Trabajo Fin de Máster
Ingeniería Electrónica, Robótica y Automática

Diseño de gemelo digital del sistema de transporte de célula de fabricación flexible

Autor: Juan Gómez Jiménez
Tutor: Juan Manuel Escaño González
Cotutor: Adolfo J. Sánchez del Pozo Fernández

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Proyecto Fin de Máster
Ingeniería Electrónica, Robótica y Automática

Metodología para el desarrollo de gemelos digitales. Aplicación a célula de fabricación flexible.

Autor:
Juan Gómez Jiménez

Tutor:
Juan Manuel Escaño González
Cotutor:
Adolfo J. Sánchez del Pozo Fernández

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2021

Proyecto Fin de Carrera: Metodología para el desarrollo de gemelos digitales. Aplicación a célula de fabricación flexible.

Autor: Juan Gómez Jiménez

Tutor: Juan Manuel Escaño González

Cotutor: Adolfo J. Sánchez del Pozo Fernández

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

A todas las personas que me han ayudado a seguir siempre adelante. Como me enseñaste tú, abuela.

Agradecimientos

A mis maestros, a Juanma y Adolfo, por ayudarme a sacar adelante este proyecto con creces.

Al proyecto DENiM, por dejarme utilizar parte de mi trabajo en éste.

A mi familia, por apoyarme siempre sin falta.

Y a Dios, por todo.

Juan Gómez Jiménez

Alumno del Máster en Ingeniería Electrónica, Robótica y Automática

Sevilla, 2021

Resumen

En el presente documento se profundiza acerca de algunos de los pasos para la creación de un gemelo digital, apoyado en ejemplos referentes al desarrollo de un gemelo digital realizado utilizando la metodología propuesta, con la configuración de comunicaciones, mediante OPC UA.

Abstract

This document goes in depth about some of the steps for the creation of a digital twin, supported by examples of the development of a digital twin using the proposed methodology, with the communication configuration, using OPC UA.

Índice

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

En la actualidad, un aspecto que ha ganado bastante valor es la información. La información es un elemento que mueve hoy en día grandes y pequeñas empresas, y su correcta gestíon ha supuesto grandes avances en muchos aspectos.

El hecho de poder gestionar la máxima información acerca de un entorno hace que sea controlable, predecible, ya que se puede cuantificar de alguna forma.

A continuación se va a introducir un término el cual va a liderar este documento. Dicho término es gemelo.

Si se busca el significado de este concepto en la RAE, y seleccionando la acepción referida a una cosa, se obtiene la siguiente definición [1]:

Igual que otra con la que normalmente forma pareja.

Por tanto, este término tiene mucha cohesión con la palabra *igual.* Esto quiere decir que comparte las mismas características, siendo difícil discernir entre el objeto original y su gemelo.

Sin embargo, ¿qué relación existe entre el término gemelo y la información? Abundando un poco en esta idea, si se consiguiera un gemelo de un sistema cualquiera, significaría que se habrían obtenido todas las características que aportan dicha información que se pretende conseguir.

Suponiendo un sistema muy complejo con infinidad de variables, si se deseara obtener y poder gestionar información de algún aspecto concreto de ese sistema, el hecho de generar un gemelo que compartiera únicamente las características que interesan para la información a tratar, se habría conseguido, de alguna manera, simplificar ese sistema complejo, haciendo más fácil la gestión de esa información.

Sin embargo, yendo un poco más allá, si el desarrollo de dicho modelo se realizara en un entorno totalmente controlable, del cual se pudieran registrar todas las variables deseadas, la gestión de la información quedaría resuelta.

Este concepto no se trata de una idea abstracta, sino algo que ya existe y de lo que se va a tratar en este trabajo, y se conoce como gemelo digital.

Existen multitud de definiciones acerca de este concepto, entre las cuales se puede mencionar, por ejemplo, aquella dada por la Universidad Internacional de Valencia, que lo explica de la siguiente manera: [2]

Es una "copia" digital de un elemento físico. Hoy en día se ha digitalizado el proceso para hacer pruebas.

Esta definición es bastante generalista y aporta poca información al lector, a la par que ofrece poca precisión. Es por ello que puede refinarse un poco junto a esta otra:

Un gemelo digital se trata modelo de un sistema real con una precisión suficiente que permita predecir cualquier comportamiento del sistema real con relevancia mínima, en un espacio y tiempo determinado.

Es imprtante tener en cuenta que esta relevancia mínima se refiere a que está relacionada con el objeto de estudio de dicho sistema, del cual se pretende hallar la máxima información posible.

El proyecto que se desarrolla a continuación se enmarca en las tareas del proyecto Europeo DENiM [3] cuyo objetivo se basa en la supervisión y optimización automática de la energía en cualquier proceso industrial mediante el uso de gemelos digitales.

2 EL MÉTODO

Finalizada la introducción de este documento, a continuación se muestra el objetivo principal del mismo. En este caso, se trata de desarrolloar un método de creación un gemelo digital de un sistema real.

El método está formado por 6 pasos bien diferenciados, que serían los siguientes:

1. Arquitectura interna
2. Modelado visual
3. Modelado funcional
4. Autoaprendizaje
5. Conectividad
6. Servicios

En el presente documento se centra en el desarrollo de los pasos 2 y 3 del método, icluyendo parte del paso 4. El resto de pasos se encuentran en proceso de desarrollo por parte de otros miembros del equipo propio del DENiM.

3 MODELADO VISUAL DEL PROCESO

En este apartado se va a hablar acerca del desarrollo de los modelos 3D correspondientes a cada uno de los elementos que componen la planta industrial de la cual se pretende generar el gemelo digital.

El hecho de la fidelidad visual entre gemelo y planta real es un factor bastante importante, que facilita la interacción entre simulador y usuario. Por otro lado, en ocasiones la geometría de un dispositivo puede ser causante de ciertos efectos dinámicos cruciales para el comportamiento del objeto.

Es por ello que, a la hora de generar el modelo 3D del dispositivo, es importante tener claro el nivel de similitud que se quiere llegar a obtener en el gemelo digital, y así poder discernir qué elementos son importantes y cuales no, no solo a nivel visual, sino en todos los aspectos, cinemático, dinámico, etc.

En este proyecto, se ha elegido Cinema 4D, un software dedicado al modelado 3D, animación y renderizado, que ha ganado importancia e interés en los últimos años en el ámbito del diseño y el *marketing* debido a su fácil aprendizaje y sus numerosas ventajas, destacando sus características para *motion graphics*, es decir, para la generación y animación de objetos.

Su gran popularidad también hace que exista una gran bibliografía en internet, así como multitud de ejemplos y explicaciones de proyectos.

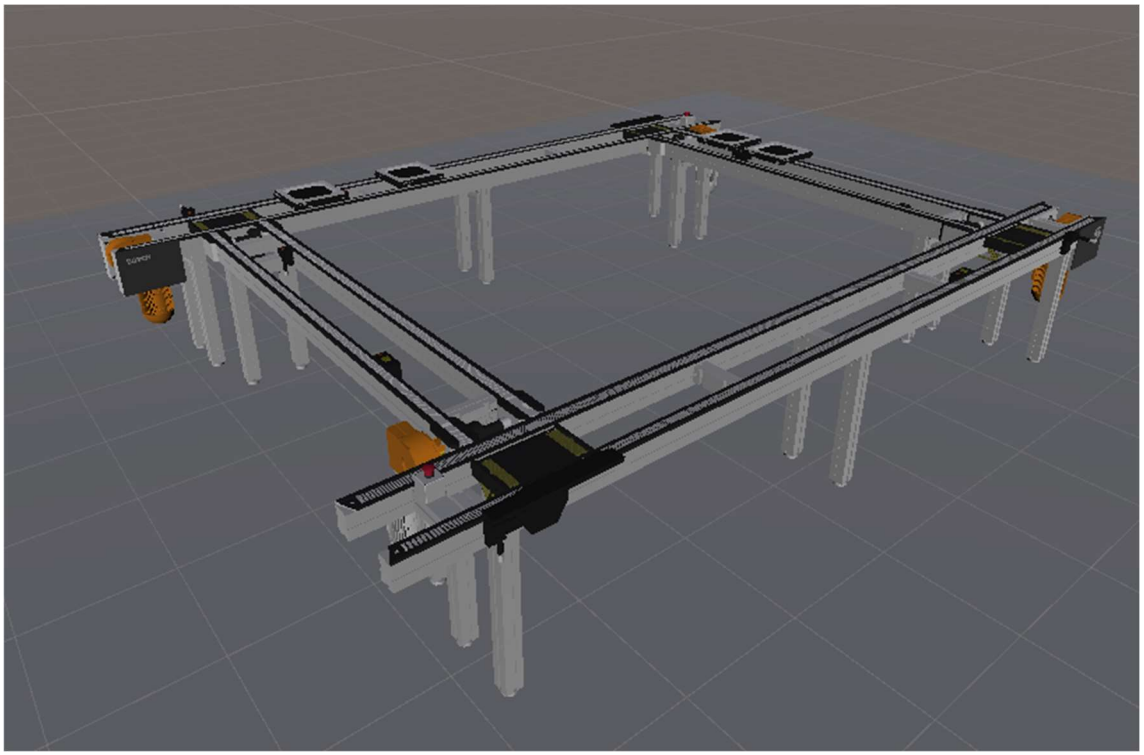


Figura 1. Modelo 3D del gemelo digital

El modelo a realizar de la planta se compone de 2 cintas largas, 2 cintas cortas, 4 pistones de cambio y un modelo de plataforma. Estos podrían denominarse elementos principales de la planta. Además, dentro de los modelos de las cintas se encuentran otros dispositivos, como son los pistones de paro, los sensores inductivos o los sensores de fuerza, elementos que también influyen en el comportamiento de la planta, y por último existen otros elementos auxiliares que no afectan en dicho comportamiento, cuyo objetivo es meramente visual, para asemejar el conjunto a la realidad, como es el caso de los motores de cada una de las cintas. Sin embargo, estos elementos podrían programarse en un futuro, y modelar su comportamiento si fuera necesario.

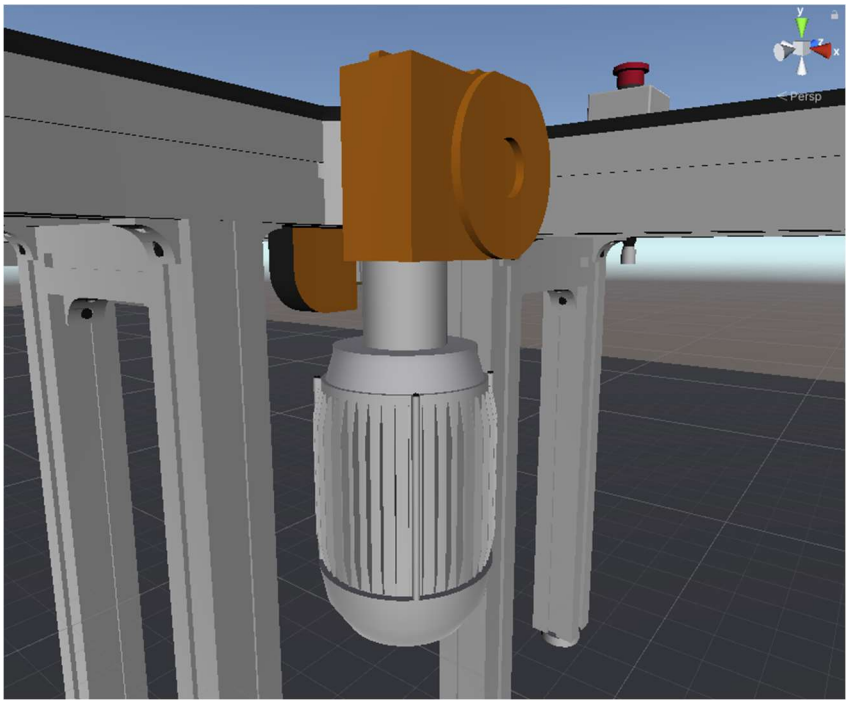


Figura 2. Modelo 3D del motor de la cinta transportadora corta.

En cuanto a la creación de cada modelo, se ha realizado a un gran nivel de detalle, teniendo en cuenta biseles en terminaciones de piezas, tornillos, esquinas redondeadas, muescas preparadas para acoplar otras piezas, o marcas de fabricante, ya que estos detalles ayudan a que, en un futuro, el usuario encuentre más realista el entorno virtual y pueda trabajar con la abstracción mínima posible.

Sin embargo, a efectos dinámicos y cinemáticos, este tipo de detalles que no afectan para nada en la interacción con otros objetos, sino que tiene una finalidad meramente visual.

Por otro lado, se da el caso de elementos que disponen de una estructura concreta que define su comportamiento, y en este caso se ha tratado de modelar de la forma más fiel posible, para conseguir que el propio motor físico de Unity 3D, generando los efectos de la gravedad, genere el comportamiento deseado sobre dicho elemento, sin tener que recurrir a programación adicional. Este es el caso, por ejemplo, del sensor de fuerza, del que se hablará más adelante.

En cuanto a la creación y exportación de animaciones de Cinema 4D a Unity 3D se encontró un pequeño obstáculo. Si se crea una animación en algún objeto en Cinema 4D y dicho modelo se exporta Unity 3D, dicha animación quedará linkada al modelo completo exportado. Esto supone dos cosas: en primer lugar, para gestionar dicha animación, se debe crear un componente de tipo animator e incluirlo en el inspector del elemento padre de ese modelo 3D, y no en el objeto sobre el que aplica la animación. Por otro lado, una vez exportado a Unity, no se puede alterar la estructura de objetos que dispone dicho modelo, eliminando o cambiando la jerarquía de alguna pieza de dicho modelo, ya que eso hará que la animación deje de funcionar.

Para ver estas ideas de forma más clara, mirando la cinta transportadora, se trata de un elemento dispone únicamente de una animación, que es el movimiento de la cadena propia de la cinta. Se ha generado una animación de este movimiento ya que se trata de un movimiento complejo para programarlo. Teniendo en cuenta esta animación, el objeto involucrado, es decir, el que se mueve, se trata únicamente de la cadena de la cinta, pero el resto de la cinta permanece inmóvil. Sin embargo, por la regla mencionada anteriormente, si se exporta toda la cinta como un único modelo 3D a Unity 3D, a la hora de crear el componente ***animator controller***, éste debe incluirse en el objeto padre de la cinta, que en este caso se llama LongBeltConveyor1. Este hecho es bastante poco intuitivo, pero a tener en cuenta para que la animación funcione correctamente.

Una vez realizada esta pequeña introducción relacionada con el modelado 3D, a continuación, se van a mostrar algunas de las técnicas más relevantes y mayormente empleadas para modelar todo el conjunto de elementos que conforman la planta en cuestión. En muchas ocasiones, es posible usar varias técnicas para conseguir el mismo resultado, es por ello que existe gran flexibilidad para modelar en 3D con este software.

3.1. Selección y desplazamiento

Esta primera técnica es quizás la más usada a la hora de modelar un objeto, ya que es muy sencilla y ofrece mucha precisión a la par que una gran versatilidad para editar en un volumen o área muy concreto.

Para poder aplicar esta técnica a un volumen determinado, es importante que este objeto esté configurado como editable. Para ello, basta con seleccionar el objeto, y pulsar el botón ***Hacer editable*** situado en la parte superior de la barra de herramientas lateral, según se muestra en la imagen 3

Figura 3. Herramienta para hacer un objeto editable.

El primer paso se basa en la selección de la parte del objeto a modificar, que puede ser un vértice, una arista o un plano del mismo. Para este proceso de **selección**, Cinema 4D ofrece varias herramientas de selección que facilitan mucho esta acción. En ocasiones también puede darse que la selección deseada puede estar formada por puntos que no son vértices del objeto. Esto no es problema, ya que también existe la opción de crear nuevos puntos en el objeto.

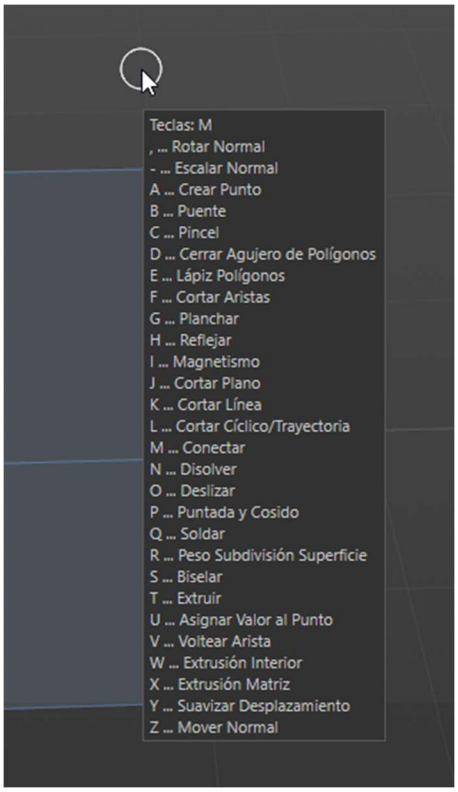


Figura 4. Menú rápido de herramientas de creación y edición, pulsando la tecla M.

Para ello, si se pulsa la tecla M se despliega una lista con una serie de herramientas de tipo **creación y edición** (figura 4). Se trata de un menú de opciones configuradas como atajos de teclado. Dentro de estas opciones se encuentra la opción **crear punto**, en la letra A. Al seleccionar esta herramienta, pulsando en cualquier parte del objeto editable sobre el que se está trabajando, se creará un nuevo punto, como si fuera un vértice más del objeto. De la misma manera, pulsando el comando M+K se selecciona la herramienta Cortar Línea, la cual permite crear una nueva arista a partir de la selección de dos vértices del objeto. Con estas dos herramientas es posible crear cualquier área de selección. Por último, si se quieren seleccionar varios puntos, aristas o planos a la vez, basta con seleccionarlos manteniento pulsada la tecla SHIFT.

Una vez hecha la selección apropiada, seleccionando la herramienta ***mover***, es el momento de realizar el **desplazamiento** de la parte seleccionada, manteniendo el resto del objeto constante, consiguiendo, de forma sencilla e intuitiva, modificar la geometría de la pieza. Para realizar este paso de forma más precisa, puede resultar más sencillo cambiar la vista 3D por alguna de las vistas 2D de cada uno de los 3 ejes principales, aquella que sea perpendicular al desplazamiento.

Como ejemplo, se va a crear el objeto mostrado en la figura 5 a partir de un cubo.

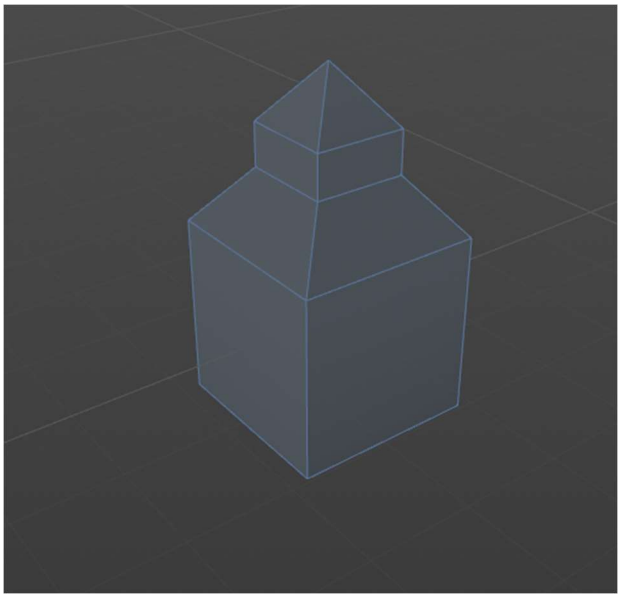


Figura 5. Ejemplo de uso de la técnica Selecćon y desplazamiento.

