo
                GameObject   centerPoint   =   Instantiate(specialPoint,
cube.transform.position, Quaternion.identity);
        centerPoint.transform.parent = cube.transform; // Opcional: hacer
que el punto sea hijo del cubo
        centerPoint.tag = "SpecialPoint"; // Asigna el tag al objeto
        DeActivateVisualizationSpecialPoints(centerPoint);
        ChangeColorToTurquoise(centerPoint); // Cambia el color a turquesa
    }
```

```
    public void CreateCylinderPoints(GameObject cylinder)
    {
        //Método que crea los puntos característicos en el cilindro
        if (cylinder == null || specialPoint == null || specialAxe == null)
        {
            Debug.LogError("GameObject or prefabs not set properly.");
            return;
        }

                Vector3   topCenter   =   cylinder.transform.position   +
cylinder.transform.up * (cylinder.transform.localScale.y);
               Vector3   bottomCenter   =   cylinder.transform.position   -
cylinder.transform.up * (cylinder.transform.localScale.y);
        Vector3 cylinderCenter = cylinder.transform.position;

        // Crea los puntos en los centros de las bases
            GameObject  topPoint  =  Instantiate(specialPoint,  topCenter,
Quaternion.identity);
         GameObject bottomPoint = Instantiate(specialPoint, bottomCenter,
Quaternion.identity);
        GameObject centerPoint = Instantiate(specialPoint, cylinderCenter,
Quaternion.identity);

        // Asegurarse de que los puntos son hijos del cilindro
        topPoint.transform.parent = cylinder.transform;
        bottomPoint.transform.parent = cylinder.transform;
        centerPoint.transform.parent = cylinder.transform;

        // Cambiar el color a turquesa
        topPoint.tag = "SpecialPoint"; // Asigna el tag al objeto
        bottomPoint.tag = "SpecialPoint"; // Asigna el tag al objeto
        centerPoint.tag = "SpecialPoint"; // Asigna el tag al objeto
        DeActivateVisualizationSpecialPoints( topPoint);
        DeActivateVisualizationSpecialPoints( centerPoint);
        DeActivateVisualizationSpecialPoints( bottomPoint);
        ChangeColorToTurquoise(topPoint);
        ChangeColorToTurquoise(bottomPoint);
        ChangeColorToTurquoise(centerPoint);

        // Crea los puntos en las tangentes de cada base
         Vector3 tangentDirection = cylinder.transform.right; // Dirección
tangencial en la base, asumiendo que es perpendicular al eje del cilindro
         float radius = cylinder.transform.localScale.x / 2; // La escala
en X es el diámetro
```

```
        GameObject topTangentPoint = Instantiate(specialPoint, topCenter +
tangentDirection * radius, Quaternion.identity);
             GameObject  bottomTangentPoint  =  Instantiate(specialPoint,
bottomCenter + tangentDirection * radius, Quaternion.identity);
        topTangentPoint.transform.parent = cylinder.transform;
        bottomTangentPoint.transform.parent = cylinder.transform;
        topTangentPoint.tag = "SpecialPoint"; // Asigna el tag al objeto
       bottomTangentPoint.tag = "SpecialPoint"; // Asigna el tag al objeto
        DeActivateVisualizationSpecialPoints( topTangentPoint);
        DeActivateVisualizationSpecialPoints( bottomTangentPoint);
        ChangeColorToTurquoise(topTangentPoint);
        ChangeColorToTurquoise(bottomTangentPoint);

        // Crear un eje que sea coaxial al eje del cilindro (Al final esto
no lo voy a implementar)
        //GameObject coaxialAxe = Instantiate(specialAxe, cylinderCenter,
Quaternion.LookRotation(cylinder.transform.forward));
        //coaxialAxe.transform.parent = cylinder.transform;
              //coaxialAxe.transform.localScale   =   new   Vector3(0.05f,
1.5f*cylinder.transform.localScale.y, 0.05f); // Ajustar el tamaño para que
coincida con la altura del cilindro
        //ChangeColorToTurquoise(coaxialAxe);
    }


    public void CreateSpherePoints(GameObject sphere)
    {
        //Método que crea los puntos característicos en la esfera
        if (sphere == null || specialPoint == null)
        {
            Debug.LogError("GameObject or prefab not set properly.");
            return;
        }

        // Centro de la esfera
        Vector3 sphereCenter = sphere.transform.position;

        // Crea un punto en el centro de la esfera
         GameObject centerPoint = Instantiate(specialPoint, sphereCenter,
Quaternion.identity);
          centerPoint.transform.parent  =  sphere.transform;  //Hace  que  el
punto sea hijo de la esfera
        centerPoint.tag = "SpecialPoint"; // Asigna el tag al objeto
        DeActivateVisualizationSpecialPoints(centerPoint);
        ChangeColorToTurquoise(centerPoint);
```

```
        // Calcula un punto tangente
         // Supongamos que queremos un punto en la parte superior de la
esfera, lo cual es tangente en la dirección del eje Y local
        float radius = sphere.transform.localScale.x / 2;  // Asumimos que
la esfera es uniformemente escalada
        Vector3 tangentPointPosition = sphereCenter + sphere.transform.up
* radius;

        // Crea un punto en la tangente
                GameObject   tangentPoint   =   Instantiate(specialPoint,
tangentPointPosition, Quaternion.identity);
        tangentPoint.transform.parent = sphere.transform; // Hacer que el
punto sea hijo de la esfera
        tangentPoint.tag = "SpecialPoint"; // Asigna el tag al objeto
        DeActivateVisualizationSpecialPoints( tangentPoint);
        ChangeColorToTurquoise(tangentPoint);
    }

    public void ChangeColorToTurquoise(GameObject shape)
    {
        // Verifica si el GameObject y su Renderer están presentes
        if (shape != null && shape.GetComponent<Renderer>() != null)
        {
            // Cambia el color del material del primer Renderer encontrado
(debería de ser el único)
             shape.GetComponent<Renderer>().material.color = new Color(64f
/ 255f, 224f / 255f, 208f / 255f);
        }
        else
        {
            Debug.LogError("El objeto specialPoint no existe o no tiene un
componente Renderer.");
        }
    }

    public void DeActivateVisualizationSpecialPoints(GameObject point)
    {
         //Hace que los puntos no estén visibles (no los elimina, solo
deshabilita el renderer, los oculta)
        //No funciona si los puntos se desactivan o si se eliminan
        Renderer renderer = point.GetComponent<Renderer>();
        renderer.enabled = false;
    }
}
```

10.15 MOVIMIENTOPORPUNTOS