Para ello, crear un cubo, y en primer lugar hacerlo editable con la herramienta descrita anteriormente. En este momento, seleccionando cada una de las caras del cubo y moviéndolas se modificaría de forma sencilla las dimensiones del prisma, aunque en este caso se va a mantener en formato cubo.

El siguiente paso es crear el punto que hará las veces de vértice de la pirámide supertior. Para ello, con el comando M+A se selecciona la herramienta Crear punto, y con ella, se elige algún punto, simplemente haciendo click en alguna zona central de la cara superior. Al hacer esto, se observa que, junto con el punto, se crean 4 aristas que unen dicho punto con cada uno de los vértices superiores (figura 6).

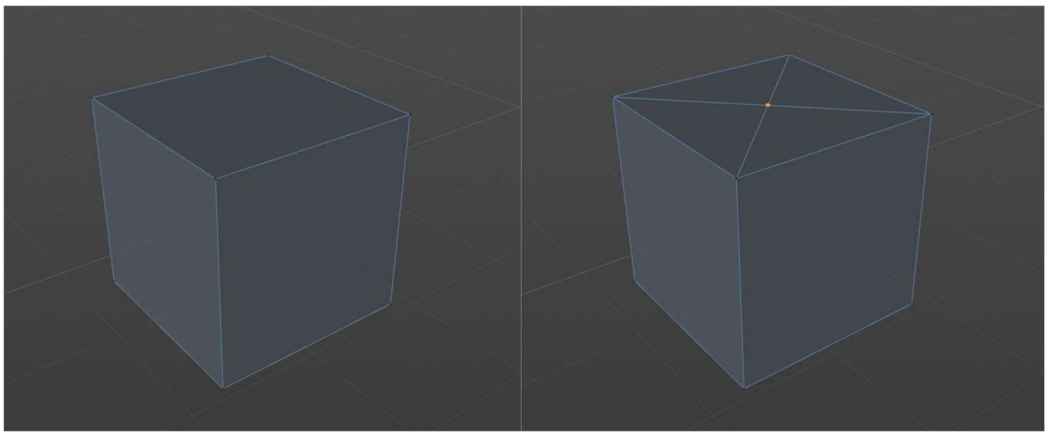


Figura 6. a) Creación de un cubo y b) Creación de vértice en la cara superior, creando de forma automática las 4 aristas que unen dicho vértice con los 4 superiores ya existentes

Una vez creado el vértice superior, seleccionando dicho punto y moviéndolo en la dirección del eje y, se formará una primera pirámide. A continuación, es el momento de crear la fase intermedia. Para ello se usa la herramienta Cortar línea (M+K). Además, este paso se hará de manera mucho más sencilla activando la herramienta fijación (figura 7), que ayuda a seleccionar puntos singulares, como intersecciones con otras aristas.

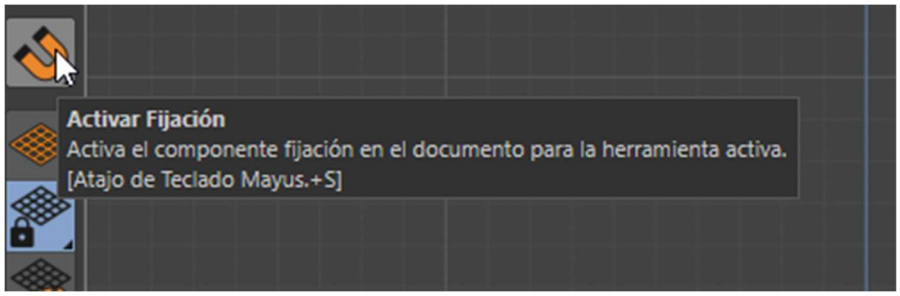


Figura 7. Herramienta fijación.

Por tanto, activando ambas herramientas, se selecciona un punto intermedio de una de las 4 nuevas aristas, y manteniendo presionada la tecla SHIFT, para crear rectas perpendiculares, se forma un primer cuadrado intermedio (figura 8). Este proceso es más sencillo hacerlo en la vista de alzado (vista superior).

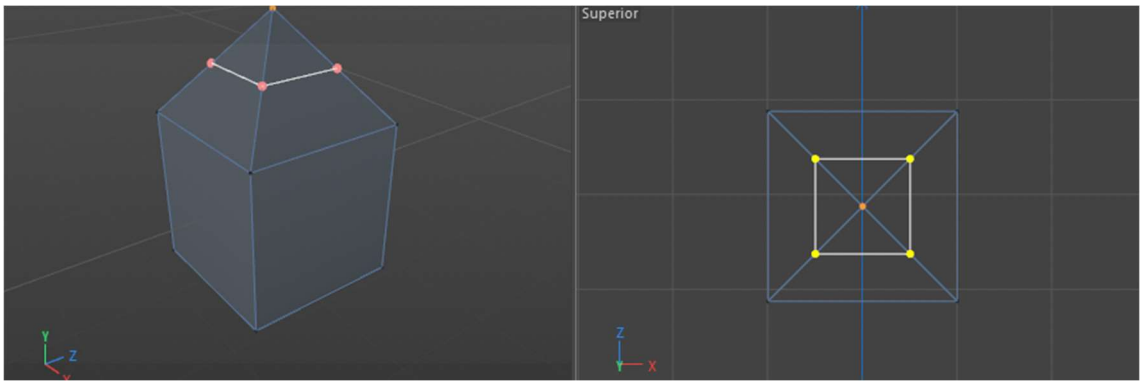


Figura 8. Creación de cuadrado intermedio mediante la herramienta Cortar línea.

Por último, falta la creación de el escalón intermedio que separa la pirámide superior de la inferior. Esto es muy sencillo, y para hacerlo, seleccionando los 4 planos de la pirámide superior, si se desplaza hacia arriba manteniendo pulsada la tecla Ctrl, a diferencia del desplazamiento realizado anteriormente, en este caso no se modifica la parte no seleccionada, la parte seleccionada se desplaza, y los huecos que se deberían aparecer se rellenan con nuevos planos que se generan automáticamente. Estas son dos opciones que se usan dependiendo de la ocasión (presionando o no la tecla Ctrl). Este último paso hace que la pieza quede completamente formada (figura 9).

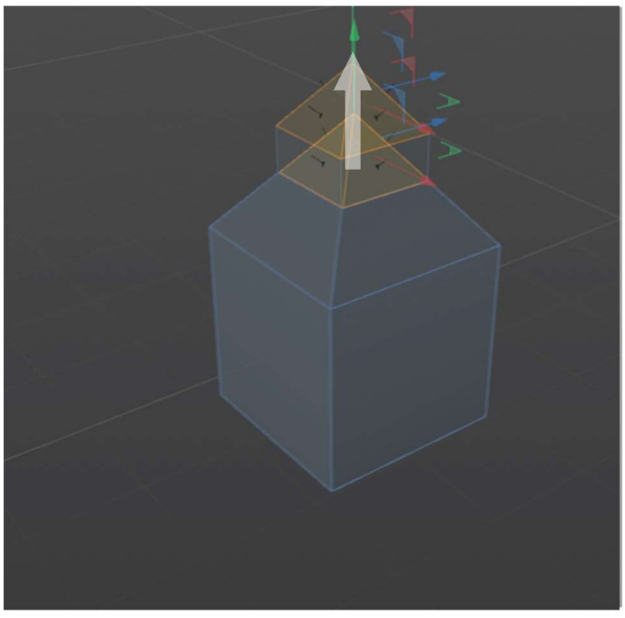


Figura 9. Creación escalón intermedio y obtención pieza final.

Esta técnica, así como puede usarse para extraer hacia afuera parte del volumen generando “picos”, de la forma inversa pueden generarse “huecos”, con la forma que se defina en el proceso de selección.

3.2. Combinación

Esta es otra técnica muy usada a la hora de modelar, y en conctreto en este proyecto. Se trata de la obtención de la pieza final a través de la combinación de dos o más piezas. Esta técnica está asociada a una herramienta llamada ***Booleano***, y es una herramienta que permite combinar los objetos A y B de las diferentes formas:

A unión B: se genera un objeto nuevo resultado de la suma de ambos objetos.

A sustracción B: se genera un objeto nuevo resultado de la resta del objeto A menos el objeto B

A intersección B: se genera un objeto nuevo resultado de la intersección de ambos objetos.

A sin B: método similar a la sustracción, aunque en este caso, el objeto generado se trata del objeto A con un hueco en aquellas partes que contacten con el objeto B, pero no se generan los límites internos del hueco provocado.

A modo de ejemplo, se va a explicar la generación de la pieza mostrada en la figura 10 a partir del objeto generado en el anterior ejemplo.

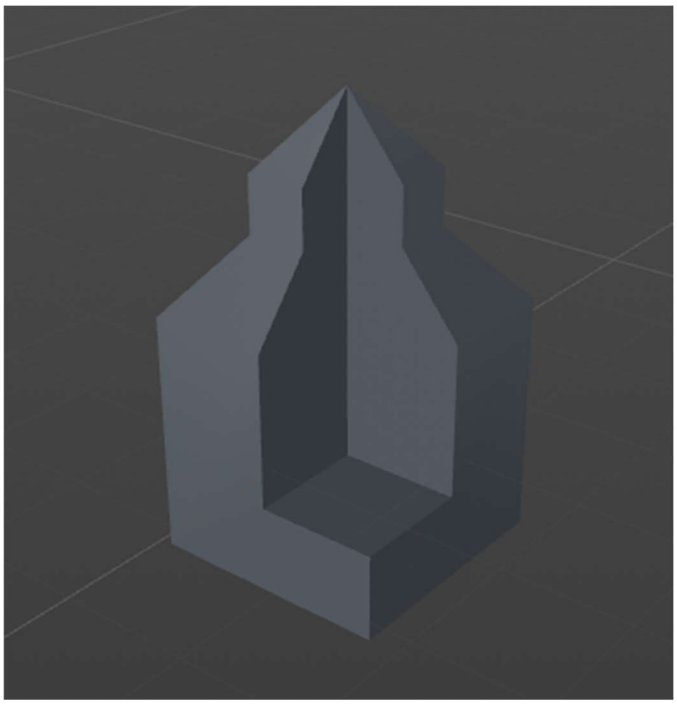


Figura 10. Ejemplo de uso de la herramienta Binario.

El primer paso es crear un cubo que haga las veces de pieza B, es decir, aquella que, será combinada con la pieza original.

Una vez se tienen las dos piezas a combinar, se posicionan de manera que la parte común a ambas piezas resulte ser igual al hueco que se pretende generar, según puede observarse en la figura 11.

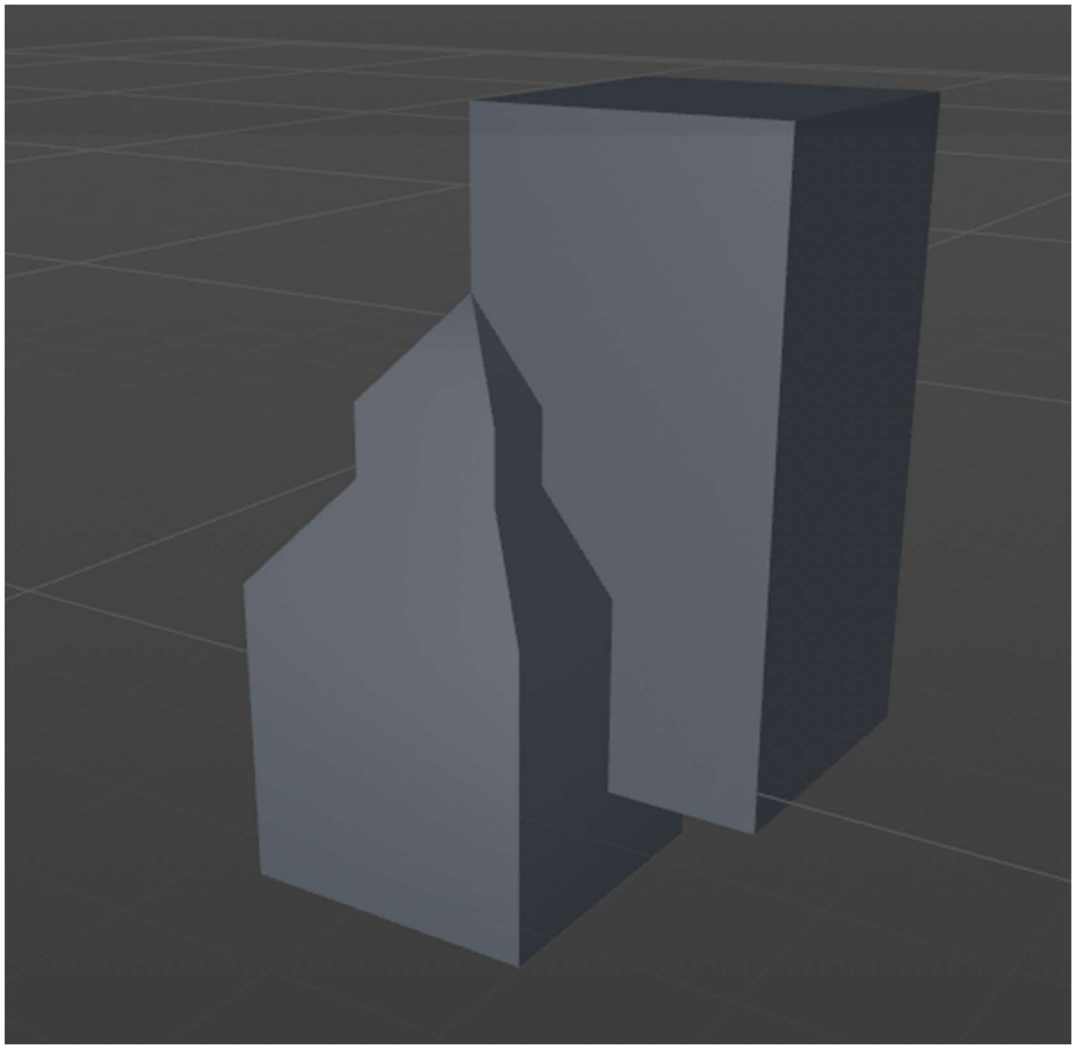


Figura 11. Posicionamiento de los objetos A y B antes de realizar el proceso de combinación.

Con todo esto listo, solo falta hacer uso de la herramienta **Binario.** Para ello, seleccionamos la herramienta en la parte superior, y se creará un nuevo objeto en el inspector llamado Binario, cuyo nombre puede ser modificado, por ejemplo, a **Pieza combinada**. Al seleccionar este objeto, aparecen todos los atributos referentes al mismo. Uno de ellos es el tipo de combinación a realizar, que en este caso es **A sustracción B.** Por último, para hacer efectiva dicha combinación, basta con añadir ambos objetos a combinar como hijos del objeto **Pieza combinada** (figura 12). Es muy importante el orden ded ambos objetos añadidos, ya que, como indica su nombre, la combinación será el primer objeto menos el segundo. Una vez añadidos ambos objetos, puede comprobarse que la combinación se realiza de forma inmediata.

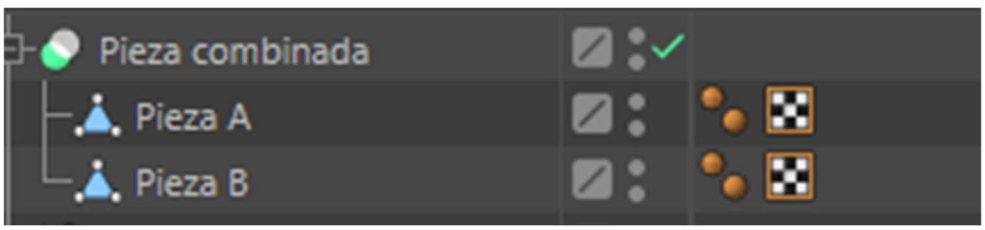


Figura 12. Las piezas A y B se ubican como hijos de la herramienta de combinación para ejecutarse.

Esta técnica se ha usado en este proyecto combinándose con otras herramientas, como puede ser la simetría o el clonado, para generar las piezas a combinar de forma rápida y precisa.

3.3. Modelo 3D de la planta real

A continuación se va a comentar cada uno de los modelos realizados en Cinema 4D.

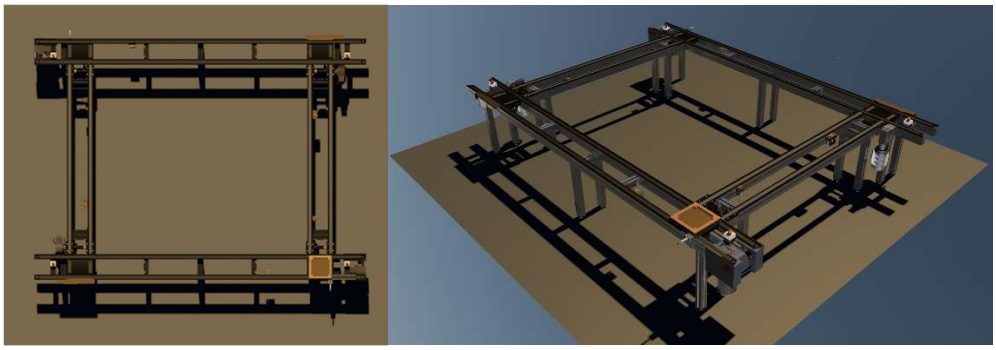


Figura 13. a) Vista superior del modelo completo de la planta y b) Vista en perspectiva de la planta.

La planta en cuestión que se está clonando en este proyecto describe una trayectoria en forma de cuadrilátero definida por cuatro cintas transportadoras, dos de ellas más largas y las otras dos más cortas, que se encargan de transportar plataformas en las cuales se colocaría el producto en producción. En cada uno de los vértices de la trayectoria se encuentran lo que se ha denominado pistones de cambio, ya que su función es realizar el cambio de dirección de las plataformas. Además, para poder controlar el estado de la planta, se han distribuido varios sensores por la planta sobre todo concentrados en las zonas de cambio de dirección. Por último, para controlar el acceso a los pistones de cambio, a la entrada se han colocado unos pistones encargados de detener el paso a futuras plataformas hasta que se quede libre el acceso a una nueva.

3.1.1 Cintas transportadoras

En primer lugar, en la figura 14 pueden verse los dos modelos de cintas transportadoras que contiene la planta. Ambas se componen de una estructura con 4 patas y dos barras horizontales donde se encuentran las cadenas que se encargan de generar el movimiento.

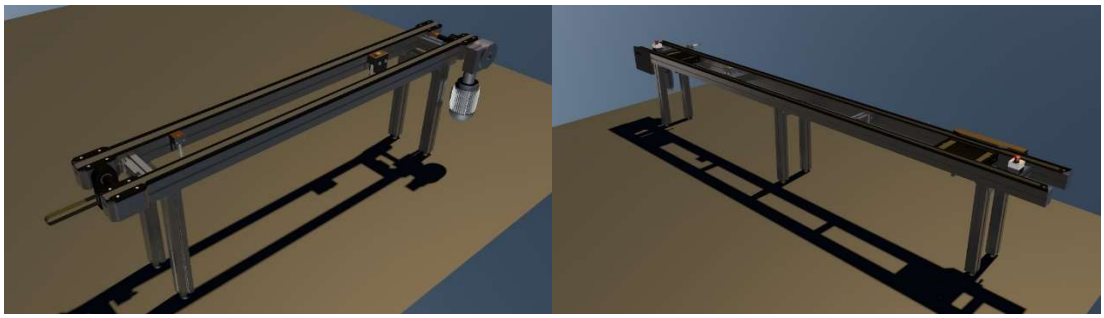


Figura 14. a) Vista general de la cinta corta. b) Vista general de la vista larga.

Por otro lado, se encuentran varias barras perpendiculares que dan firmeza a la estructura, como muestra la figura 15. Estas barras tienen en cada una de sus caras un riel cuya función es la de poder acoplar elementos de forma sencilla en cualquier lugar de la estructura.

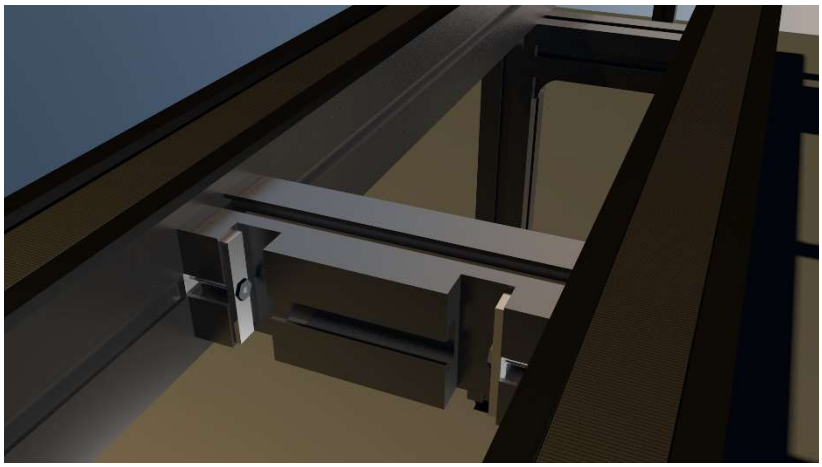


Figura 15. Barra intermedia que conecta y da firmeza a la estructura de la cinta.

Para conseguir la apariencia y el movimiento de la cadena, se ha realizado el modelado de un eslabón de la misma, el cual se ha multiplicado mediante la herramienta clonador, siguiendo la trayectoria que describe el perfil de la cadena, hasta conseguir toda la estructura. Luego, se ha creado una animación para obtener la apariencia de movimiento, ya que el movimiento real se programará en un script, como se explicará más adelante. La figura 16 muestra dos vistas en detalle de la cinta, y se puede apreciar el gran realismo que ofrece este software de modelado 3D, tanto al nivel de detalles como a nivel de texturas.

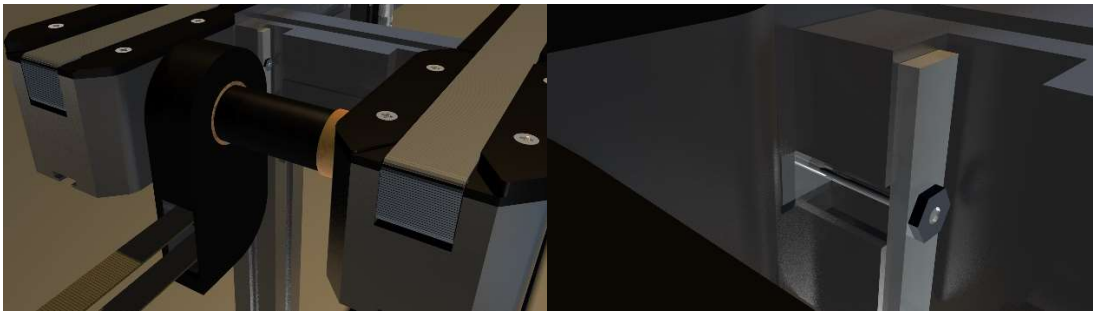


Figura 16. a) Detalle de cinta corta y b) Primer plano de tornillo de cinta corta

Además, se han añadido detalles que no aportan nada al comportamiento de las cintas, sino que añaden realismo al modelo. Por ejemplo, es el caso del motor o la seta de emergencia como se puede observar en la figura 17.

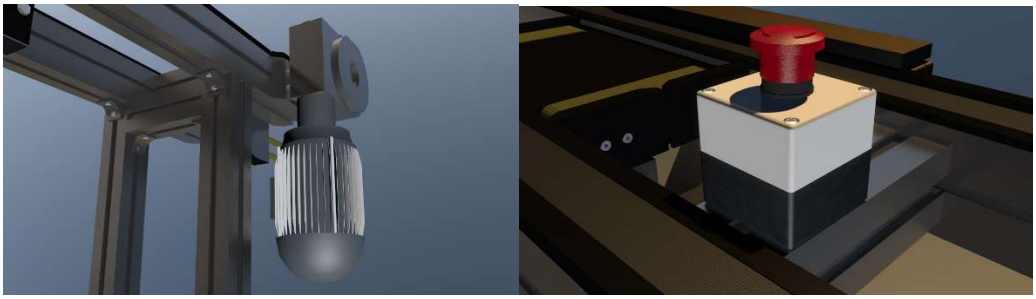


Figura 17. Detalle de motor y seta de emergencia, elementos meramente visuales.

Figura 18. Detalle de marca de la cinta transportadora larga.

3.1.2 Pistón de cambio

Existen cuatro pistones de cambio en la planta, colocados en cada una de las esquinas del recorrido que describen las cintas. Este dispositivo, como se ha comentado anteriormente, realiza la función de cambiar la dirección del movimiento de una cinta a otra.

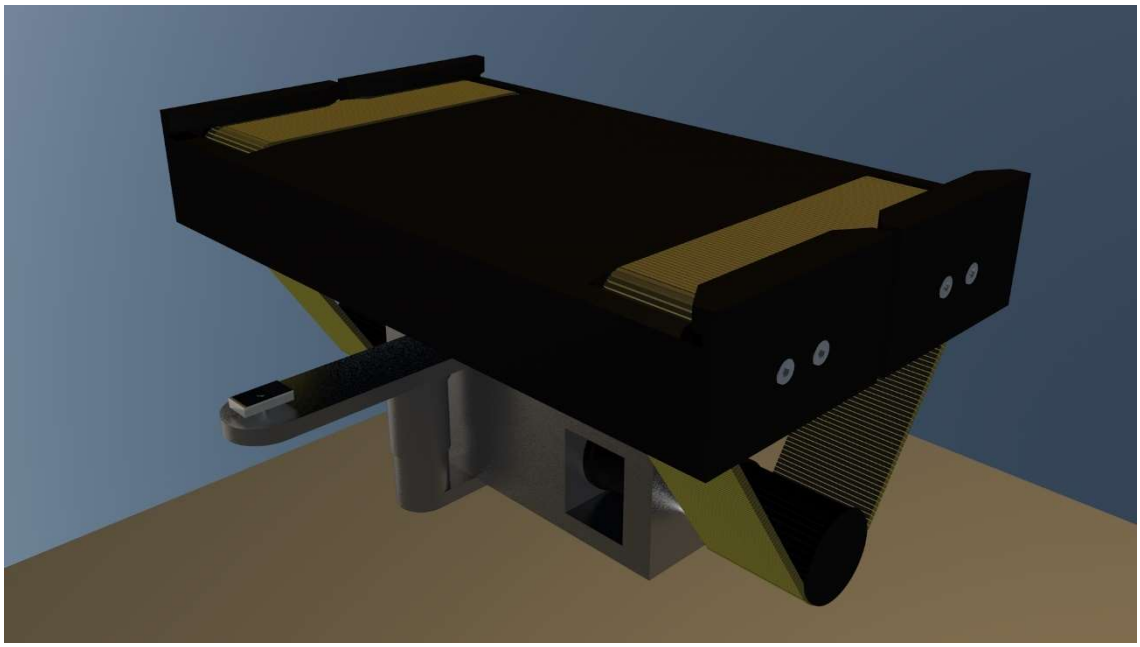


Figura 19. Vista general del pistón de cambio.

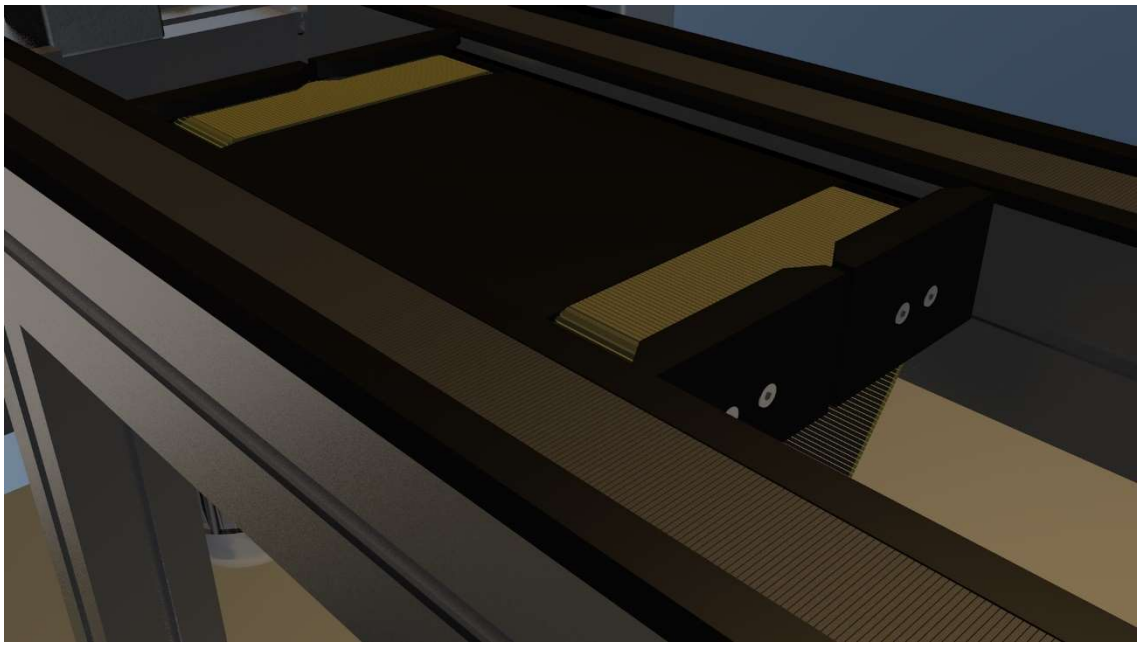


Figura 20. Pistón de cambio encajado en cinta transportadora larga.

Se compone de su propia cinta transportadora, y aparte cuenta con una cinta auxiliar, colocada en la parte inferior del dispositivo, que se conecta con la cinta transportadora corta contigua. Esta cinta se conoce con el nombre de ***seguidor*** (figura 21), ya que su función es transmitir el movimiento de la cinta contigua al pistón, debido a que éste no cuenta con un motor propio. Por esta razón, no es posible gestionar el arranque / paro de esta cinta, sino que se comporta de la misma manera que la cinta. Por otro lado, este elemento cuenta con otro movimiento de subida y bajada, para ponerse a nivel de una cinta u otra, ya que las cintas larga y corta no tienen la misma altura. Este movimiento de subida y bajada lo realiza el propio pistón a base de aire comprimido, al igual que el resto de pistones que forman la planta. El suministro de aire comprimido se respalda por un depósito dedicado, del cual se hablará más adelante. Un detalle importante respecto al modelado 3D es que, mientras que el pistón realiza la subida o bajada, el eje de la cinta se mantiene inmóvil, y esto hace que se produzca un estiramiento en la cinta. Este estiramiento se generó mediante una animación que posteriormente se explicará el modo de gestionarla en Unity. Dicha animación se generó modificando la escala de la cinta en el eje vertical, cuadrándolo con la velocidad de subida y bajada del pistón.

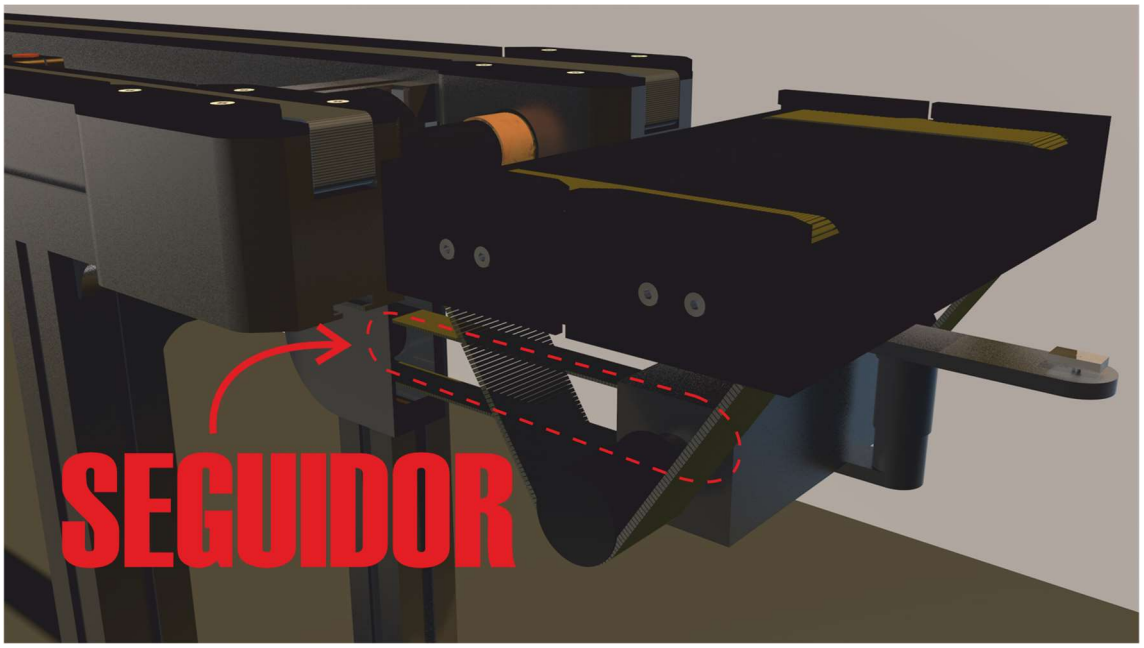


Figura 21. Seguidor que conecta la cinta transportadora corta con el pistón de cambio.

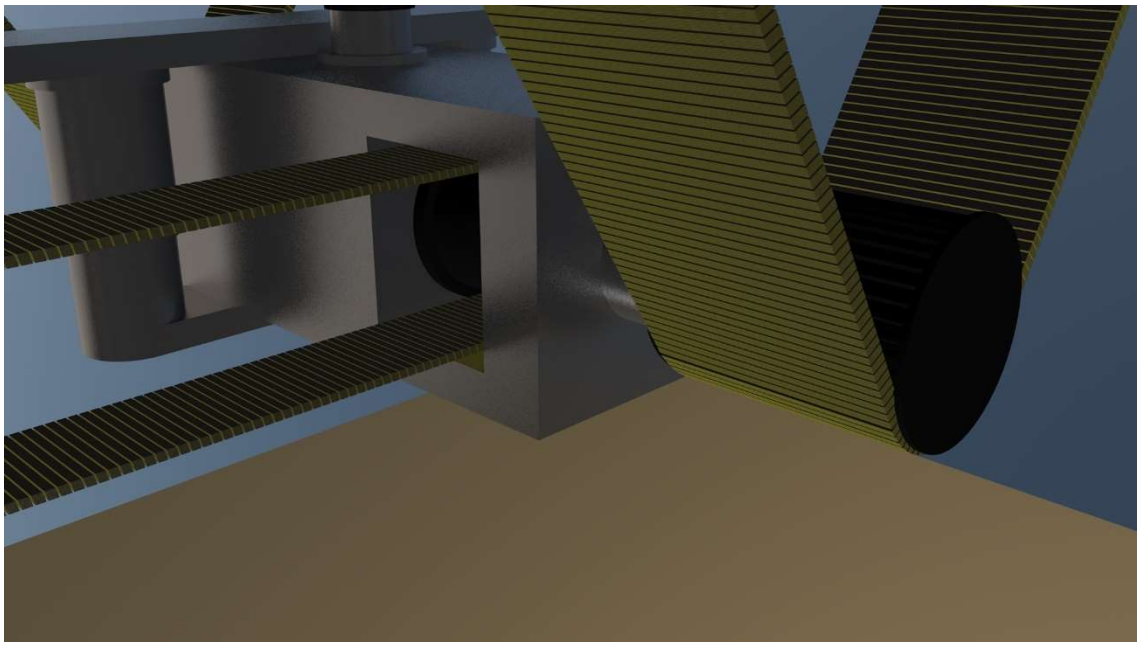


Figura 22. Detalle del seguidor.

3.1.3 Pistón de paro

Este elemento se compone de dos partes, el sensor y el actuador. Se trata de un tope que se eleva, obstruyendo el paso de futuras plataformas. El sensor inductivo da informa ción al sistema sobre el paso de las plataformas, para activar el tope en el momento adecuado. En la figura 23 se muestra el dispositivo acoplado a la estructura, y nuevamente puede apreciarse el gran nivel de detalle al modelar este dispositivo, teniendo en cuenta elementos como los tornillos o la marca, elementos totalmente innecesarios para el funcionamiento, pero que aportan realismo a la escena.

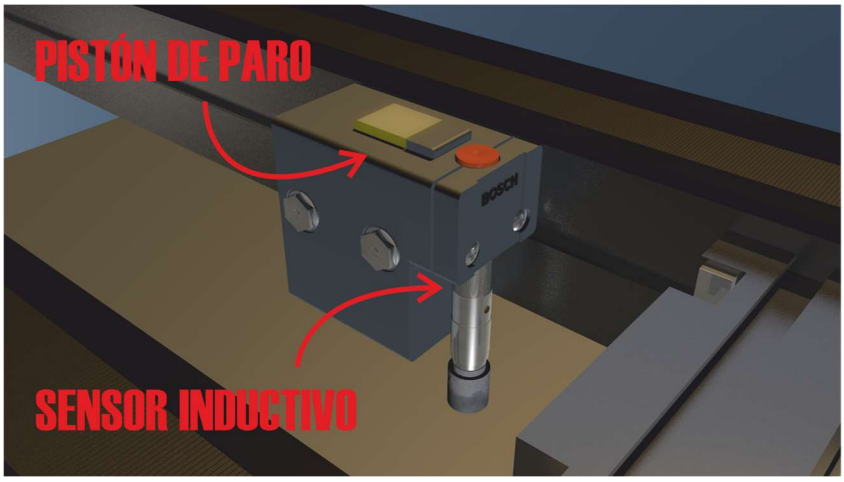


Figura 23. Pistón de paro con sensor inductivo.