```
using System.Collections;
using System.Collections.Generic;
//using System.Drawing;
using System.Security.Cryptography;
using System.Threading;
using UnityEngine;

public class MovimientoPorPuntos : MonoBehaviour
{
     public bool isActive = false;  //Atributo que dice si el modo está activo o no
     private List<Transform> selectedPoints = new List<Transform>(); // Lista para guardar los puntos seleccionados
    private GameObject parentObject; //padre de los puntos (0,2,4), pieza que se mueve
    private GameObject parentObject2 = null; //padre de los puntos (1,3,5)
     int hasStarted = 0;  //Variable para saber cuando se dispara a la primera bola y así fijar el primer game object que se le disparó
    int count = 0;


    private void OnEnable()
    {
        EventHub.OnSelectedSpecialPoints += MoverPorPuntos;
        EventHub.OnSpecialPointsCanvas += Alternate;
    }

    private void OnDisable()
    {
        EventHub.OnSelectedSpecialPoints -= MoverPorPuntos;
        EventHub.OnSpecialPointsCanvas -= Alternate;
    }

    void MoverPorPuntos(RaycastHit hit)
    {
         //Se encarga de la lógica para aplicar el moviemiento cuando el raycast da en los puntos especiales
                        if (hit.transform != null && hit.transform.CompareTag("SpecialPoint"))
        {
            if (count == 0 || count == 2 || count == 4)
            {
```

```
                if (hasStarted == 1 && hit.transform.parent.gameObject ==
parentObject)
                {
                    selectedPoints.Add(hit.transform); // Agregar el punto
a la lista de puntos seleccionados
                    count = selectedPoints.Count;
                    AplicarMovimiento(hit);
                }
                else if (hasStarted == 0)
                {
                    selectedPoints.Add(hit.transform); // Agregar el punto
a la lista de puntos seleccionados
                    count = selectedPoints.Count;
                    hasStarted = 1;
                    AplicarMovimiento(hit);
                }
            }
            else if (count == 1 || count == 3 || count == 5)
            {
                 //Esto es para asegurarse que el usuario solo pueda usar
los puntos característicos de 2 objetos, y en el orden correcto
                if (count == 1 && hit.transform.parent.gameObject !=
parentObject)
                {
                    parentObject2 = hit.transform.parent.gameObject;
                }
                if (hasStarted == 1 && hit.transform.parent.gameObject ==
parentObject2)
                {
                    selectedPoints.Add(hit.transform); // Agregar el punto
a la lista de puntos seleccionados
                    count = selectedPoints.Count;
                    AplicarMovimiento(hit);
                }
            }
        }
    }

    void AplicarMovimiento(RaycastHit hit)
    {
         //Se encarga de la lógica para aplicar el movimiento cuando el
raycast da en los puntos especiales
        if (count <= 6)
        {
            ChangeColorToGreen(hit.transform.gameObject);
        }
```

```
        if (count == 1)
        {
             //Guarda el objeto que se va a mover, para luego impedir que
se seleccionen puntos de otro objeto
            parentObject = hit.transform.parent.gameObject;
        }
        if (count == 2) // Solo para los dos primeros puntos
        {
            Transform firstPoint = selectedPoints[0];
            Transform secondPoint = selectedPoints[1];

            // Calcula la diferencia de posición entre el primer y segundo
punto seleccionado
                    Vector3   positionOffset   =   secondPoint.position   -
firstPoint.position;

            // Mueve el objeto padre del primer punto seleccionado
            parentObject.transform.position += positionOffset;
        }
         else if (count == 4) // Cuando se han seleccionado 4 puntos en
total
        {
                Transform  refPoint  =  selectedPoints[0];  //  Primer  punto
seleccionado como referencia
            Transform targetPoint = selectedPoints[count - 2]; // Penúltimo
punto seleccionado
            Transform newPoint = selectedPoints[count - 1]; // último punto
seleccionado

                    RotateAroundPoint2(parentObject,   refPoint.position,
newPoint.position    -    refPoint.position,    targetPoint.position    -
refPoint.position);
        }
        else if (count == 6)
        {
            RotateAroundPoint3(parentObject, selectedPoints[0].position);
        }
    }

      void  RotateAroundPoint2(GameObject  obj,  Vector3  point,  Vector3
targetDir, Vector3 currentDir)
    {
        //Método encargado de posicionar la pieza cuando se seleccionan 2
puntos en cada pieza
        // Normaliza los vectores de dirección
```

```
        targetDir.Normalize();
        currentDir.Normalize();

       // Calcula el eje de rotación como el perpendicular al plano formado
por los vectores targetDir y currentDir
                Vector3   axisOfRotation   =   Vector3.Cross(currentDir,
targetDir).normalized;

        // Calcula el ángulo necesario para la rotación
             float  angle  =  Vector3.SignedAngle(currentDir,  targetDir,
axisOfRotation);

        // Aplica la rotación alrededor del punto especificado utilizando
el eje calculado
        if (axisOfRotation != Vector3.zero) // Asegurarse de que el eje no
sea cero
        {
            obj.transform.RotateAround(point, axisOfRotation, angle);
        }
    }

    void RotateAroundPoint3(GameObject obj, Vector3 point)
    {
        //Método encargado de posicionar la pieza cuando se seleccionan 3
puntos en cada pieza
        // Calcular normales de los planos
           Vector3  normal1  =  Vector3.Cross(selectedPoints[2].position  -
selectedPoints[0].position,          selectedPoints[4].position          -
selectedPoints[0].position).normalized;
           Vector3  normal2  =  Vector3.Cross(selectedPoints[3].position  -
selectedPoints[1].position,          selectedPoints[5].position          -
selectedPoints[1].position).normalized;

        // Eje de rotación dado
              Vector3   axisOfRotation   =   selectedPoints[3].position   -
selectedPoints[0].position;

        // Calcula el ángulo necesario para alinear las normales
               float   angle   =   Vector3.SignedAngle(normal1,   normal2,
axisOfRotation);

        // Aplicar rotación
        if (axisOfRotation != Vector3.zero)
        {
            obj.transform.RotateAround(point, axisOfRotation, angle);
        }
```