3.1.4 Sensor de fuerza

Este elemento se basa en un sensor inductivo al que se le ha acoplado una estructura compuesta por una pieza grande que rota alrededor de un eje, activando el sensor cuando se hace presión sobre ella. Esta estructura convierte el sensor inductivo en lo que podríamos llamar un sensor de presión o de fin de carrera. Más adelante se comentará de forma más extensa su funcionamiento y objetivo.

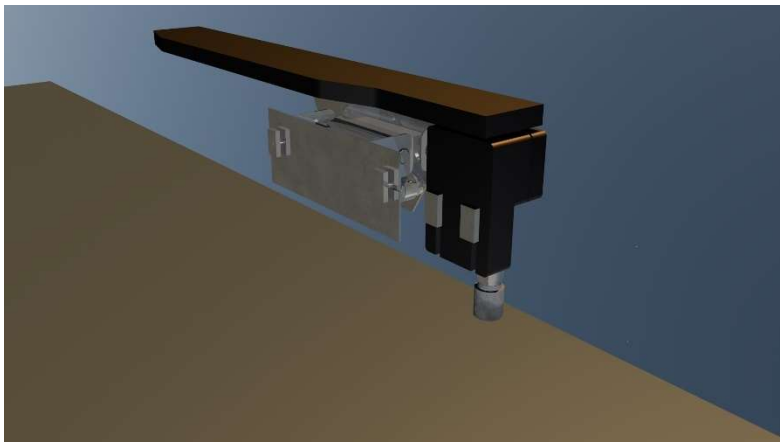


Figura 24. Vista general del sensor de fuerza

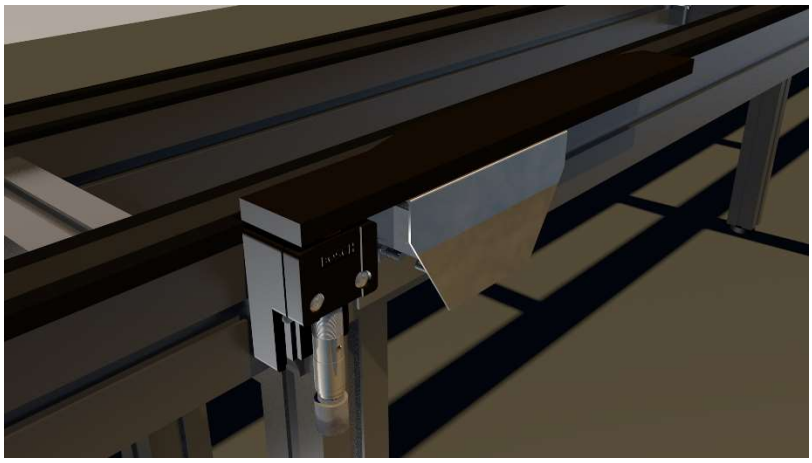


Figura 25. Sensor de fuerza acoplado en la cinta.

En la figura 26 pueden verse varias vistas de perfil del dispositivo, apreciando la parte del eje. En estas vistas destaca también la apreciación de los reflejos que generan las texturas empleadas en objetos como los tornillos, el soporte o la pieza exterior que rota alrededor del eje, texturas que tratan de emular materiales metálicos.

Figura 26. Detalles del sensor de fuerza acoplado en la cinta.

3.4. Exportación de los modelos de Cinema 4D a Unity 3D

Una vez realizados todos los modelos de los elementos industriales de la planta, es el momento de exportarlos a Unity, software donde se crearán los modelos analíticos, se programará el comportamiento de cada uno.

La forma de exportar cada modelo es guardarlo con formato fbx. Este se trata de un formato que empaqueta todo el modelo y funciona bastante bien entre estos dos softwares, conservando a la perfección la jerarquía del objeto con todas sus partes, aceptando también el uso de las animaciones creadas en Cinema 4D.

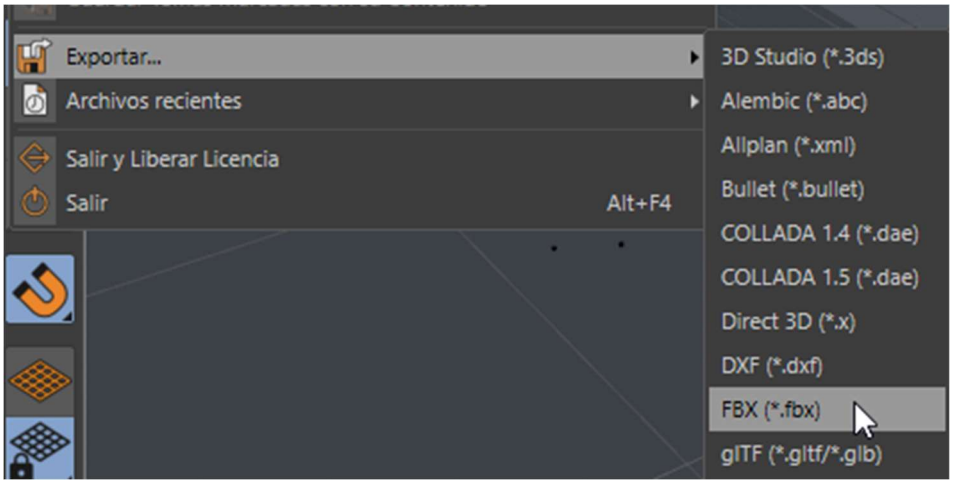


Figura 27. Exportar el modelo 3D en formato FBX.

Una vez creado el archivo fbx, se guarda en alguna carpeta que esté dentro del directorio del proyecto de Unity, y de esa forma, ya será accesible desde el mismo explorador de Unity.

El problema que se ha hallado durante el desarrollo de este proyecto está relacionado con las texturas. Como se ha comentado, Cinema 4D es un software de modelado 3D que ofrece una calidad muy superior, y sobre todo destacan las texturas, cuya apariencia y comportamiento con la luz le dan a los modelos un gran realismo. Sin embargo, al exportar dichos modelos a Unity, se pierde la mayoría de la información relacionada con las texturas, conservando únicamente detalles como el color.

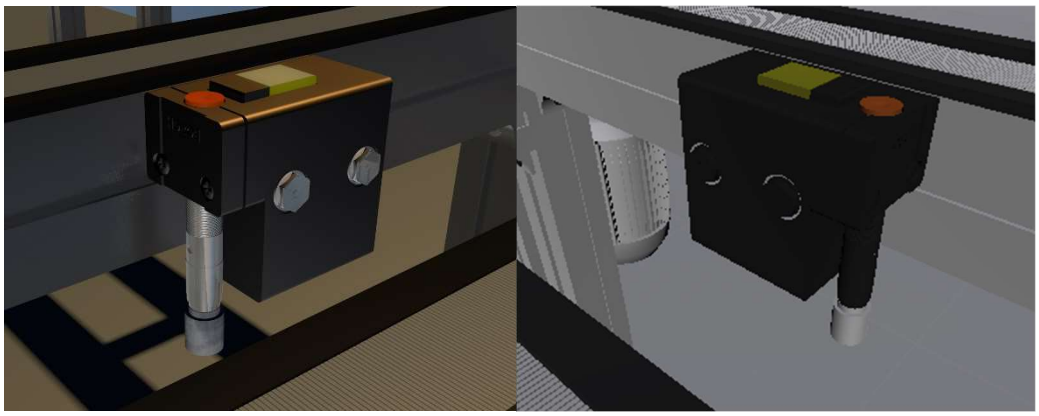


Figura 28. Diferencia entre modelo 3D del pistón de paro en Cinema 4D y en Unity 3D tras ser exportado.

Para solucionar este problema, se estuvo haciendo una búsqueda sobre alternativas para lograr en Unity ese realismo que ofrece Cinema con sus texturas, y se encontró un tipo de texturas denominadas ***PBR (Physical Based Rendering).*** Este tipo de texturas hacen referencia a una técnica de renderizado la cual trata de calcular cómo se comporta la luz al interactuar con la textura en si, tratando la reflexión de la luz, las sombras que origina, etc. Este tipo de texturas están formadas por varias capas, cada una de ellas con una información concreta de algún parámetro, como por ejemplo, el nivel de detalle, el color del material, el desplazamiento de los polígonos, la cantidad de reflexión, el dedtalle de la superficie, y otro tipo de información como transparencia, refracción, etc. [4]

Sin embargo, finalmente no se añadieron dichas texturas al proyecto final de Unity 3D por no ser un aspecto de gran importancia, ya que con el nivel de detalle en el modelado era suficiente, y éste no pierde ningún detalle en la exportación.

4 MODELOS ANALÍTICOS. MOTOR FÍSICO DE UNITY 3D

Una vez explicado el modelado visual de cada uno de los elementos que van a componer el gemelo digital, es el momento de pasar a la programación del comportamiento de cada uno de dichos elementos desarrollados, de tal forma que reflejen un comportamiento lo más fiel a la realidad, y esto implica que dicho comportamiento no sea ideal, es decir, que esté sujeto a los mismos fallos que se generen o puedan generarse en el elemento real. Consegir esa fidelidad en cuanto al comportamiento no es algo trivial ni que se pueda calcular de forma analítica. De hecho, si fuera así, se predeciría cualquier tipo de anomalía antes de que ocurriera. Es por ello que, para lograr que el gemelo se comporte de la manera más parecida posible a la realidad, se define, por un lado, el modelo analítico formado por ecuaciones matemáticas, y por otro lado se van recogiendo datos del elemento real que, haciendo uso de inteligencia artificial, hacen que el simulador vaya aprendiendo del elemento real, y así corregir discrepancias y llegar a predecir fallos.

En este apartado se va a explicar primeramente el desarrollo de los modelos analíticos que definen el comportamiento de los elementos del simulador, que luego serán nutridos con la información proveniente del aprendizaje del gemelo.

El nivel de fidelidad entre el modelo analítico y la realidad es un detalle a definir por el programador, y se elegirá dependiendo de varios factores, como el tipo de sistema que se esté modelando, la tarea que realiza el mismo, etc. Por ejemplo, para modelar el comportamiento de un robot que desempeñe una tarea muy precisa con un margen de error mínimo, será necesario un desarrollo más exhaustivo que otro robot que realice una tarea más básica, en el cual se podrán despreciar ciertos aspectos dinámicos que no sean relevantes para el modelo.

Una vez dicho esto, Unity3D aporta una gran ayuda a este cometido, ya que dispone de lo que se conoce como **motor físico**, el cual se encarga del cálculo de efectos cinemáticos y dinámicos básicos. Esto facilita mucho el desarrollo, ya que efectos como los asociados a la gravedad o el rozamiento de un objeto son gestionados por dicho motor.

A continuación, se va a explicar el desarrollo de cada uno de los modelos analíticos que definen el comportamiento básico de los elementos modelados anteriormente.

Sin embargo, es importante tener claro el concepto que a partir de ahora se va a denominar como elemento físico.

El **elemento físico** es la parte del objeto donde los efectos físicos e interacciones con otros objetos son relevantes para el análisis del comportamiento del objeto. Es por ello que, en Unity, será la única parte del objeto donde se habilitará este tipo de efectos, siendo el resto del objeto un aporte meramente visual, ya que su interacción física no aporta información de interés y no haría más que aumentar la complejidad del modelo.

Esto implica que la delimitación del elemento físico dependerá en cierta medida del nivel de detalle que se necesite alcanzar en el modelo. Poniendo un ejemplo, en la figura puede apreciarse un modelo 3D de una cinta trasportadora. A la hora de definir los elementos físicos de dicho objeto, es muy distinto si se desea modelar únicamente la dinámica de una caja deslizándose por la parte superior de la cinta, que modelar, por ejemplo, la fuerza de sujección de los tornillos de la plataforma, o las deformaciones que la parte superior de la cinta pueda generar sobre las patas de la misma. Es por ello que, dependiendo del estudio que se quiera realizar, habrá efectos que sean más importantes, y otros que sean despreciables.

Una vez explicado el concepto, cada uno de los elementos físicos se configuran conforme a las necesidades del modelo, activando aquellas dependencias físicas que sean necesarias. De forma general, los efectos a tener en cuenta en un elemento físico son los siguientes:

- La acción de la gravedad
- Fuerzas de rozamiento
- Colisiones con otros objetos

Estos efectos se configuran en cada elemento físico dependiendo de la importancia que ofrezca al modelo, y están íntimamente relacionados con el motor físico de Unity.

Para configurar los parámetros dinámicos de un objeto en Unity, es necesario añadir el componente ***Rigidbody,*** seleccionando la opción ***Add component*** en el inspector (figura 29).

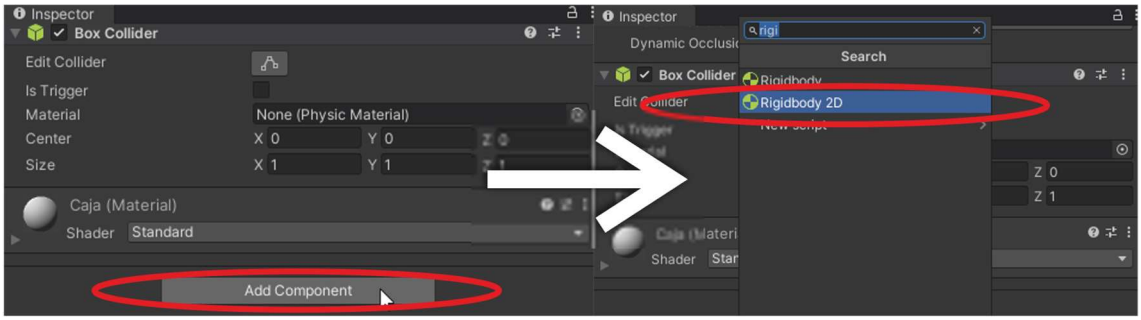


Figura 29. Adición del componente Rigidbody a un objeto.

Este componente permite activar la acción de la gravedad sobre el objeto (opción ***Use Gravity***), modificar la masa del mismo, y otros parámetros asociados a la física del objeto. Para establecer el rozamiento del objeto, sin embargo, es necesario crear un ***Physic material*** (figura 30), configurarlo y añadirlo al objeto.

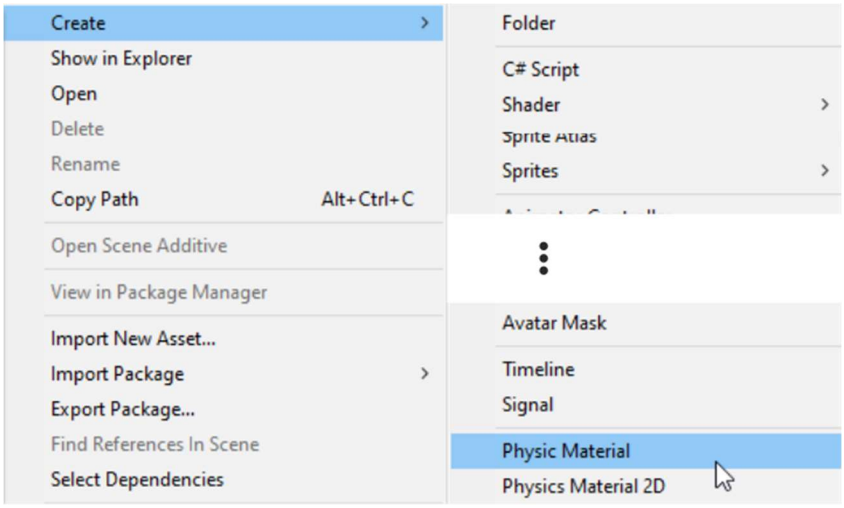


Figura 30. Creación de un material físico.

Este elemento permite definir la rugosidad del objeto al que se haya asignado, y será muy importante en el desarrollo de los objetos que se describen a continuación.

Por último, es necesario conocer el concepto de ***Collider***. El collider es el volumen que delimita un objeto con propósitos de colisiones físicas. Este componente, que es invisible, no tiene por qué ser de la misma forma que el objeto al que pertenece, y de hecho, lo más común es que se trate de una forma más sencilla y simplificada del objeto para ganar en eficiencia. Existen varios tipos de colliders en Unity, agrupados en dos grupos:

- **Primitive colliders.** Estos son los colliders más sencillos y que consumen menos carga de procesador. En 3D, son 3, Sphere Collider, Box Collider y Capsule Collider. Se puede generar un collider compuesto por varios colliders de este tipo, y dimensionando y posicionando bien cada uno de ellos, puede conseguirse que delimiten el objeto del cual dependen, ganando bastante en eficiencia. De hecho, lo normal a la hora de generar un collider de un objeto es que con uno o varios de este tipo sea suficiente, ya que no suele hacer falta delimitar el objeto concreto de forma muy precisa a la hora de contemplar las colisiones del mismo. Poniendo como ejemplo un juego de tipo tercera persona, en el que hay que dirigir un personaje, suele ser muy común usar en este caso un *Capsule Collider,* ajustarlo a las dimensiones del personaje.
- **Mesh colliders.** Existen casos en los cuales, aún creando un collider compuesto por multitud de

primitive colliders, la precisión conseguida no es suficiente. En ese caso, una alternativa es hacer uso de los *mesh colliders.* Este tipo de colliders toman la forma exacta del objeto al que pertenecen. Si el objeto tiene un gran nivel de detalle, usar un mesh collider hará que la carga computacional sea bastante mayor, por eso es recomendable usarlo solo en caso de que sea necesario. Por otro lado, tiene limitaciones a la hora de contemplar colisiones. Un objeto con Mesh collider no va a generar ningún tipo de collisiones con otro objeto. Para solucionar este problema, existen dos soluciones. La primera es que, si el objeto que tiene este tipo de collider está fijo o tiene un movimiento programado, es decir, que no depende de la gravedad y su movimiento está totalmente definido, puede definirse como Is kinematic. Esto permitirá que otro elemento colisione con este. La segunda opción es activar la casilla de ***Convex.*** Esto hace que se genere un collider simplificado del objeto, rellenando huecos y perdiendo algo de precisión, pero ganando en eficiencia, y permitiendo que éste colisione con otros objetos, pudiendo depender de la fuerza de la gravedad.

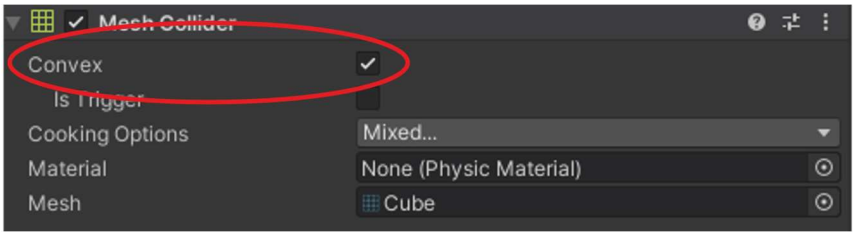


Figura 31. Configuración de un Mesh Collider como Convex.

A continuación, se va a explicar cada uno de los modelos analíticos desarrollados para los elementos modelados, que serán asociados a elementos físicos concretos.

4.1. Desarrollo de los modelos analíticos.

En este apartado se va a desarrollar la explicación del modelo analítico de cada uno de los elementos que forman la planta, que se agrupan en tres grandes entidades: sensores, actuadores y otros objetos. A continuación, se enumeran todos los elementos clasificados en estos tres grupos.

- Sensores:
  - Sensor inductivo

- Actuadores:
  - Cinta transportadora larga
  - Cinta transportadora corta
  - Pistón de cambio
  - Pistón de paro

- Otros objetos
  - Compresor de aire

Por tanto, ahora se va a explicar uno a uno cada uno de los elementos que mencionados, comenzando por los actuadores.

4.1.1 Actuadores

Los actuadores son los elementos de la planta que ejecutan algún proceso de movimiento o empuje para actuar sobre otros elementos y así lograr el buen funcionamiento global del sistema. El funcionamiento básico

de los actuadores es ejecutar su proceso en base a una variable que será gestionada por los sensores que dependen de ese proceso.

Para lograr un direccionamiento y gestión sencilla de todos los actuadores creados en la planta, se ha creado una clase común a todos ellos, que contiene los parámetros que definen a un actuador. Esta clase se llama ***ActuatorComponent.***

4.1.1.1 ActuatorComponent

Se trata de una clase dedicada a facilitar el direccionamiento y unificación de todos los actuadores.

En este caso, la clase dispone de 4 campos importantes, que son ***Name***, ***Number***, ***Type*** y ***flag.***

Name almacena el nombre del objeto del actuador.

Number contiene el número del actuador.

Type clasifica el actuador dentro de una de estas 4 categorías: LongConveyor, ShortConveyor, Piston y AirCompress. Este campo facilita mucho la programación a la hora de trabajar en grupos de actuadores, como se verá más adelante.

Flag almacena el valor de la o las salidas de dicho actuador. Deben ser salidas booleanas, y un actuador puede tener una o varias.

A continuación se muestra el código asociado a dicha clase.

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


//////////////////////////////
// VARIABLES FOR THE ACTUATOR //
//////////////////////////////
public class ActuatorComponent : MonoBehaviour
{
   public string Name;
   public int Number;
   public string Type;
   public bool[] flag;
}
```

4.1.1.2 Cinta Transportadora Larga

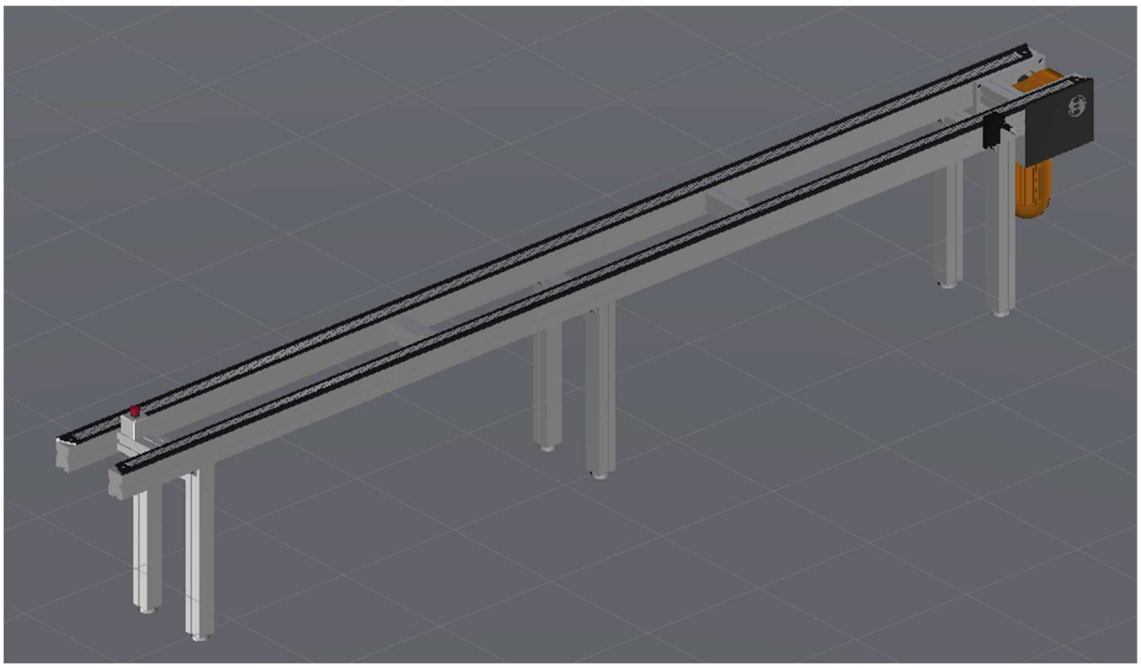


Figura 32. Modelo 3D de la cinta transportadora larga importado en Unity.

A la hora de modelar el comportamiento de la cinta transportadora larga, al igual que la mayoría de elementos que se van a describir en este trabajo, se han despreciado consideraciones como la tensión del motor necesaria para mover la cinta, el consumo energético, etc. Sin embargo, esas apreciaciones serán abordadas en un futuro, centrando este proyecto en un primer acercamiento a un modelo de comportamiento.

En primer lugar, se va a explicar el proceso de importación del modelo 3D a Unity, en el que se usará un procedimiento idéntico al resto de elementos.

Haciendo un breve recordatorio, una vez modelado el elemento en Cinema4D junto con su animación, y exportado en formato fbx, se va a guardar el archivo dentro del directorio del proyecto de Unity3D. Concretamente, la ruta de guardado es ***Assets > Objects > Conveyors > LongConveyor.***

Una vez guerdado el modelo en el lugar correcto, antes de colocarlo en la escena, es necesario configurar correctamente la animación del mismo. Para ello, se busca el modelo 3D en el explorador del proyecto, y se selecciona la pestaña ***Animation.*** En esta pestaña se pueden configurar multitud de parámetros referentes a la animación o animaciones que contenga el modelo importado. En este caso, sólo son necesarios varios ajustes. En primer lugar, al tratarse de un movimiento cíclico, activar la casilla ***Loop Time.*** Esto hará que la animación se reproduzca de forma continua. Por otro lado, también existe la opción de recortar el tiempo de la animación. En este caso, interesa que la duración de la animación sea mínima, para que al parar la cinta, la animación cese al mismo tiempo. Como se trata de una animación cíclica que muestra el movimiento de los eslabones de la cinta, basta con coger 3 o 4 fotogramas donde el último coincida con el primero. Haciendo un rápido ajuste visual, se ha elegido una sección de la animación que va desde el fotograma 1 al 9. De todas formas, estos detalles, al tener un propósito meramente visual, no son de crucial importancia.

Para terminar, se le pone nombre a la animación, y ya quedaría totalmente configurada.

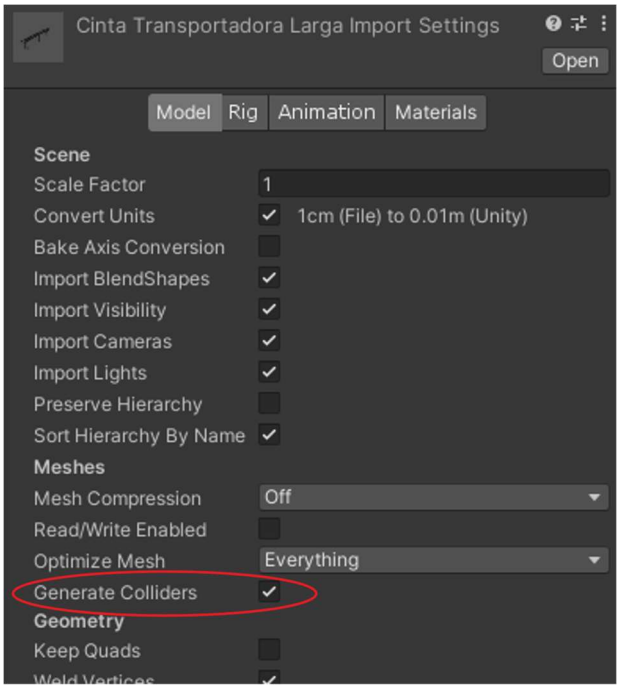


Figura 33. Activar la opción Generate Colliders al importar el modelo

Como último paso antes de introducir el elemento en la escena, en la pestaña Models se muestran varias opciones relacionadas con la importación de la geometría del modelo. Muchas opciones vienen seleccionadas por defecto, y únicamente será necesario añadir la opción ***Generate colliders***. Esto hará que, al añadir el objeto a la escena, cada pieza de dicho elemento traiga su propio ***Mesh Collider***.

Una vez hecho esto, se crean dos cintas transportadoras largas, llamadas ***LongConveyor0*** y ***LongConveyor1.*** A partir de este punto, toda la explicación va referida a ambas cintas, aunque para simplificar la explicación, se hablará siempre de la cinta 0.

4.1.1.2.1 Animación

Para poder gestionar la animación de la cinta cuando ésta se active, es necesario crear un nuevo ***Animator Controller,*** que en este caso se llama Cinta larga, y servirá para ambas cintas, la 0 y la 1. Este elemento permite crear un diagrama de estados en los cuales cada estado puede gestionar la activación o desactivación de una animación (figura 34). En este caso, se trata de un diagrama muy sencillo con dos estados, uno de ellos para la cinta parada (Estado 1), y el otro para la cinta en movimiento (Estado 2). En el estado 2, en el inspector se añade la animación de la cinta, definida anteriormente. Por último, es necesario añadir las transiciones entre un estado y otro, y la condición de cada una se gestiona con una nueva variable propia del animator controller, ***flag.***

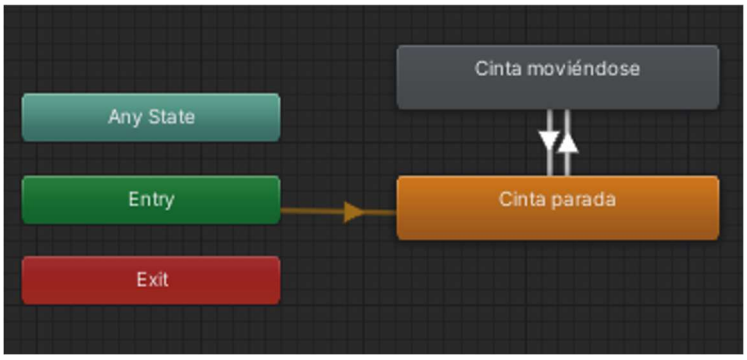


Figura 34. Animator controller de la cinta larga.

Esta variable se crea en la pestaña ***Parameters*** del animator controller, y las condiciones serán que la variable se active (Estado 1 -> Estado 2), o se desactive (Estado 2 -> Estado 1).

Una vez diseñado el animator controller, es necesario conectarlo con la cinta. Para ello, en el inspector de la cinta creada (figura 35), añadir un nuevo componente llamado ***Animator.*** Es importante que, aunque el elemento al que afecta la animación sea la cadena de la cinta, es preciso añadir el componente al objeto mas superior de la cinta, ya que, al importar el modelo de Cinema, las animaciones, en cierta manera, están conectadas al objeto más superior de la jerarquía del modelo importado.

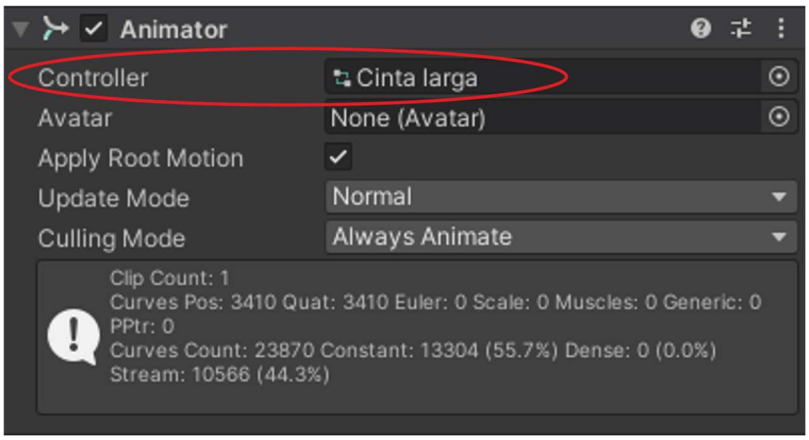


Figura 35. Adición animator controller de la cinta larga

Por tanto, el componente Animator se añade al objeto ***LongBeltConveyor0***, y dentro de este componente, seleccionando el pulsador del campo Controller, se añade el animator controller creado anteriormente, ***Cinta larga.***

4.1.1.2.2 Programación

Es el momento de explicar la programación del modelo analítico de la cinta transportadora larga. Un aspecto importante que se ha procurado llevar a cabo en este proyecto es utilizar una programación lo más desacoplada y estandarizada posible, de tal forma que la adición de un nuevo elemento sea lo más sencillo posible, sin tener que reprogramar lo que ya está incluído en el proyecto.

Todos los scripts de los modelos analíticos de los objetos de este proyecto se han situado en la siguiente ruta: ***Assets > Scripts > Models.*** Dentro de esta carpeta, dependiendo del tipo de elemento, se clasigica en Actuator o Sensor. Además se ha incluído la carpeta Objects, para algún elemento extra. En este caso, el programa de la cinta transportadora larga, ***LongBeltConveyor.cs***, se ha ubicado en la carpeta ***Actuators.***

Al principio de esta clase se encuentra la definición de variables:

- **cintaAnim:** Variable de tipo *Animator*, de la cual se obtendrá el parámetro flag para gestionar la animación.
- **cinta:** Variable de tipo *Rigidbody* que alamacenará todos los parámetros físicos de la cinta.
- **actuatorComponent:** variable de tipo *ActuatorComponent,* clase explicada anteriormente que aporta los parámetros básicos de un actuador.

A continuación se encuentra el método ***Awake()***. Toda la programación contenida en este método se ejecuta una vez, al iniciarse el script. Es por ello que este método está destinado a la inicialización de variables.

Como puede observarse, en este caso, se inicializan las variables previamente creadas. La variable ***actuatorComponent*** se incluye como clase en el gameObject de la cinta larga. Una vez incluída, se rellenan los campos de la misma (nombre y tipo del objeto, valor por defecto del actuador). Por último se inicializan las variables ***cinta y cintaAnim*** con los valores del Rigidbody y el Animator, respectivamente.

Por último se encuentra el método ***FixedUpdate().*** Este método, junto con ***Update(),*** se ejecuta de manera cíclica. La diferencia entre ambos es que ***FixedUpdate()*** se ejecuta una cantidad de veces fija cada segundo,

mientras que ***Update()*** lo hace una cantidad variable, dependiente del rendimiento de cada ordenador. El método ***FixedUpdate()*** se usa más para acciones físicas, animaciones y movimientos, donde es necesario mantener un tiempo controlado. Es por ello que, en este caso, se hace uso de este método.

Dentro de este método se incluye la programación del movimiento de la cinta. En este caso, para conseguir este comportamiento, se han combinado dos acciones que suceden de forma continua:

1. Desplazamiento físico de la cinta una distancia de terminada por Time.fixedDdeltaTime * LongBeltConveyorModel.Parameters.Speed y en la dirección del eje x del objeto (transform.right) durante un ciclo del programa.

La variable Time.fixedDeltaTime contiene el tiempo de actualización del programa en cuanto a cuestiones físicas. La velocidad deseada se asigna en la variable Speed declarada en un script auxiliar llamado ModelSupport, que incluye toda la información adicional a los modelos. En este caso, incluye la clase LongConveyorModel, la cual contiene el valor de la velocidad (speed) y el valor de la fricción de la cinta (BeltFriction). Este movimiento físico creado hace que se transmita a la plataforma que se coloque encima, provocando el desplazamiento de la misma. Para este movimiento, se hace uso del método ***cinta.MovePosition()***.

2. Asignación instantánea de la posición inicial a la cinta. Este movimiento no se rige por cuestiones físicas, por lo que no se transmite a la caja que está encima. Esta asignación es necesaria para evitar que la cinta se mueva a lo largo del tiempo. Sin embargo, combinando el movimiento con la asignación de la posición inicial, a una frecuencia considerable, se consigue como resultado ese desplazamiento de cualquier objeto que se sitúe encima, sin que se aprecie movimiento alguno en el objeto de la cinta. Esta asignación de la posición inicial se realiza declarando directamente el valor deseado en la posición mediante ***cinta.position***.

Esta rutina de movimiento se ejecuta cuando se cumple la condición a comprobar, que es la activación de la variable ***actuatorComponent.flag[0].*** Las variables que gestionan la activación de la o las salidas de cada actuador están almacenadas en el vector flag. En este caso, la cinta larga sólo posee una salida, que es el movimiento de la misma, y dicha variable se trata de la posición 0 del vector flag.

Por último se encuentra la gestión de la animación de la cinta, que debe activarse cada vez que lo indique la variable flag. La gestión de la animación se realiza mediante una variable creada en el animator controller, variable que gestiona el paso del estado "animación cinta parada" al estado "animación cinta activa", teniendo en cuenta que en el primer caso se trata de un estado vacío, y el segundo inicia la animación propia de la cinta. Por tanto, al final del código, se copia el valor de la variable flag a la variable del animator controller.

```
cintaAnim.SetBool("flag", actuatorComponent.flag[0]);
```

A continuación, se muestra el código completo del modelo analítico de la cinta transportadora larga.