```
    }


    void Alternate(RaycastHit hit)
    {
        isActive = !isActive;
        EventHub.ResetActivation();
        if (isActive)
        {
            EventHub.ActivateOnlyObjectAll();
                                    GameObject[]     specialPoints     =
GameObject.FindGameObjectsWithTag("SpecialPoint");

            //Devuelve el color de los puntos especiales a turquesa
            foreach (GameObject point in specialPoints)
            {
                ChangeColorToTurquoise(point);
            }
            selectedPoints.Clear();
            count = 0;
            hasStarted = 0;
        }

                                GameObject[]       primaryObjects       =
GameObject.FindGameObjectsWithTag("PrimaryObject");
        foreach (GameObject primaryObject in primaryObjects)
        {
            foreach (Transform child in primaryObject.transform)
            {
                if (child.CompareTag("SpecialPoint"))
                {
                    if(isActive)
                    {
                        ChangeColorToTurquoise(child.gameObject);
                    }
                    child.gameObject.SetActive(true);
                    //child.gameObject.SetActive(isActive);
                    Renderer renderer = child.GetComponent<Renderer>();
                    renderer.enabled = isActive;
                }
            }
        }
    }

    public void ChangeColorToTurquoise(GameObject shape)
    {
```

```
        // Verifica si el GameObject y su Renderer están presentes
        if (shape != null && shape.GetComponent<Renderer>() != null)
        {
            // Cambia el color del material del primer Renderer encontrado
             shape.GetComponent<Renderer>().material.color = new Color(64f
/ 255f, 224f / 255f, 208f / 255f);
        }
        else
        {
            Debug.LogError("El objeto specialPoint no existe o no tiene un
componente Renderer.");
        }
    }

    public void ChangeColorToGreen(GameObject shape)
    {
        // Verifica si el GameObject y su Renderer están presentes
        if (shape != null && shape.GetComponent<Renderer>() != null)
        {
            // Cambia el color del material del primer Renderer encontrado
             shape.GetComponent<Renderer>().material.color = new Color(0f,
1f, 0f);
        }
        else
        {
            Debug.LogError("El objeto specialPoint no existe o no tiene un
componente Renderer.");
        }
    }
}
```

10.16 GROUPCREATION

```
using AX.Generators;
using System.Collections;
using System.Collections.Generic;
using UnityEditor.VersionControl;
using UnityEngine;

public class GroupCreation : MonoBehaviour
{
    private GameObject initialText;  //Texto inicial en el canvas
```

```
    private GameObject finalText;    //Texto una vez que se ha activado la
función de creación de grupos
    public bool isGrouping = false;
    private GameObject groupParent; // Padre de todos los objetos agrupados
(usado para hacer el ajuste de posición)
     private GameObject groupParentNew; // Padre de todos los objetos
agrupados definitivo (después de hacer el ajuste de posición)
    public GameObject toroidePrefab;   //Prefab de los ejes de rotación
    public GameObject grabAxisXPrefab;  //Prefab de los ejes de traslación
     public GameObject extrusionSphere;    //Prefab de las esferas de
redimensionad
       public Material transparentMaterial;    //Material modificado
transparente
    private static int groupCounter = 0; // Contador estático para el número
de grupos (usado para cambiar su nombre)
    public Vector3 objectPos = Vector3.zero;


    private void Awake()
    {
        //Busca los objetos con los textos al principio de todo
                        initialText     =     FindInChildren(transform,
"Text_MakeGroup").gameObject;
                         finalText      =     FindInChildren(transform,
"Text_ClickToFinish").gameObject;
        initialText.SetActive(true); // Cambia el texto
        finalText.SetActive(false);
    }
    private void OnEnable()
    {
        EventHub.OnGroupObjectsCanvas += Alternate;
        EventHub.OnSelectedGroupObjects += MakeGroup;
    }

    private void OnDisable()
    {
        EventHub.OnGroupObjectsCanvas -= MakeGroup;
        EventHub.OnSelectedGroupObjects -= Alternate;
    }

    void MakeGroup(RaycastHit hit)
    {
                if    (isGrouping    &&    hit.collider    !=    null    &&
hit.collider.gameObject.activeSelf)
        {
            // Asigna el objeto golpeado como hijo del padre del grupo
```

```
            //SpecialPointsActivation(hit.collider.gameObject);
             hit.collider.gameObject.transform.SetParent(groupParent.trans
form);
        }
    }


    void Alternate(RaycastHit hit)
    {
        if(isGrouping)
        {
            isGrouping = false;
            initialText.SetActive(true);
            finalText.SetActive(false);
            EventHub.ResetActivation();

            // Añade MeshFilter y MeshRenderer al grupo padre
                                    MeshFilter     parentMeshFilter     =
groupParent.AddComponent<MeshFilter>();
                                  MeshRenderer     parentMeshRenderer     =
groupParent.AddComponent<MeshRenderer>();

            // Establece un material para el MeshRenderer
            parentMeshRenderer.material = transparentMaterial;

            //Oculta la visualización de los puntos característicos
            SpecialPointsDeactivation(groupParent);
            // Llama a la función para combinar las mallas de los hijos
            CombineChildMeshes(groupParent, parentMeshFilter);

           //Calcula la posición del centro del padre al origen para hacer
el truco de la posición
            Mesh mesh = parentMeshFilter.mesh;
            objectPos = mesh.bounds.center;
            //SpecialPointsActivation(groupParent);
            groupParent.transform.position -= objectPos;

            // Mover todos los hijos del oldParent al newParent
            while (groupParent.transform.childCount > 0)
            {
                Transform child = groupParent.transform.GetChild(0);
                child.SetParent(groupParentNew.transform, true);
            }

            // Añade MeshFilter y MeshRenderer al grupo padre
```

```
                                    MeshFilter     newParentMeshFilter     =
groupParentNew.AddComponent<MeshFilter>();
                                  MeshRenderer     newParentMeshRenderer     =
groupParentNew.AddComponent<MeshRenderer>();

            //Oculta la visualización de los puntos característicos
            SpecialPointsDeactivation(groupParentNew);

            CombineChildMeshes(groupParentNew, newParentMeshFilter);
            Destroy(groupParent);

            // Establece un material para el MeshRenderer
            newParentMeshRenderer.material = transparentMaterial;
            groupParentNew.transform.position = objectPos;
            //SpecialPointsActivation(groupParentNew);