```
////////////////////////////////
// ACTION OF LONG BELT CONVEYOR //
////////////////////////////////
public class LongBeltConveyor : MonoBehaviour
{
    Animator cintaAnim;
    Rigidbody cinta;
    ActuatorComponent actuatorComponent;

    void Awake()
```

```
        {
            actuatorComponent = gameObject.AddComponent<ActuatorComponent>() as ActuatorComponent;
            actuatorComponent.flag = new bool[1];
            actuatorComponent.flag[0] = true;
            actuatorComponent.Name = transform.name;
            actuatorComponent.Type = "LongConveyor";

            cinta = GetComponentInChildren<Rigidbody>();
            cintaAnim = GetComponentInParent<Animator>();
        }


        void FixedUpdate()
        {
            if (actuatorComponent.flag[0])
            {
                cinta.position += transform.right * LongBeltConveyorModel.Parameters.Speed * Time.fixedDeltaT
ime;
                cinta.MovePosition(cinta.position - transform.right * LongBeltConveyorModel.Parameters.Speed *
 Time.fixedDeltaTime);
            }
            cintaAnim.SetBool("flag", actuatorComponent.flag[0]);
        }
    }
    ////////////////////////////////
```

4.1.1.3 Cinta transportadora corta

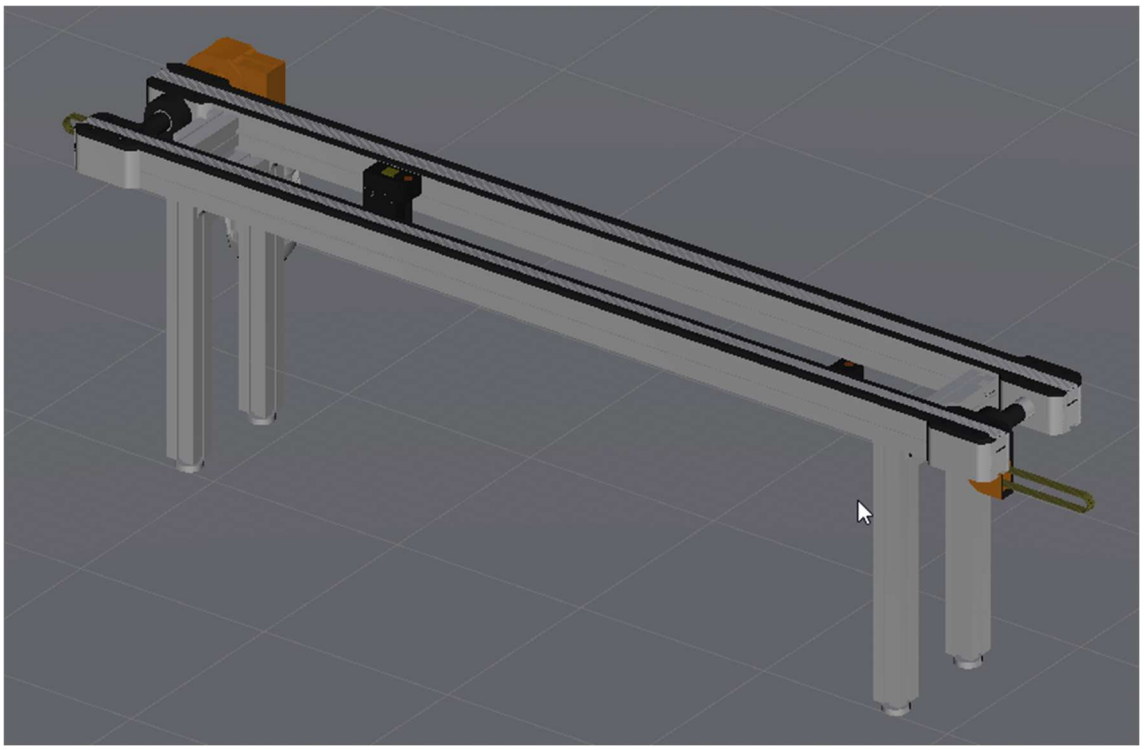


Figura 36. Modelo 3D de la cinta transportadora corta importado en Unity.

La explicación de este elemento de la planta va a ser muy breve, ya que su desarrollo es igual al de la cinta larga, cambiando únicamente el modelado 3D del mismo. Sin embargo, la existencia de un script dedicado a la cinta corta se debe a poder así definir parámetros diferentes en la clase ***actuatorComponent***, cambiando en este caso el nombre del actuador a *ShortConveyor*. Al existir únicamente esta diferencia entre ambos modelos analíticos, existiría la posibilidad de unificar ambos modelos en uno solo. Sin embargo, se han querido mantener ambos, para así tener un script para cada actuador, facilitando así posibles ampliaciones en uno de los dos actuadores sin tener que modificar el otro.

A continuación, se adjunta el código de la cinta transportadora corta:

```
using UnityEngine;
using System.Collections;

////////////////////////////////
// ACTION OF SHORT BELT CONVEYOR //
////////////////////////////////
public class ShortBeltConveyor : MonoBehaviour
{
    Animator cintaAnim;
    Rigidbody cinta;
    ActuatorComponent actuatorComponent;

    void Awake()
```

```
        {
            actuatorComponent = gameObject.AddComponent<ActuatorComponent>() as ActuatorComponent;
            actuatorComponent.flag = new bool[1];
            actuatorComponent.flag[0] = true;
            actuatorComponent.Name = transform.name;
            actuatorComponent.Type = "ShortConveyor";

            cinta = GetComponentInChildren<Rigidbody>();
            cintaAnim = GetComponentInParent<Animator>();
        }


        void FixedUpdate()
        {
            if (actuatorComponent.flag[0])
            {
                cinta.position += transform.right * ShortBeltConveyorModel.Parameters.Speed * Time.fixedDeltaT
ime;
                cinta.MovePosition(cinta.position - transform.right * ShortBeltConveyorModel.Parameters.Speed
* Time.fixedDeltaTime);
            }
            cintaAnim.SetBool("flag", actuatorComponent.flag[0]);
        }
    }
```

4.1.1.4 Pistón de cambio

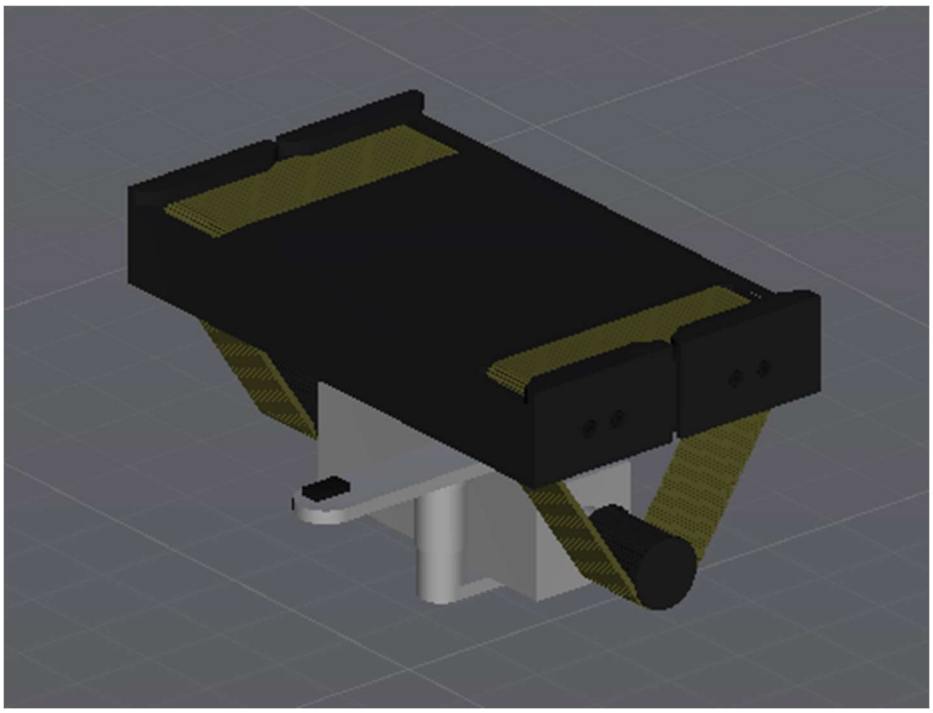


Figura 37. Pistón de cambio.

Este elemento es el encargado de hacer posible el cambio de dirección de las plataformas de una cinta transportadora a otra. Su modelado ha resultado algo complejo en algunos aspectos como el estiramiento y contracción realista de las correas propias de la cinta transportadora.

Una vez diseñado el modelo en Cinema4D, se guarda en la carpeta ***Assets > Scripts > Models > Conveyors,*** preparado para su uso en Unity 3D.

4.1.1.4.1 Animación

Dentro del proyecto, antes de importarlo, es necesario definir las animaciones. Este objeto contiene animaciones relacionadas con dos movimientos: el desplazamiento de la cinta y el alargamiento y contracción de la misma al elevarse o bajarse.

Sin embargo, la existencia de estos dos movimientos y la posibilidad de que se den a la vez hace que la recreación sea algo compleja, y la solución a la que se ha llegado es crear tantas animaciones como combinaciones puedan darse entre ambos movimientos, como por ejemplo, cinta en movimiento y pistón arriba, o cinta parada y pistón bajando, etc.

Todas estas animaciones se han creado en Cinema 4D de forma secuencia como una única. Por tanto, en el momento de importar en Unity 3D, es necesario separarlas y nombrar cada una. Esto se realiza seleccionando el modelo del pistón en la ruta mencionada, y abriendo la pestaña de animación. En la imagen 38 puede verse la separación de cada animación con su respectivo nombre.

Figura 38. Clasificación de todas las animaciones del pistón de cambio.

Una vez clasificadas correctamente, es necesario crear el Animator controller, encargado de gestionar la activación de la animación correcta en el momento adecuado.

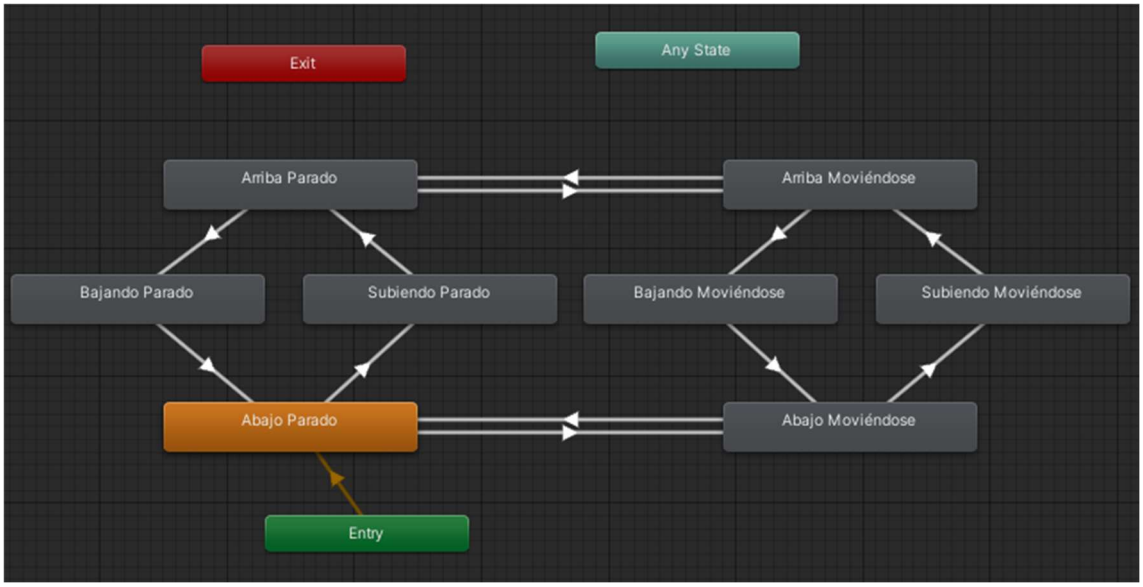


Figura 39. Diagrama de estados del Animator controller del pistón de cambio.

Como puede apreciarse en la figura 39, en este caso, el diagrama de estados es bastante más complejo ya que el hecho de que haya un total de 8 animaciones distintas hace que se complique un poco. La transición entre estados es gestionada por dos variables, que son ***flag,*** variable relacionada con el movimiento de la cinta, y ***flagPiston,*** variable que gestiona la subida/bajada del pistón. Una vez configurado por completo el animator controller, por último, será necesario añadirlo como componente a cada uno de los pistones.

Para ello, se crean 4 pistones de cambio, colocados en cada una de las esquinas formadas por las cintas transportadoras, uniendo así cada pistón una cinta larga con una corta.

Una vez creados los pistones, seleccionando cada uno, en el inspector se añade el componente ***Animator***, y dentro de éste, en el campo *Animator controller,* se selecciona CP_Cinta, el controlador de las animaciones de dicho pistón.

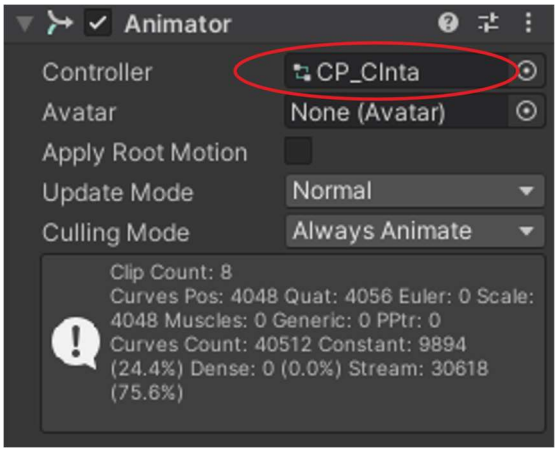


Figura 40. Adición del animator controller al pistón de cambio.

4.1.1.4.2 Programación

El modelo analítico de este elemento se basa en dos rutinas de movimiento, que serían el movimiento horizontal de la cinta transportadora, y el movimiento vertical del propio pistón de cambio.

El script que se ha programado agrupa estas dos tareas. La rutina de la cinta transpostadora es exactamente la misma que se ha explicado previamente, por lo que su explicación será muy superficial, haciendo más incapiñe en la rutina de subida y bajada.

En primer lugar se declaran todas las variables necesarias para este programa:

- ***sentidoCinta:*** variable que define el sentido de movimiento que describirá la cinta transportadora.
- ***speed:*** velocidad de movimiento de la cinta transportadora.
- ***pistonAnim:*** variable tipo Animator que gestiona las animaciones del pistón de cambio.
- ***dinamicoP:*** variable que almacena los parámetros del rigidbody del elemento ***DynamicPositionChangePistonModule***.
- ***cinta:*** variable que almacena los parámetros del rigidbody del elemento ***Physical*** (este elemento modela la cinta transportadora del pistón de cambio).
- ***actuatorComponent:*** variable que almacena los parámetros de este actuador referentes a la clase ***actuatorComponent.***

Una vez definidas todas las variables, se procede a su inicialización dentro del método ***Awake().*** En primer lugar se inicializan los campos de la clase actuatorComponent. Una observación es que en este caso, este elemento cuenta con dos variables de actuación, que son las que gestionan la cinta y la subida/bajada del pistón.

Después se inicializan las variables de los ***rigidbody***. Por último, se inicializa ***pistonAnim***, variable que gestionará las animaciones del pistón.

Una vez finalizada la inicialización, se entra en el método ***FixedUpdate()***. Este método contiene un bucle anidado con las dos variables de activación de este elemento. Si la primera varible del vector flag se activa (***flag[0]***), se procede a ejecutar el código de movimiento de la cinta transportadora, igual que el ya explicado en el elemento previo. La rutina de subida y bajada del pistón se ha incluido dentro de esta condición como bucle anidado. Esta segunda rutina es gestionada por la segunda variable del vector flag ***(flag[1])***, aunque no es condición única. También se comprueba que el pistón no se encuentre ya en la posición final del movimiento a efectuar. Es decir, la activación de la variable ***flag[1]*** indica la subida del pistón. Pero para que se efectúe la subida, además se comprueba que el pistón no esté ya en el límite de subida, y viceversa en el caso de bajada. Los límites de subida y bajada del pistón se han definido en el vector ***Limits*** contenido en la estructura ***Parameters*** contenida en la clase ***ChangePistonModuleModel***, que a su vez se trata de una clase contenida en el script ***ModelsSupport***. Esta definición de parámetros aislada se ha realizado para ganar en comodidad y orden, facilitando la modificación de los mismos en el caso de que sea necesario, sin tener que buscar éstos en el código del elemento. Para encontrar los valores de los límites de subida y bajada, se ubicó el pistón de forma manual al

mismo nivel de altura que cada una de las cintas, guardando el valor del eje y que resultaba en ese momento. Es importante tener en cuenta que este valor se trata de la altura en ejes locales del elemento.

Para efectuar la subida/bajada del pistón, se usa también el método ***MovePosition***, y la velocidad de este movimiento (***speedUpDown***) se lee de la estructura ***Parameters***, contenida en el script de parámetros ***ModelsSupport.***

Por último, puede observarse la gestión de las variables del animator controller dentro de cada condición, activando la animación de la cinta con la variable ***flag***, o la animación de la subida/bajada mediante la variable ***flagPiston***, y cambiando el valor de las variables mediante el método pistonAnim.SetBool("variable", "value of variable"), indicando el nombre y valor de la variable.

A continuación se añade el código asociado al pistón de cambio.

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


//////////////////////////
// ACTION OF CHANGE PISTON //
//////////////////////////
public class ChangePistonModule : MonoBehaviour
{
    public int sentidoCinta = 1;
    float speed;
    Animator pistonAnim;
    Rigidbody dinamicoP, cinta;
    ActuatorComponent actuatorComponent;


    void Awake()
    {
        actuatorComponent = gameObject.AddComponent<ActuatorComponent>() as ActuatorComponent;
        actuatorComponent.flag = new bool[2];
        actuatorComponent.flag[0] = true;
        actuatorComponent.Name = transform.name;
        actuatorComponent.Type = "Piston";


        dinamicoP = transform.GetChild(0).GetComponent<Rigidbody>();
        cinta = transform.GetChild(0).GetChild(1).GetComponent<Rigidbody>();
        pistonAnim = GetComponentInParent<Animator>();
    }


    void FixedUpdate()
    {
```

```
        speed = ChangePistonModuleModel.Parameters.Speed * sentidoCinta;
        pistonAnim.SetFloat("speed", speed);

        if (actuatorComponent.flag[0])
        {
            cinta.position -= transform.right * speed * Time.fixedDeltaTime;
            cinta.MovePosition(cinta.position + transform.right * speed * Time.fixedDeltaTime);

            pistonAnim.SetBool("flag", true);

            if (actuatorComponent.flag[1] && dinamicoP.transform.localPosition.y < ChangePistonModuleMo
del.Parameters.Limits[1])
            {
                dinamicoP.MovePosition(dinamicoP.position + transform.up * ChangePistonModuleModel.Para
meters.SpeedUpDown * Time.fixedDeltaTime);
                cinta.MovePosition(cinta.position + (transform.up * ChangePistonModuleModel.Parameters.Spe
edUpDown + transform.right * speed) * Time.fixedDeltaTime);

                pistonAnim.SetBool("flagPiston" , true);
            }

            else if (!actuatorComponent.flag[1] && dinamicoP.transform.localPosition.y > ChangePistonModu
leModel.Parameters.Limits[0])
            {
                dinamicoP.MovePosition(dinamicoP.position - transform.up * ChangePistonModuleModel.Para
meters.SpeedUpDown * Time.fixedDeltaTime);
                cinta.MovePosition(cinta.position - (transform.up * ChangePistonModuleModel.Parameters.Spe
edUpDown - transform.right * speed) * Time.fixedDeltaTime);

                pistonAnim.SetBool("flagPiston" , false);
            }
        }

        else
        {
            pistonAnim.SetBool("flag", false);

            if (actuatorComponent.flag[1] && dinamicoP.transform.localPosition.y < ChangePistonModuleMo
del.Parameters.Limits[1])
            {
```

```
                dinamicoP.MovePosition(dinamicoP.position + transform.up * ChangePistonModuleModel.Para
meters.SpeedUpDown * Time.fixedDeltaTime);
                cinta.MovePosition(cinta.position + transform.up * ChangePistonModuleModel.Parameters.Spe
edUpDown * Time.fixedDeltaTime);
                pistonAnim.SetBool("flagPiston" , true);
            }

            else if (!actuatorComponent.flag[1] && dinamicoP.transform.localPosition.y > ChangePistonModu
leModel.Parameters.Limits[0])
            {
                dinamicoP.MovePosition(dinamicoP.position - transform.up * ChangePistonModuleModel.Para
meters.SpeedUpDown * Time.fixedDeltaTime);
                cinta.MovePosition(cinta.position - transform.up * ChangePistonModuleModel.Parameters.Spee
dUpDown * Time.fixedDeltaTime);
                pistonAnim.SetBool("flagPiston" , false);
            }
        }
    }
}
```

4.1.1.5 Pistón de paro

El pistón de paro es un elemento industrial sencillo cuya misión es la de detener el movimiento de los objetos que circulan por una cinta transportadora, normalmente para evitar acumulación, o para generar cierta distancia entre cada uno, y así evitar fallos en el funcionamiento de la planta.