            //Añade los scripts al nuevo objeto creado
                                        var     posicionadorCilindros      =
groupParentNew.AddComponent<PosicionarCilindros>();
               posicionadorCilindros.cilindroPrefab = grabAxisXPrefab; //
Asigna el prefab de grabAxisX

                                           var      rotationalAxis      =
groupParentNew.AddComponent<RotationalAxis>();
              rotationalAxis.toroidePrefab = toroidePrefab; // Asigna el
prefab de toroide

                                         var      ExtrusionSpheres      =
groupParentNew.AddComponent<ExtrusionSpheres>();
            ExtrusionSpheres.sphere = extrusionSphere; // Asigna el prefab
de las esferas de extrusión

                                           var      modoControlador      =
groupParentNew.AddComponent<ModoControlador>(); //Esta línea importa
           groupParentNew.tag = "PrimaryObject"; // Asigna el tag al objeto

                                       MeshCollider     meshCollider     =
groupParentNew.AddComponent<MeshCollider>();
            meshCollider.convex = true;

            //Añade el empty scale al nuevo objeto creado

            GameObject emptyScale = new GameObject("EmptyScale");
            emptyScale.transform.SetParent(groupParentNew.transform);
            emptyScale.transform.localPosition = Vector3.zero;
            emptyScale.transform.localRotation = Quaternion.identity;
```

```
            emptyScale.transform.localScale = Vector3.one; // Asegura que
la escala sea neutra; ajusta seg□n sea necesario
              //Destruyo los hijos para que no molesten, pero no quiero
destruir los specialpoints
           // Primero, mover todos los 'SpecialPoints' a ser hijos directos
de 'groupParentNew'
            // Lista para guardar temporalmente los SpecialPoints
            List<Transform> specialPointsToMove = new List<Transform>();

            // Buscar todos los SpecialPoints
            foreach (Transform child in groupParentNew.transform)
            {
                foreach (Transform grandchild in child)
                {
                    if (grandchild.CompareTag("SpecialPoint"))
                    {
                        specialPointsToMove.Add(grandchild);
                    }
                }
            }

            // Mover los SpecialPoints recopilados a ser hijos directos de
groupParentNew
            foreach (Transform specialPoint in specialPointsToMove)
            {
                specialPoint.SetParent(groupParentNew.transform, true);
            }


            // Luego, destruir todos los hijos de 'groupParentNew' que no
sean 'SpecialPoints'
            foreach (Transform child in groupParentNew.transform)
            {
                  if (!child.CompareTag("SpecialPoint") && child.name !=
"EmptyScale") //Hay que asegurarse de que no se destruyen los puntos
característicos
                {
                    Destroy(child.gameObject);
                }
            }

            var cotas = groupParentNew.AddComponent<TextoCotas>();
             cotas.CreateCanvasText(groupParentNew, emptyScale);  //Añade
las cotas y el texto por primera vez
```

```
        }
        else
        {
            isGrouping = true;
            EventHub.ResetActivation();
            initialText.SetActive(false); // Oculta el objeto
            finalText.SetActive(true);
           CreateNewGroupParent(); // Crear un nuevo padre para el próximo
grupo que se cree
        }
    }

    private void CreateNewGroupParent()
    {
        // Incrementa el contador cada vez que se crea un nuevo grupo
        groupCounter++;

         // Crea un nuevo objeto padre para los objetos agrupados y añade
el contador al nombre
        groupParent = new GameObject("GroupParent_" + groupCounter);
        groupParentNew = new GameObject("GroupParent_new" + groupCounter);
    }

       private  void  CombineChildMeshes(GameObject  parent,  MeshFilter
parentMeshFilter)
    {
         //Método que se encarga de combinar las mallas de los hijos para
luego asignar la malla conjunta al objeto padre
        // Consigue todos los MeshFilters de los hijos (asegurarse de que
solo tomas los hijos directos)
                                    MeshFilter[]        meshFilters        =
parent.GetComponentsInChildren<MeshFilter>(false);

        CombineInstance[] combine = new CombineInstance[meshFilters.Length
- 1]; // -1 para excluir el MeshFilter del padre
        int i = 0;

        foreach (var meshFilter in meshFilters)
        {
              if (meshFilter.transform == parent.transform) continue; //
Excluye el MeshFilter del propio parent

            combine[i].mesh = meshFilter.sharedMesh;
           combine[i].transform = meshFilter.transform.localToWorldMatrix;
            i++;
        }
```

```
        // Combina todas las mallas de los hijos en una sola malla
        Mesh combinedMesh = new Mesh();
        combinedMesh.CombineMeshes(combine);

        // Asigna la malla combinada al MeshFilter del padre
        parentMeshFilter.mesh = combinedMesh;
    }

    void SpecialPointsDeactivation(GameObject parent)
    {
        //Desactivar los puntos especiales para que no combinen sus mesh
        foreach (Transform child in parent.transform)
        {
            foreach (Transform grandchild in child.transform)
            {
                if (grandchild.CompareTag("SpecialPoint"))
                {
                      // Hacer que el 'SpecialPoint' sea hijo directo de
'groupParentNew'
                    grandchild.gameObject.SetActive(false);
                }
            }
        }
    }

    void SpecialPointsActivation(GameObject parent)
    {
        //Vuelvo a activar a los special points
        //Desactivar los objetos especiales para que no combinen sus mesh
        foreach (Transform child in parent.transform)
        {
            foreach(Transform grandchild in child.transform)
            {
                if (grandchild.CompareTag("SpecialPoint"))
                {
                      // Hacer que el 'SpecialPoint' sea hijo directo de
'groupParentNew'
                    grandchild.gameObject.SetActive(true);
                }
            }
        }
    }

    Transform FindInChildren(Transform parent, string name)
    {
```

```
        //Busca los hijos que tengan un determinado nombre y devuelve los
hijos
        foreach (Transform child in parent)
        {
            if (child.name == name)
                return child;

            Transform result = FindInChildren(child, name);
            if (result != null)
                return result;
        }
        return null;
    }
}
```

10.17 TEXTOCOTAS

```
using AX.Generators;
using Unity.VisualScripting;
using UnityEngine;
using UnityEngine.UI;

public class TextoCotas : MonoBehaviour
{
    private void OnEnable()
    {
        EventHub.OnActualizarCotas += UpdateTexts;
    }

    private void OnDisable()
    {
        EventHub.OnActualizarCotas -= UpdateTexts;
    }

    public void CreateCanvasText(GameObject shape, GameObject emptyScale)
    {
         bool firstCreation = true;  //Variable que determina si es la
primera vez que se añaden las cotas a esa pieza

        Canvas canvas = null;
        if (canvas == null)
        {
            // Crea y configura un nuevo Canvas si no se ha asignado uno
```

```
            GameObject canvasObj = new GameObject("Dimension Canvas");
            canvasObj.tag = "CanvasCotas";
            canvas = canvasObj.AddComponent<Canvas>();
            canvas.transform.SetParent(emptyScale.transform);
            canvas.renderMode = RenderMode.WorldSpace;
            CanvasScaler scaler = canvasObj.AddComponent<CanvasScaler>();
             scaler.dynamicPixelsPerUnit = 10;  // Ajustar esto según las
necesidades de visualización
            canvasObj.AddComponent<GraphicRaycaster>();

            // Configura tamaño y posición del Canvas
            RectTransform rect = canvas.GetComponent<RectTransform>();
            rect.sizeDelta = new Vector2(50, 50);  // Tamaño del canvas en
el mundo
            rect.position = shape.transform.position;
        }

        UpdateTexts(shape.transform, canvas, firstCreation);
    }

    void UpdateTexts(Transform pieza, Canvas canvas, bool firstCreation)
    {
        MeshFilter meshFilter = pieza.GetComponent<MeshFilter>();
        if (meshFilter != null)
        {
            // Elimina textos antiguos si existen
            foreach (Transform child in canvas.transform)
            {
                Destroy(child.gameObject);
            }

            Mesh mesh = meshFilter.mesh;
            Vector3 size = mesh.bounds.size;
            Vector3 scale = pieza.localScale;
              Vector3 localSize = new Vector3(size.x * scale.x, size.y *
scale.y , size.z *scale.z );

            // Crea textos actualizados en relación a los ejes locales de
la pieza
                CreateText(localSize.x, "Ancho", pieza.TransformPoint(new
Vector3(0,    size.y    /    2,    -size.z    /    2)),    pieza.rotation    *
Quaternion.LookRotation(Vector3.forward),       localSize.x,       canvas,
firstCreation, Color.red);
                CreateText(localSize.z, "Largo", pieza.TransformPoint(new
Vector3(size.x    /    2,    size.y    /    2,    0)),    pieza.rotation    *
```

```
Quaternion.LookRotation(Vector3.left), localSize.z, canvas, firstCreation,
Color.blue);
                CreateText(localSize.y, "Altura", pieza.TransformPoint(new
Vector3(size.x      /      2,      0,      0)),      pieza.rotation      *
Quaternion.LookRotation(Vector3.forward),       localSize.y,       canvas,
firstCreation, Color.green);
        }
    }

      void  CreateText(float  dimension,  string  dimensionName,  Vector3
position, Quaternion rotation, float pieceDimension, Canvas canvas, bool
firstCreation, Color textColor)
    {
        GameObject textObj = new GameObject(dimensionName + " Text");
        textObj.transform.SetParent(canvas.transform);
        textObj.transform.position = position;
        textObj.transform.rotation = rotation;

        dimension = dimension * 1000;
        TextMesh text = textObj.AddComponent<TextMesh>();
       text.text = dimensionName + ": " + dimension.ToString("F2") + "mm";
        text.fontSize = 24;
        text.color = textColor;
        text.anchor = TextAnchor.MiddleCenter;

         // Fijar el tamañoo del texto a un ancho fijo proporcional a la
dimensión
         float targetWidth = 0.19f * dimension / 1000;  // Ancho fijo en
metros
         float textLength = text.text.Length * text.fontSize * 0.01f; //
Estimación del largo del texto
          float scale = targetWidth / textLength; // Calcula la escala
necesaria para mantener el texto dentro del ancho deseado
        text.transform.localScale = Vector3.one * scale;

        if (firstCreation)
        {
            //Quito la visualizacion de las cotas solo si es la primera vez
que se crean
             VisualizacionCotasOff();
        }
    }


    void VisualizacionCotasOff()
    {
```

```
        //Método que quita la visualización de las cotas
        // Busca todos los GameObjects con la etiqueta "CanvasCotas"
                                    GameObject[]       canvasObjects       =
GameObject.FindGameObjectsWithTag("CanvasCotas");

        if (canvasObjects.Length == 0)
        {
                  Debug.Log("No  se  encontraron  Canvas  con  la  etiqueta
'CanvasCotas'");
        }
        else
        {
              // Itera sobre cada GameObject encontrado con la etiqueta
"CanvasCotas"
            foreach (GameObject canvasObject in canvasObjects)
            {
                // Desactiva cada hijo del GameObject
                foreach (Transform child in canvasObject.transform)
                {
                    child.gameObject.SetActive(false);
                }
            }
        }
    }
}
```

10.18 COTASCANVAS

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

public class CotasCanvas : MonoBehaviour
{
    public bool cotasActivas = false;
    private GameObject initialText;
    private GameObject finalText;

    private void OnEnable()
    {
        EventHub.OnCotasCanvas += Alternate; // Se suscribe al evento
    }
```

```
    private void OnDisable()
    {
        EventHub.OnCotasCanvas -= Alternate;
    }


    public void Awake()
    {
        // Comprueba si el nombre del GameObject es "Cotas_Canvas"
        if (transform.name == "Cotas_Canvas")
        {
                                initialText    =    FindInChildren(transform,
"Text_CotasOnn").gameObject;
                                 finalText    =    FindInChildren(transform,
"Text_CotasOff").gameObject;
            initialText.SetActive(true); // Oculta el objeto
            finalText.SetActive(false);
        }
        else
        {
            Debug.Log("No se llama asi");
        }
    }

    void Alternate(RaycastHit hit)
    {
        //Método que gestiona el encendido apagado de las cotas cuando se
golpea al canvas
        if (cotasActivas)
        {
            cotasActivas = false;
            initialText.SetActive(true);
            finalText.SetActive(false);
            EventHub.ResetActivation();
        }
        else
        {
            cotasActivas = true;
            UpdateText();
            initialText.SetActive(false);
            finalText.SetActive(true);
            EventHub.ResetActivation();
        }
    }

    Transform FindInChildren(Transform parent, string name)
```

```
    {
        //Devuelve los hijos de un objeto
        foreach (Transform child in parent)
        {
            if (child.name == name)
                return child;

            Transform result = FindInChildren(child, name);
            if (result != null)
                return result;
        }
        return null;
    }

    void UpdateText()
    {
       //Actualiza el texto por si la pieza ha sido redimensionada mientras las cotas estaban desactivadas

                            ModoControlador[]      allControladores      = FindObjectsOfType<ModoControlador>();

        // Itera sobre cada objeto y verifica si la variable isActive es true
        foreach (ModoControlador controlador in allControladores)
        {
            //if (controlador.activated)
            //{
                //Aquí tenemos todas las piezas que están activadas
                //Esto ajusta las cotas
                GameObject cotasCanvas = GameObject.Find("Cotas_Canvas");
                                    CotasCanvas    cotasCanvasScript    = cotasCanvas.GetComponent<CotasCanvas>();

                if (cotasCanvasScript.cotasActivas)
                {
                                                    Canvas    canvas    = controlador.transform.GetComponentInChildren<Canvas>();
                   EventHub.ActualizarCotas(controlador.transform, canvas, false);
                }

            //}
        }
    }
}
```