El modelado de este elemento, debido a su simpleza, se ha realizado dentro de la cinta larga, por lo que ya estaría añadido al proyecto, a falta de su programación.

El comportamiento básico de este elemento es la subida y bajada del pistón. Este movimiento es bastante sencillo de programar, por lo que no requiere de ninguna animación.

Por otro lado, este modelo también contempla un sensor acoplado. Sin embargo, de esta parte se hablará en otro apartado.

4.1.1.5.1 Programación

La rutina que ejecuta este elemento es bastante sencilla de programar. En primer lugar se definen las variables necesarias, que en este caso son únicamente tres.

A continuación se encuentra el método ***Awake()***, dedicado a la inicializazción de dichas variables. Como puede observarse en el código abajo mostrado, en primer lugar se definen los campos del ***actuatorComponent***, como son el nombre y tipo del actuador, y el valor asociado a su salida. Luego se inicializa la variable dedicada al Rigidbody, y por último el vector ***limits[]***, el cual contendrá los valores asociados a las dos posiciones finales que debe tomar el pistón al realizar el desplazamiento. El valor de ambos límites se ha encontrando seleccionando el valor de la componente ***y*** al colocar manualmente el pistón en las dos posiciones deseadas, la posición de reposo, donde se mantiene a la altura de la cinta, permitiendo el paso de la plataforma, y la posición activa, donde se eleva lo suficiente para evitar que la plataforma continúe moviéndose, manteniéndose bloqueada por éste al llegar a su posición.

Una vez definidos todos estos campos, se ejecuta la rutina de movimiento, dentro del método ***FixedUpdate().*** En este caso, se comprueba la condición de subida o bajada del pistón mediante una estructura tipo ***if()-else if()***.

Las condiciones son dos: condición de subida del pistón, y condición de bajada del mismo. Ambas son similares. La condición de subida se basa en que el primer valor del vector ***flag[]*** esté activado, y el pistón no esté en el límite superior del movimiento, es decir, que no esté ya subido. Para la bajada, la condición es igual, pero se mira el segundo valor del vector, y que el pistón no esté en el límite inferior del movimiento. Para ejecutar el movimieto de subida o bajada, se acude al método MovePosition, definiendo la dirección y velocidad del movimiento. A continuación se encuentra el código explicado:

```
using System;
using UnityEngine;

//////////////////////////
// ACTION OF STOP ACTUATOR //
//////////////////////////
public class StopActuatorModule : MonoBehaviour
{
   Rigidbody StopActuatorObject;
   ActuatorComponent actuatorComponent;
   float[] limits = new float[2];

   void Awake()
   {
      actuatorComponent = gameObject.AddComponent<ActuatorComponent>() as ActuatorComponent;
      actuatorComponent.flag = new bool[1];
      actuatorComponent.Name = transform.name;
      actuatorComponent.Type = "Piston";

      StopActuatorObject = GetComponent<Rigidbody>();

      limits[0] = StopActuatorObject.transform.localPosition.z;
      limits[1] = limits[0] - 0.2f;
   }


   void FixedUpdate()
   {
      if (actuatorComponent.flag[0] && StopActuatorObject.transform.localPosition.z > limits[1])
         StopActuatorObject.MovePosition(StopActuatorObject.position + transform.up * StopActuatorMo
duleModel.Parameters.SpeedUpDown * Time.fixedDeltaTime);
      else if (!actuatorComponent.flag[0] && StopActuatorObject.transform.localPosition.z < limits[0])
         StopActuatorObject.MovePosition(StopActuatorObject.position - transform.up * StopActuatorMod
```

```
uleModel.Parameters.SpeedUpDown * Time.fixedDeltaTime);
        }
    }
    ////////////////////////////
```

4.1.1.6 Compresor y depósito de aire comprimido

Como se ha ido explicando, existen varios pistones ubicados por la planta: cuatro pistones de cambio y cuatro pistones de paro. Estos elementos funcionan a base de aire comprimido, el cual genera esa fuerza necesaria para mantenerlos en posición de subida. Por tanto, el estado en el que alguno de los pistones se encuentra subido supone un gasto concreto de aire, dependiendo del tipo de pistón que sea.

Este aire es suministrado por un depósito dedicado, el cual cuenta sensores para controlar la capacidad del mismo.

Concretamente, se han definido **tres umbrales de capacidad**. El primero de ellos, el ***MaxCapacity,*** define el estado en el que el depósito se considera lleno. Por otro lado se encuentran el ***MinCapacity*** y ***MinCapacityToEnable***. De todas formas, de estos parámetros se hablará con más detalle más adelante.

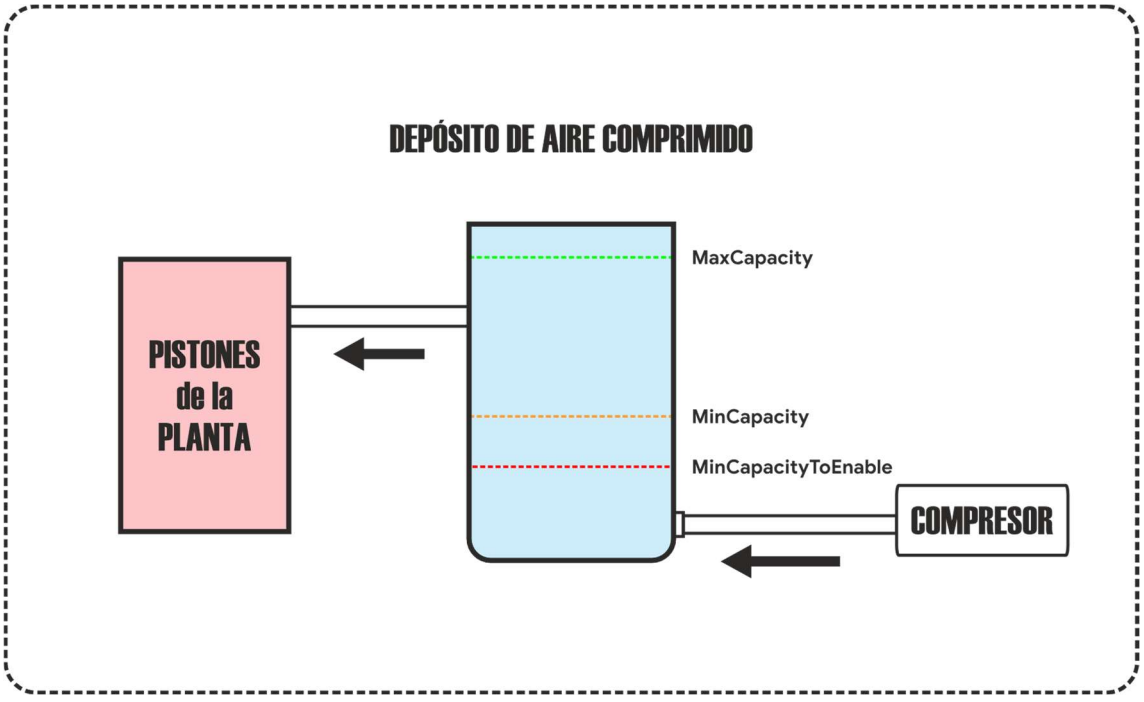


Figura 41. Esquema general del sistema depósito-compresor.

Si el control detecta que la capacidad del depósito resulta inferior a ***MinCapacity*** el sistema activa el funcionamiento del compresor, el cual comienza el proceso de llenado del depósito, llenado que se ha modelado de forma lineal, es decir, a una **velocidad constante**. Este proceso de llenado continuará en funcionamiento hasta que se alcance ***MaxCapacity***. Por otro lado, si la capacidad llega a ser inferior al ***MinCapacityToEnable***, debido a un momento de acción en paralelo de multitud de pistones, de forma adicional se activa un mecanismo de seguridad que **detiene el funcionamiento de todos los pistones** hasta recuperar el nivel de llenado del depósito, es decir, hasta ***MaxCapacity.***

Sin embargo, este proceso de automatización del funcionamiento del compresor no se ha abordado en este proyecto, sino que se ha abordado en otro proyecto aparte…

Una vez introducido este nuevo elemento, se va a comentar los scripts que definen su funcionamiento. En este caso existen dos scripts principales, que son ***AirTank*** y ***AirCompress.***

El script ***AirCompress***, como su nombre indica, contiene el modelo de funcionamiento dedl compresor.

En primer lugar, se define la variable ***InstantPower***, variable que indica la potencia que aporta el compresor. Además, se crea la variable de tipo ***ActuatorComponent***, variable en la cual se almacenan todos los atributos del compresor.

A continuación se inicializan dichas variables, dentro del método ***Awake()***.

```
void Awake()
  {
    actuatorComponent = gameObject.AddComponent<ActuatorComponent>() as ActuatorComponent;
    actuatorComponent.flag = new bool[1];
    actuatorComponent.Name = transform.name;
    actuatorComponent.Type = "AirCompress";
    PowerConsumption.AirCompressor = 0;
  }
```

En este caso, el compresor cuenta con una única salida, para activar / desactivar el aporte de aire al depósito. A parte, se define el nombre del actuador, y se añade el campo ***Type***, donde se aclara que el actuador es de tipo ***AirCompress.*** Este campo se ha añadido para localizar este elemento de forma rápida y directa, como se verá luego en el script del depósito. Además, se inicializa el gasto de la potencia del compresor a 0. Este parámetro se ha definido en una clase distinta llamada PowerConsumption, y aunque se trata del único parámetro de esa clase, se ha separado para reservar una clase para todos los parámetros relacionados con el gasto de potencia que se creen en un futuro.

A continuación, dentro del método ***FixedUpdate()***, se asigna el valor de la variable ***InstantPower***, en función de la salida del actuador, la variable ***flag.*** Por el momento, se ha definido un aporte tipo todo/nada, es decir, si se activa el compresor, éste aportará una cantidad constante de 10, y si se desactiva, inmediatamente el aporte pasa a ser nulo. Por último, dentro de este método también se añade la expresión de la potencia consumida, que va acumulando el valor de ***InstantPower.*** A continuación se adjunta el código de este script.

```
using UnityEngine;
using System.Collections;

//////////////////////////
// AIR COMPRESS PROCESS //
//////////////////////////
public class AirCompress : MonoBehaviour
{
  // DECLARATION OF GLOBAL VARIABLES
  double InstantPower;
  ActuatorComponent actuatorComponent;
```

```
    void Awake()
    {
        actuatorComponent = gameObject.AddComponent<ActuatorComponent>() as ActuatorComponent;
        actuatorComponent.flag = new bool[1];
        actuatorComponent.Name = transform.name;
        actuatorComponent.Type = "AirCompress";
        PowerConsumption.AirCompressor = 0;
    }


    void FixedUpdate()
    {
        if (actuatorComponent.flag[0])
            InstantPower = 10;
        else
            InstantPower = 0;

        PowerConsumption.AirCompressor += InstantPower;
    }
}
```

Por otro lado, se encuentra el script ***AirTank***, el cual modela el comportamiento del depósito de aire comprimido. Este script se divide en varios métodos, para dividir todo el código en pequeñas funciones y hacerlo más legible y sencillo.

Al comienzo del script se crean las variables. En este caso son las siguientes:

- ***Pistons*** se trata de un vector de tipo ***ActuatorComponent*** que almacena todos los pistones de la planta en un vector, para así actuar sobre la salida de cada uno.
- ***Compress*** es una variable de tipo ***ActuatorComponent*** en la cual se almacena el componente del compresor anteriormente explicado, y así poder leer el valor de su salida.
- ***Capacity*** es la variable que almacena el valor de la capacidad del tanque de aire. Esta variable indicará en todo momento el valor real de este parámetro. Se crea con el valor nulo por defecto.
- ***SensorComponent*** es una variable de tipo SensorComponent, la estructura que almacenaba los atributos de cada sensor. En este caso se almacenará el valor las variables que chequean las comparaciones de la capacidad del depósito con los umbrales establecidos, variables que indicarán al control el modo de actuar. Al inicializar este componente, se incluye como componente del objeto AirTank, y se inicializan sus parámetros, como el vector ***flag***, que almacena el valor que tomarán los sensores del depósito, y por el nombre, que en este caso toma el nombre del gameobject, es decir, ***AirTank.***

```
public class AirTank : MonoBehaviour
{
```

```
// DECLARATION OF GLOBAL VARIABLES
ActuatorComponent[] Pistons;
ActuatorComponent Compress;
public double Capacity = 0;
SensorComponent sensorComponent;

void Awake()
{
   sensorComponent = gameObject.AddComponent<SensorComponent>() as SensorComponent;
   sensorComponent.flag = new bool[] {false, false};
   sensorComponent.Name = transform.name;
}
```

Una vez inicializado el elemento SensorComponent en el método ***Awake(),*** a continuación se emplea el método ***Start()*** para una segunda inicialización. Se emplean ambos métodos ya que el método ***Start()*** nunca se ejecuta hasta que se haya ejecutado por completo el método ***Awake().***

```
void Start()
{
   InitializePistons();

   foreach (ActuatorComponent actuator in transform.GetComponentsInChildren<ActuatorComponen
t>())
      if (actuator.Type == "AirCompress")
      {
         Compress = actuator;
         break;
      }
}
```

Dentro del método ***Start()***, primeramente se ejecuta el método ***InitialicePistons().*** Este primer método se encarga de buscar todos los actuadores de la planta, y aquellos que sean de tipo ***Piston*** los almacena en un vector de tipo ***ActuatorComponent()***, almacenando también el número de pistones en la variable ***count.*** Este vector permitirá actuar de forma rápida sobre cualquiera de los pistones de la planta.
Una vez inicializados los pistones, para terminar la inicialización, se busca el componente del actuador del compresor y se almacena también en la variable ***Compress***.

```
void FixedUpdate()
{
```

```
        // Turn off the actuators if the capacity of the tank is lower than the minimun capacity to enable
        if (Capacity < AirTankModel.Parameters.MinCapacityToEnable)
            DisablePistons();

        DynamicAirTank();

        SensorsTank();
    }
```

Por consiguiente, se ejecuta el método ***FixedUpdate().*** Este método contiene la rutina principal del script. Dentro de él se ejecutan tres métodos en bucle. Sin embargo, antes de explicarlos, es necesario hablar de la clase ***AirTankModel.*** Esta clase está definida en el script ***ModelsSupport,*** script donde se definen parámetros relativos a cada uno de los modedlos. En este caso, la clase ***AirTankModel*** contiene una estructura denominada ***Parameters*** que, como su nombre indica, contiene parámetros propios del tanque de aire. Dichos parámetros son los siguientes:

- CompressorGain: ganancia del compresor.
- StopModuleCost: gasto de aire que produce el pistón de paro.
- ChangePistonCost: gasto de aire que produce el pistón de cambio.
- RemanentCost: gasto constante de aire debido a imperfecciones.
- MaxCapacity: umbral de capacidad máxima del depósito.
- MinCapacity: umbral de capacidad baja del depósito.
- MinCapacityToEnable: umbral de capacidad mínima indispensable para el uso de los pistones.

Una vez aclarado este conjunto de parámetros, volviendo al método ***FixedUpdate(),*** se encuentra el método ***DisablePistons().*** Este método básicamente se encarga de poner a cero la salida de todos los pistones. Es por ello que, para que se ejecute este método, se debe cumplor la condición de que la capacidad del depósito sea inferior a ***MinCapacityToEnable***.

El siguente método que se aparece es ***DynamicAirTank().*** Este método se encarga de calcular el valor de la variable ***Capacity***. Para ello, en primer lugar calcula el gasto de aire en el depósito causado por cada uno de los pistones que estén activos, separándolos por tipos, ya que cada uno posee un gasto diferente. Luego, añade el gasto por imperfecciones, que es bastante menor al de los pistones pero hay que tenerlo en cuenta. Por último, comprueba el aporte de aire proveniente del compresor, siéste está activado. Si es el caso, la variable ***CompressorGain*** indica la cantidad de aire transmitida en cada iteración.

Por último se encuentra el método ***SensorsTank()***. Este método se encarga de agrupar en un vector de dos variables booleanas el resultado de la comparación de la capacidad con los parámetros ***MinCapacity*** y ***MaxCapacity.*** Estos son los indicadores a usar por el programa de automatización del sistema de control del depósito.

A continuación, se incluye el código completo del script ***AirTank:***

```
using UnityEngine;
using System.Collections;

//////////////////////
```

```
// TANK AIR PROCESS //
//////////////////////
public class AirTank : MonoBehaviour
{
  // DECLARATION OF GLOBAL VARIABLES
  ActuatorComponent[] Pistons;
  ActuatorComponent Compress;
  public double Capacity = 0;
  SensorComponent sensorComponent;

  void Awake()
  {
    sensorComponent = gameObject.AddComponent<SensorComponent>() as SensorComponent;
    sensorComponent.flag = new bool[] {false, false};
    sensorComponent.Name = transform.name;
  }

  void Start()
  {
    InitializePistons();

    foreach (ActuatorComponent actuator in transform.GetComponentsInChildren<ActuatorComponen
t>())
      if (actuator.Type == "AirCompress")
      {
        Compress = actuator;
        break;
      }
  }


  void FixedUpdate()
  {
    // Turn off the actuators if the capacity of the tank is lower than the minimun capacity to enable
    if (Capacity < AirTankModel.Parameters.MinCapacityToEnable)
      DisablePistons();

    DynamicAirTank();
```

```
        SensorsTank();
    }

    void InitializePistons()
    {
        ActuatorComponent[] Aux = transform.GetComponentInParent<Main>().GetComponentsInChildren<ActuatorComponent>();

        int count = 0;
        foreach (ActuatorComponent actuator in Aux)
            if (actuator.Type == "Piston")
                count++;

        Pistons = new ActuatorComponent[count];
        foreach (ActuatorComponent actuator in Aux)
            if (actuator.Type == "Piston")
            {
                Pistons[Pistons.Length - count] = actuator;
                count--;
            }
    }

    void DisablePistons()
    {
        foreach (ActuatorComponent piston in Pistons)
            if (piston.Name == "DynamicStopActuatorModule")
            {
                if (piston.flag[0] == true)
                    piston.flag[0] = false;
            }
            else if (piston.flag[1] == true)
                piston.flag[1] = false;
    }

    void DynamicAirTank()
    {
        foreach (ActuatorComponent piston in Pistons)
            if (piston.Name == "DynamicStopActuatorModule")
```

```
        {
            if (piston.flag[0] == true)
                Capacity -= AirTankModel.Parameters.StopModuleCost;
        }
        else if (piston.flag[1] == true)
            Capacity -= AirTankModel.Parameters.ChangePistonCost;

    // Constant loss due to imperfections

    Capacity -= AirTankModel.Parameters.RemanentCost;

    //

    if (Compress.flag[0])
        Capacity += AirTankModel.Parameters.CompressorGain;
}

void SensorsTank()
{
    if (Capacity < 0)
        Capacity = 0;
    else if (Capacity > AirTankModel.Parameters.MaxCapacity)
        Capacity = AirTankModel.Parameters.MaxCapacity;

    // Update Sensor Value
    sensorComponent.flag = new bool[]
    {
        Capacity >= AirTankModel.Parameters.MinCapacity,
        Capacity >= AirTankModel.Parameters.MaxCapacity
    };
}
}
```

4.1.2 Sensores

Los sensores son los elementos encargados de captar información referente al estado de la planta, para poder activar ó desactivar los actuadores en el momento adecuado. En este caso solo se ha implementado un tipo de

sensor: un modelo de sensor inductivo. Los sensores inductivos son un tipo de sensor que trabajan mediante un campo magnético producido al generar una corriente eléctrica a través del devanado interno. Su forma de detección de objetos se basa en la variación de la corriente debido a la interacción de los objetos con el campo magnético generado. Es por ello que los objetos detectables son metálicos.