10.19 KEYBOARD

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.EventSystems;
using TMPro;

public class Keyboard : MonoBehaviour
{
    string buttonText;            //Texto de los botones, para saber qué
valor representa cada uno
    TextMeshProUGUI localText;  //Valor que hay en el texto en la casilla
que el jugador quiere cambiar
    RaycastHit lastObjectHit;   // Último objeto golpeado por el raycast
        [HideInInspector]   public   bool   exactPositioningActive   =
false;   //Atributo que controla si la función está activa o no
    [HideInInspector] public bool objectSelected = false;
    Vector3 oldScale; //Para aplicar el ajuste de escala de las esferas de
extrusión (así aprovecho una función que ya hice)
    Vector3 newScale; //Nueva escala para cambiar el objeto



    private void OnEnable()
    {
        EventHub.OnPositionCanvas += GetHitInfo; //Coge la informacion del
cuadro de texto que se golpeo
        EventHub.OnGetObjectPosition += UpdateHitObjectInfo;
    }

    private void OnDisable()
    {
        EventHub.OnPositionCanvas -= GetHitInfo;
        EventHub.OnGetObjectPosition -= UpdateHitObjectInfo;
    }

    public void UpdateHitObjectInfo(RaycastHit hit)
    {
       //Después de golpear el objeto, actualizo el panel con su información
        // Obtener la posición y rotación del objeto impactado
        objectSelected = true;
        lastObjectHit = hit;
```

```
        Vector3 hitPosition = hit.collider.gameObject.transform.position;
                                        Vector3        hitRotation        =
hit.collider.gameObject.transform.eulerAngles;

        // Buscar y actualizar los textos correspondientes
        UpdateText("PosX_B", hitPosition.x);
        UpdateText("PosY_B", hitPosition.y);
        UpdateText("PosZ_B", hitPosition.z);
        UpdateText("RotX_B", hitRotation.x);
        UpdateText("RotY_B", hitRotation.y);
        UpdateText("RotZ_B", hitRotation.z);

        // Obtener el MeshFilter del objeto impactado
                                      MeshFilter        meshFilter        =
hit.collider.gameObject.GetComponent<MeshFilter>();
        if (meshFilter != null)
        {
            Mesh mesh = meshFilter.sharedMesh;

            // Obtener las dimensiones locales del mesh
            Vector3 meshSize = mesh.bounds.size;
            Vector3 scale = lastObjectHit.collider.transform.localScale;
            oldScale = scale;

             // Actualizar los textos correspondientes a las dimensiones
locales
            UpdateText("DimX_B", scale.x * meshSize.x );
            UpdateText("DimY_B", scale.y * meshSize.y );
            UpdateText("DimZ_B", scale.z * meshSize.z );
        }
        else
        {
            Debug.LogError("El objeto impactado no tiene un MeshFilter.");
        }
    }

    private void UpdateText(string objectName, float value)
    {
         //Actualiza el display con los datos del objeto ipactado por el
raycast
        GameObject obj = GameObject.Find(objectName);
        if (obj != null)
        {
                                    TextMeshProUGUI     textComponent     =
obj.GetComponentInChildren<TextMeshProUGUI>();
            if (textComponent != null)
```

```
            {
                textComponent.text = value.ToString("F3");
            }
            else
            {
                Debug.LogError("Se encontró el objeto, pero no el texto");
            }
        }
        else
        {
            Debug.LogError("No se encontro el objeto");
        }
    }

    public void EnterPressed()
    {
        //El usuario ha pulsado enter
        if (objectSelected && exactPositioningActive)
        {
            ApplyTextValuesToObject();

            //Esto ajusta las cotas
            GameObject cotasCanvas = GameObject.Find("Cotas_Canvas");
                                    CotasCanvas     cotasCanvasScript     =
cotasCanvas.GetComponent<CotasCanvas>();

            if (cotasCanvasScript.cotasActivas)
            {
                                                    Canvas      canvas      =
lastObjectHit.collider.transform.GetComponentInChildren<Canvas>();
                EventHub.ActualizarCotas(lastObjectHit.collider.transform,
canvas, false);
            }
        }
    }

    public void ApplyTextValuesToObject()
    {
        //Aplica al objeto los valores que están en el display (posición,
rotación y dimensión)
        if (lastObjectHit.collider != null)
        {
            // Leer los valores de los textos para posición y rotación
            float posX = ReadTextValue("PosX_B");
            float posY = ReadTextValue("PosY_B");
            float posZ = ReadTextValue("PosZ_B");
```

```
            float rotX = ReadTextValue("RotX_B");
            float rotY = ReadTextValue("RotY_B");
            float rotZ = ReadTextValue("RotZ_B");

            // Actualiza la posición y rotación del objeto impactado
               lastObjectHit.collider.gameObject.transform.position = new
Vector3(posX, posY, posZ);
             lastObjectHit.collider.gameObject.transform.eulerAngles = new
Vector3(rotX, rotY, rotZ);

            // Lee los valores de los textos para las dimensiones
            float dimX = ReadTextValue("DimX_B");
            float dimY = ReadTextValue("DimY_B");
            float dimZ = ReadTextValue("DimZ_B");

            // Obtener el MeshFilter del objeto impactado
                                        MeshFilter      meshFilter      =
lastObjectHit.collider.gameObject.GetComponent<MeshFilter>();
            if (meshFilter != null)
            {
                Mesh mesh = meshFilter.sharedMesh;

                 // Calcula el factor de escala necesario para las nuevas
dimensiones
                Vector3 meshSize = mesh.bounds.size;
                 Vector3 scaleFactor = new Vector3(dimX / meshSize.x, dimY
/ meshSize.y , dimZ / meshSize.z);

                // Aplica la escala al objeto impactado
                  lastObjectHit.collider.gameObject.transform.localScale =
scaleFactor;
                newScale = scaleFactor;
                RestoreChildrensScales(scaleFactor);//Ajusta el tamañoo de
las esferas de extrusión

                // Busca el hijo directo llamado "EmptyScale"
               //Cuando se redimensiona un objeto, también hay que ajustar
la escala del "emptyScale"
                                      Transform    emptyScaleTransform    =
lastObjectHit.collider.gameObject.transform.Find("EmptyScale");
                if (emptyScaleTransform != null)
                {
                    // Calcular el factor de escala inverso
                          Vector3 inverseScaleFactor = new Vector3(1 /
scaleFactor.x, 1 / scaleFactor.y, 1 / scaleFactor.z);
```

```
                    // Aplicar la escala inversa al objeto "EmptyScale"
                    emptyScaleTransform.localScale = inverseScaleFactor;
                }
                else
                {
                        Debug.LogError("No se encontró el objeto hijo
'EmptyScale'.");
                }
            }
            else
            {
                    Debug.LogError("El objeto impactado no tiene un
MeshFilter.");
            }
        }
    }

    private float ReadTextValue(string objectName)
    {
       //Lee los valores que están escritos en el canvas con la información
del objeto
        GameObject obj = GameObject.Find(objectName);
        if (obj != null)
        {
                                    TextMeshProUGUI     textComponent     =
obj.GetComponentInChildren<TextMeshProUGUI>();
            if (textComponent != null)
            {
                if (float.TryParse(textComponent.text, out float value))
                {
                    return value;
                }
                else
                {
                      Debug.LogError("No se pudo convertir el texto a un
float en: " + objectName);
                    return 0f;
                }
            }
            else
            {
                Debug.LogError("Se encontró el objeto, pero no el texto");
                return 0f;
            }
        }
        else
```

```
        {
            Debug.LogError("No se encontró el objeto");
            return 0f;
        }
    }


    public void StartPositioningPressed()
    {
        // Se inicia la opción para modificar de manera exacta al objeto
                                    GameObject        buttonObject        =
EventSystem.current.currentSelectedGameObject;
        Image buttonImage = buttonObject.GetComponent<Image>();

        if (exactPositioningActive)
        {
            ResetTexts(); //Resetea el valor del panel por un guión
            exactPositioningActive = false;
            objectSelected = false;
            buttonImage.color = Color.white;
            EventHub.ResetActivation();
        }
        else
        {
            exactPositioningActive = true;
            buttonImage.color = Color.green;
            EventHub.ResetActivation();
        }
    }

    public void OnButtonPress()
    {
        // Obtener el botón que disparó el evento
        Debug.Log("Estoy dentro de la funcion");
                                    GameObject        buttonObject        =
EventSystem.current.currentSelectedGameObject;

        if (buttonObject != null)
        {
            // Obtener el texto del botón
                                                        buttonText        =
buttonObject.GetComponentInChildren<TextMeshProUGUI>().text;

            Debug.Log("Botón presionado: " + buttonText);
            if (localText != null)
            {
```

```
                SumarNumero();
            }
        }
    }

    void GetHitInfo(RaycastHit hit)
    {
        //Consigue la informacion sobre donde ha golpeado el raycast para
solo cambiar el valor de ese recuadro
                                                    localText           =
hit.collider.gameObject.GetComponentInChildren<TextMeshProUGUI>();
        localText.text = "";

        if (localText == null)
        {
            Debug.LogError("TextMeshProUGUI  componente no encontrado en "
+ hit.collider.gameObject.name);
        }
        Debug.Log("Has golpeado campeon");

    }

    public void SumarNumero()
    {
        //Va añadiendo a un string los valores de los botones que se van
pulsando
        localText.text = localText.text + buttonText;
    }

    public void ResetValue()
    {
        //Resetea la posición y rotación
        if (lastObjectHit.collider != null)
        {
            // Leer los valores de los textos
            float posX = 0;
            float posY = 0;
            float posZ = 0;
            float rotX = 0;
            float rotY = 0;
            float rotZ = 0;

            // Actualizar la posición y rotación del objeto impactado
               lastObjectHit.collider.gameObject.transform.position = new
Vector3(posX, posY, posZ);
```

```
            lastObjectHit.collider.gameObject.transform.eulerAngles = new
Vector3(rotX, rotY, rotZ);

            UpdateHitObjectInfo(lastObjectHit);
        }
    }

    public void ResetTexts()
    {
        UpdateText("PosX_B", 0f);
        UpdateText("PosY_B", 0f);
        UpdateText("PosZ_B", 0f);
        UpdateText("RotX_B", 0);
        UpdateText("RotY_B", 0);
        UpdateText("RotZ_B", 0);
        UpdateText("DimX_B", 0);
        UpdateText("DimY_B", 0);
        UpdateText("DimZ_B", 0);
    }

    void RestoreChildrensScales(Vector3 scaleFactor)
    {
        //Los puntos característicos se devuelven a su escala original si
la pieza ha sido redimensionada
           Vector3 newScale = new Vector3(0.2f/scaleFactor.x, 0.2f /
scaleFactor.y, 0.2f / scaleFactor.z);
        foreach (Transform child in lastObjectHit.transform)
        {
            // Aplica la escala a todos los hijos
            child.localScale = newScale;
        }


    }

}
```

10.20 DUPLICATEOBJECTS

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

```
public class DuplicateObjects : MonoBehaviour
{
    void OnEnable()
    {
        EventHub.OnDuplicateObjectsHit += DuplicateObject;
    }

    void OnDisable()
    {
        EventHub.OnDuplicateObjectsHit -= DuplicateObject;
    }

    void DuplicateObject(RaycastHit hit)
    {
        //Método que se encarga de duplicar los objetos
        // Encuentra todos los objetos en la escena que tienen el script
ModoControlador
        ModoControlador[] objetos = FindObjectsOfType<ModoControlador>();

       // Revisa cada objeto, para ver si están activos. En caso afirmativo,
los duplica
        foreach (ModoControlador objeto in objetos)
        {
            // Si la variable 'activated' es true, destruye el objeto
            if (objeto.activated)
            {
                // Instanciar (duplicar) el objeto en la misma posición y
con la misma rotación

               GameObject objetoDuplicado = Instantiate(objeto.gameObject,
objeto.gameObject.transform.position,
objeto.gameObject.transform.rotation);

                // Se cambia el nombre del objeto duplicado
                        objetoDuplicado.name  =  objeto.gameObject.name  +
"_Duplicado";

                 //No se quiere duplicar los hijos: (ejes de traslación,
rotación y redimensionado).
                //Esos ejes ya se añadirán al nuevo objeto más adelante
                EliminarHijosInnecesarios(objetoDuplicado);
            }
        }
    }

    void EliminarHijosInnecesarios(GameObject objetoPadre)
```

```
    {
        //Elimina los hijos de dentro de Empty Scale, porque si no luego se quedan fijos
                                Transform       emptyScaleTransform       = objetoPadre.transform.Find("EmptyScale");
        if (emptyScaleTransform != null)
        {
            // Eliminar todos los hijos de "EmptyScale"
            foreach (Transform child in emptyScaleTransform)
            {
                Destroy(child.gameObject);
            }
        }
        else
        {
            Debug.LogWarning("'EmptyScale' no se encontró entre los hijos de " + objetoPadre.name);
        }
    }
}
```

10.21 SAVEOBJECT

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEditor.Formats.Fbx.Exporter;
using UnityEngine;

public class SaveObject : MonoBehaviour
{
                    [SerializeField]         string         path         = "C:\\Users\\Pedro\\OneDrive\\TFG\\Piezas_Exportadas";   //Ruta  donde  se exporta la pieza

    void OnEnable()
    {
        EventHub.OnSaveObjectsHit += ExportarPiezaComoFBX;
    }

    void OnDisable()
    {
        EventHub.OnSaveObjectsHit -= ExportarPiezaComoFBX;
```

```
    }

    public void ExportarPiezaComoFBX(RaycastHit hit)
    {
         // Encuentra todos los objetos en la escena que tengan el script
ModoControlador
        ModoControlador[] objetos = FindObjectsOfType<ModoControlador>();

        // Revisa cada objeto
        foreach (ModoControlador objeto in objetos)
        {
            // Si la variable 'activated' es true, destruye los hijos
            if (objeto.activated)
            {
               GameObject objetoDuplicado = Instantiate(objeto.gameObject,
objeto.gameObject.transform.position,
objeto.gameObject.transform.rotation);

                // Se cambia el nombre del objeto duplicado, aunque luego
se elimina
                        objetoDuplicado.name  =  objeto.gameObject.name  +
"_Duplicado";

               // Eliminar todos los hijos del objeto duplicado usando una
lista temporal para evitar problemas de enumeración
                List<GameObject> children = new List<GameObject>();
                foreach (Transform child in objetoDuplicado.transform)
                {
                    children.Add(child.gameObject);
                }
                 //Destruye los hijos del objeto duplicado porque solo se
quiere exportar la pieza en sí
                foreach (GameObject child in children)
                {
                    DestroyImmediate(child);
                }

                // Se asegura del que el directorio existe
                                            string    directoryPath    =
System.IO.Path.GetDirectoryName(path);
                if (!System.IO.Directory.Exists(directoryPath))
                {
                    System.IO.Directory.CreateDirectory(directoryPath);
                }

                // Exportar como FBX
```

```
                    string fullPath = System.IO.Path.Combine(path,
objetoDuplicado.name + ".fbx");
                ModelExporter.ExportObject(fullPath, objetoDuplicado);
                    Debug.Log(objetoDuplicado.name + " exportada a: " +
fullPath);

                // Destruir el objeto duplicado después de exportar
                DestroyImmediate(objetoDuplicado);

                // Resetear la activación
                EventHub.ResetActivation();
                break;
            }
        }
    }
}
```

10.22 CANVASCLICKANIMATION

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEditor.Formats.Fbx.Exporter;
using UnityEngine;

public class SaveObject : MonoBehaviour
{
                    [SerializeField]        string        path        =
"C:\\Users\\Pedro\\OneDrive\\TFG\\Piezas_Exportadas";   //Ruta  donde  se
exporta la pieza

    void OnEnable()
    {
        EventHub.OnSaveObjectsHit += ExportarPiezaComoFBX;
    }

    void OnDisable()
    {
        EventHub.OnSaveObjectsHit -= ExportarPiezaComoFBX;
    }

    public void ExportarPiezaComoFBX(RaycastHit hit)
    {
```

```
        // Encuentra todos los objetos en la escena que tengan el script ModoControlador
        ModoControlador[] objetos = FindObjectsOfType<ModoControlador>();

        // Revisa cada objeto
        foreach (ModoControlador objeto in objetos)
        {
            // Si la variable 'activated' es true, destruye los hijos
            if (objeto.activated)
            {
                GameObject objetoDuplicado = Instantiate(objeto.gameObject, objeto.gameObject.transform.position, objeto.gameObject.transform.rotation);

                // Se cambia el nombre del objeto duplicado, aunque luego se elimina
                objetoDuplicado.name = objeto.gameObject.name + "_Duplicado";

                // Eliminar todos los hijos del objeto duplicado usando una lista temporal para evitar problemas de enumeración
                List<GameObject> children = new List<GameObject>();
                foreach (Transform child in objetoDuplicado.transform)
                {
                    children.Add(child.gameObject);
                }
                //Destruye los hijos del objeto duplicado porque solo se quiere exportar la pieza en sí
                foreach (GameObject child in children)
                {
                    DestroyImmediate(child);
                }

                // Se asegura del que el directorio existe
                string directoryPath = System.IO.Path.GetDirectoryName(path);
                if (!System.IO.Directory.Exists(directoryPath))
                {
                    System.IO.Directory.CreateDirectory(directoryPath);
                }

                // Exportar como FBX
                string fullPath = System.IO.Path.Combine(path, objetoDuplicado.name + ".fbx");
                ModelExporter.ExportObject(fullPath, objetoDuplicado);
```

```
                    Debug.Log(objetoDuplicado.name + " exportada a: " +
fullPath);

                // Destruir el objeto duplicado después de exportar
                DestroyImmediate(objetoDuplicado);

                // Resetear la activación
                EventHub.ResetActivation();
                break;
            }
        }
    }
}
```