La medida resultante de estos sensores puede ser analógica, mostrando la distancia al elemento detectado, o booleana, activando o desactivando la salida del sensor en función de si la distancia al objeto detectado sobrepasa un umbral establecido. En este caso, se ha modelado un sensor inductivo booleano, ya que la información necesaria para esta planta se basa en indicar la presencia o no de un objeto.

4.1.2.1 SensorComponent

Esta clase se ha creado para clasificar todos los sensores existentes en la planta.

La clase consta de un campo ***Name*** dedicado para el nombre del objeto que constituye el sensor, y el campo ***Number***, que indica el número de sensor. Estos dos campos permiten direccionar de manera muy sencilla los sensores.

Además, puede observarse un tercer campo, ***flag***. Esta es la variable booleana que almacenará el valor dfel sensor. Dicha variable se ha definido de forma vectorial (bool[] flag). Esto se ha hecho para poder hacer un buen uso del método ***set***. Este se trata de un método interno que traen de serie todas las variables, y que se ejecuta siempre que se le asigne un valor. Si no se indica nada en ese método, lo único que ocurre es que la variable cambia su valor al nuevo indicado. Sin embargo, en este caso se ha aprovechado este método para ejecutar todo aquello que sea necesario en el momento en el que cambia de valor algún sensor.

Una vez realizado el modelo 3D del sensor, para modelar el efecto de captar objetos metálicos cerca, se ha acudido al uso del ***trigger***. El ***trigger*** es un tipo de configuración que se puede aplicar a un collider para que gestione una variable de salida en función de la interacción de otros objetos con el espacio que define dicho collider. Al configurarse como ***trigger***, se desactivan las colisiones en este componente.

Por tanto, el proceso es sencillo. Una vez importado el modelo 3D del sensor, se selecciona dicho objeto, y en el inspector se añade un nuevo collider. En este caso se ha elegido el ***capsule collider***, ya que es la forma que mejor se ajusta. Se activa la opción de trigger, se desactiva la textura en este collider para evitar que sea visible (se desactiva el ***Mesh Collider***) y se ajustan los límites de dicho collider sabiendo que se trata del rango de acción del sensor. Y con estos ajustes, ya estaría totalmente preparado el sensor para ser programado.

A continuación se muestra el código de la clase ***SensorComponent***:

```
public class SensorComponent : MonoBehaviour
{
    public string Name;
    public int Number;

    // When change the sensor value, it launchs any functions
    public bool[] flag
    {
        get { return flagValue; }
        set
        {
            flagValue = value;

            if (AutomationParameters.internalAutomation)
```

```
            InternalAutomation.AutomationProcess(ref Automation.Memory);
        }
    }
    private bool[] flagValue;
}
```

4.1.2.2 Programación del script InductiveSensor

Al comienzo de este programa se definen las variables. En este caso sólo se necesita una, de tipo SensorComponent, clase que contiene todos los parámetros necesarios para identificar y clasificar cada sensor de la planta. Todos estos parámetros se inicializan en el método Awake(), definiendo el nombre del mismo como el nombre del objeto (***sensorComponent.Name = transform.name;***). Es por eso que a cada objeto sensor se le ha puesto el mismo nombre acompañado de un índice que hace las veces de índice (***InductiveSensor0***). Como variable de salida se define un vector de longitud unitaria (***sensorComponent.flag = new bool[1];***).

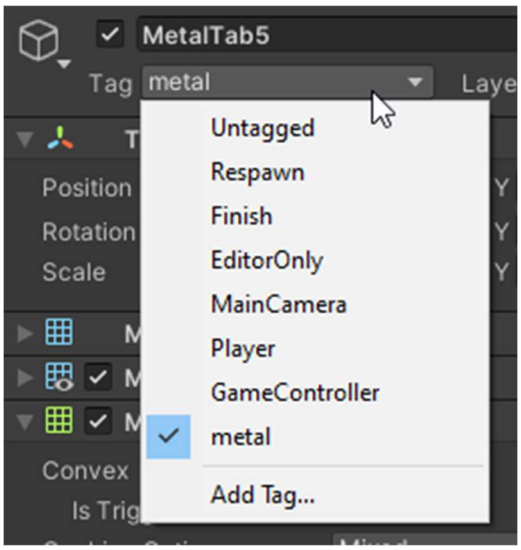


Figura 42. Creación de nuevo tag llamado "metal".

Dentro del método ***Awake()*** se encuentra la rutina de gestión de la variable de salida en base a la detección de objetos. El método ***OnTriggerEnter()*** se ejecuta cuando algún objeto entra en la región definida por el trigger. Dentro de éste, se acude a una estructura de tipo ***if()*** para verificar que el objeto detectado es de tipo metálico. Una vez cumplida la condición, se activa la variable propia de ese sensor. De forma análoga, se hace lo mismo con el método OnTriggerExit(), pero en este caso desactivando la variable asociada.

Para conseguir distinguir un objeto metálico de otro que no lo es, se ha añadido una etiqueta llamada ***metal***, etiqueta que se ha colocado en todos los objetos que deben ser metálicos. Para añadir una nueva etiqueta, basta con seleccionar el objeto, y en la parte superior del inspector se encuentra el campo tag, que permite seleccionar algunas predefinidos o crear una nueva (figura 42).

En este caso, los elementos metálicos a detectar por los sensores son unas piezas metálicas colocadas en las esquinas de cada plataforma (figura 43).

Figura 43. Detalle pieza metálica de plataforma

```
public class InductiveSensor : MonoBehaviour
{
    // DECLARATION OF GLOBAL VARIABLES
    SensorComponent sensorComponent;

    void Awake()
    {
        sensorComponent = gameObject.AddComponent<SensorComponent>() as SensorComponent;
        sensorComponent.flag = new bool[1];
        sensorComponent.Name = transform.name;
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.tag == "metal")
            sensorComponent.flag = new bool[]{true};
    }

    private void OnTriggerExit(Collider other)
    {
        if (other.tag == "metal")
            sensorComponent.flag = new bool[]{false};
    }
}
```

4.1.2.3 Sensor de fuerza

En cuanto al flujo de movimiento de la planta que se está estudiando en este trabajo, hay varios puntos o instantes relevantes y delicados, que son los cambios de dirección de las plataformas en cada una de las esquinas.

Se trata de un instante en el cual la plataforma llega por una de las cintas, entra en un pistón de giro, el cual, en el momento adecuado, eleva o baja la plataforma, hasta alcanzar la altura de la siguiente cinta, momento en el cual la plataforma pasa a ser desplazada por la nueva cinta. Se trata, por tanto, no sólo un cambio en la dirección del movimiento, sino un cambio en la altura. En todo este proceso, el pistón de cambio debe conocer el momento en el que una nueva plataforma ha entrado completamente en el pistón, para comenzar la subida / bajada, y el momento en el que la plataforma ha salido completamente, para regresar a su altura inicial. Este hecho supone, obviamente, el uso de sensores. Sin embargo, existen varios factores que dificultan su uso en esta situación. Primeramente, las chapas metálicas que dispone cada plataforma para ser detectada por los sensores inductivos son de un tamaño muy reducido. Esto dificulta primeramente el hecho de conocer si la plataforma ha entrado o salido totalmente del pistón de cambio, debido a la pequeña superficie de detección. Por otro lado, la variación de la altura también supone un handicap en cuanto al uso de sensores en el tope, ya que, o bien, si los sensores se colocan en una altura fija, el movimiento provocaría la desactivación errónea del sensor, o bien los sensores se fijan con el movimiento de la plataforma, que quizás resulte algo complejo y engorroso. En cualquier caso, esta tarea requeriría el uso de más de un sensor inductivo para evitar la aparición de problemas y conocer en todo momento el estado de la plataforma con precisión.

Aunque puede ser una solución totalmente válida y viable, en este caso, se han colocado en cada una de las esquinas cuatro estructuras acopladas a sensores inductivos, de manera que, por su configuración, transforman la detección por inducción a una detección por contacto o fuerza.

Se trata de una placa longitudinal de un tamaño algo más grande que el ancho de cada una de las plataformas que, colocada de forma vertical, está sujeta a un eje de rotación que permite el giro de dicha pieza. Dicha pieza, en la parte inferior, tiene colocada una chapa metálica similar a la que se ubica en cada una de las plataformas, que es detectada por un sensor inductivo vertical situado justo debajo. El funcionamiento básico de esta estructura se basa en que, al llegar una plataforma y empujar dicha pieza, esta comienza a girar alrededor de su eje de rotación, haciendo que la pieza metálica deje de ser detectada por el sensor, como puede verse en las figuras 44 a y b. Por último, se ha colocado un muelle que hace que, en estado de reposo, la pieza se mantenga en la posición original, menteniendo la pieza metálica en el campo de acción del sensor.

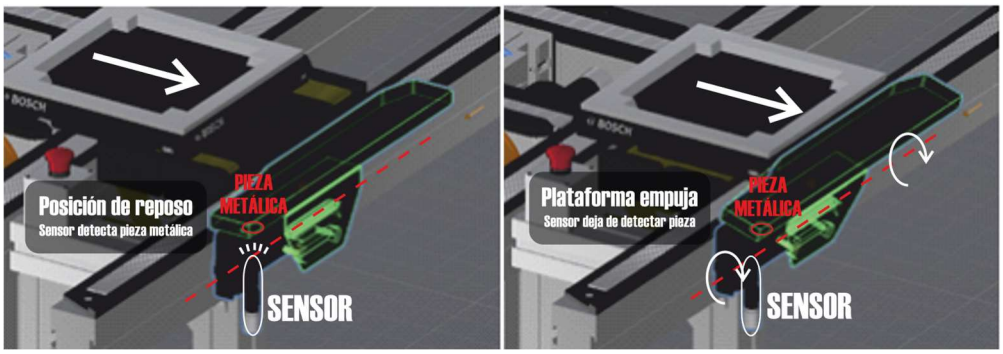


Figura 44. a) Sensor de fuerza en posición de reposo, con sensor detectando pieza metálica y b) Sensor de fuerza girado por empuje de plataforma, sin detectar pieza metálica.

Esto, en primer lugar, hace que la superficie de activación pase a ser mucho mayor, y soluciona el problema de la incertidumbre sobre si la plataforma ha salido totalmente del pistón de cambio o no, ya que el sensor no cambiará de estado hasta que la plataforma deja de estar en contacto con la pieza giratoria. Por otro lado, evita usar más de un sensor, que siempre suele suponer mayor dificultad a la hora de desarrollar el programa de control.

Una vez explicado lo que se llamará a partir de ahora como sensor de fuerza, es el momento de explicar el desarrollo de su simulador en Unity.

Una vez creado el modelo 3D por piezas usando las técnicas explicadas anteriormente, el modelo es exportado a Unity 3D. Las piezas del modelo 3D se han dividido en dos grupos para trabajar en Unity de forma más sencillas. Los grupos son StaticForceSensorModule y DynamicForceSensorModule, que se corresponden con las partes fijas y las partes móviles del sensor, respectivamente. Dentro de la parte estática se encuentra el propio sensor inductivo y el soporte de la estructura. El resto de piezas se agrupan en la parte dinámica.

Una vez importado a Unity, y colocado en la posición correcta, es el momento de configurar todos los parámetros. En primer lugar, se añade el componente ***Rigidbody*** a ambos grupos, sin embargo, en el grupo estático se activa la opción ***Is Kinematic***, que, como se ha comentado anteriormente, desactiva los efectos dinámicos como la gravedad en dicho objeto. Con esto, la parte dinámica ya es dependiente de la gravedad.

Sin embargo, si se ejecuta el programa en este momento, se produciría una colisión entre ambos objetos originada en el eje de rotación. Esto se debe a que, si no se asigna esa unión como una articulación de rotación, Unity 3D no lo interpreta como tal, y comienza a generar fuerzas de interacción entre ambos objetos, hasta romper esa unión. Por tanto, el siguiente paso es configurar correctamente la articulación de rotación en Unity.

Por tanto, seleccionando el objeto que agrupa la parte dinámica, es decir, ***DynamicForceSensorModule***, se añade un nuevo componente denominado ***Hinge Joint***, aquel que genera uniones tipo bisagra entre dos objetos.

Si se observa este componente en el inspector, aparece infinidad de atributos. En primer lugar, el campo ***Connected body*** indica el objeto que compartirá la unión. En este caso, seleccionar el objeto ***StaticForceSensorModule.*** Los dos siguientes parámetros, ***anchor*** y ***axis*** indican la posición de unión del objeto actual, y el eje de rotación de la articulación, ambos parámetros representados en coordenadas locales. En este caso, al crear el modelo 3D de este objeto, se ha colocado el eje de coordenadas justo en el centro de rotación, es decir, en el centro del hueco por donde pasa el eje de la parte estática. De esta forma, la posición de unión de este objeto es el origen de coordenadas.

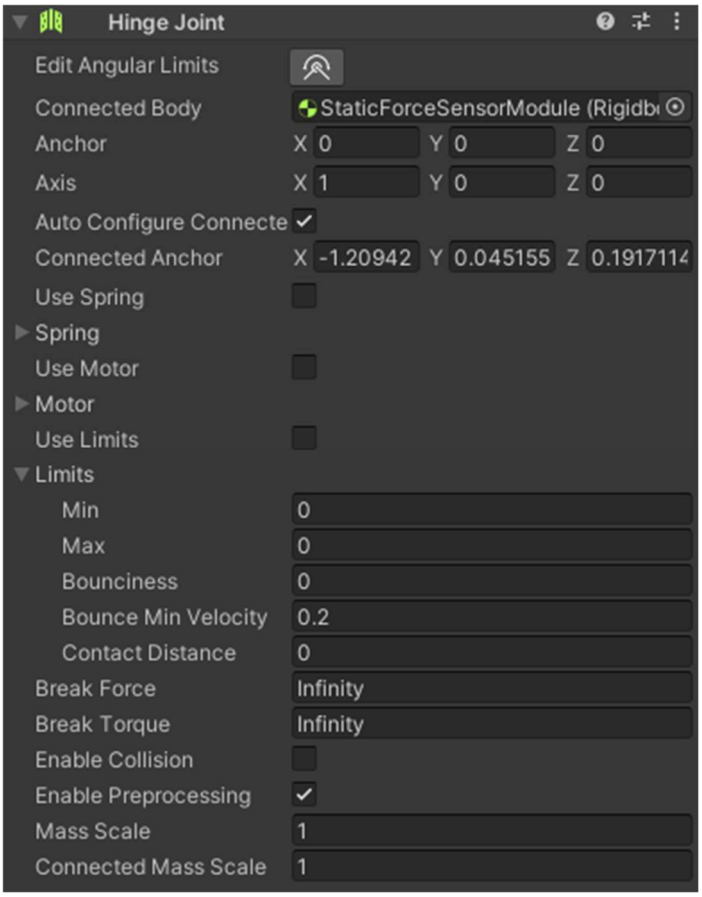


Figura 45. Parámetros de configuración del componente Hinge Joint.

Siguiendo con los parámetros de configuración de la articulación, a continuación también pregunta acerca de la posición de unión del otro objeto. Sin embargo, en este caso existe la opción de que Unity lo calcule de forma automática. En ese caso, tomará como punto de unión el mismo del objeto uno, pero en las coordenadas locales del objeto 2.

El resto de parámetros se han mantenido en su valor por defecto. Este componente también ofrece la posibilidad de definir la fuerza y el torque necesario para romper la unión. Sin embargo, se mantienen por defecto a infinito. De esta forma, ambos objetos ya quedan unidos y la parte dinámica rota libremente dentro del hueco que ofrece el soporte.

El siguiente paso para terminar de modelar el sensor de fuerza es añadir los efectos que proporciona el muelle, el cual hace que la pieza vuelva a la posición original cuando no hay ninguna plataforma empujando.

Para modelar este comportamiento, seleccionando de nuevo el elemento ***DynamicForceSensorModule,*** se debe añadir el componente ***Spring Joint.*** Este componente cuenta con unos atributos muy similares a los de ***Hinge Joint.*** En este caso, el objeto que se elige en el parámetro anchor como objeto que comparte la unión no es ***StaticForceSensorModule***, sino un objeto que forma parte de éste, el cual, a la hora de diseñar el modelo, se ha colocado en la parte inferior del soporte, ahí donde existe la unión en el objeto real. Dicho objeto tiene el nombre ***Cube***. Es por eso que la posición de unión con el otro elemento se concreta en (0,0,0.16), moviéndolo ligeramente en el eje z para cuadrar mejor la posición respecto a la realidad, ya que en este caso, los ejes del objeto Cube no se encuentran en el centro de la pieza sino en el extremo de la misma. De todas formas, la posición de unión es algo orientativo que no necesita de excesiva precisión. La posición de unión de la parte dinámica se calcula para que sea más o menos el centro de la parte superior de la pieza, en concreto, el centro de la pieza llamada ***Arriba***.

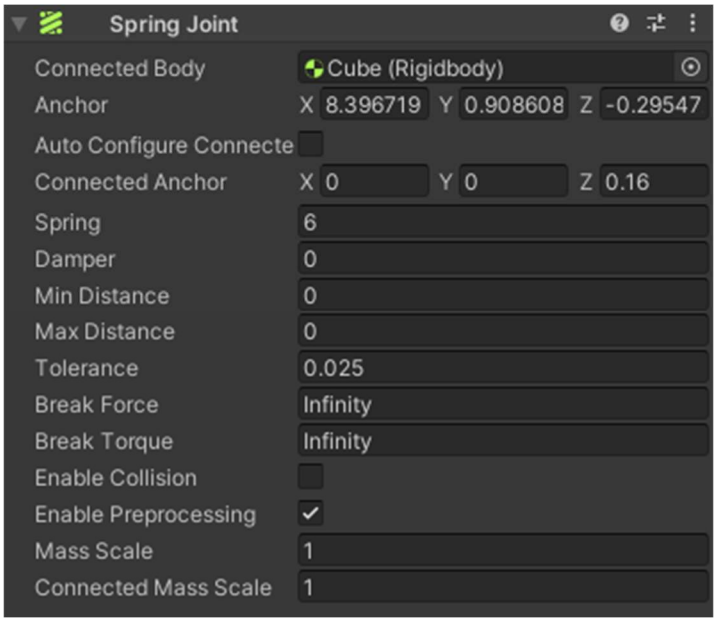


Figura 46. Parámetros de configuración del componente Spring Joint.

El resto de parámetros se deja igualmente con su valor por defecto, a excepción de ***Spring,*** que indica la fuerza del resorte, que se ha elegido un valor de 6. Este valor se ha alcanzado a base de realizar experimentos y tratar de conseguir una fuerza mínima capaz de conseguir devolver la posición original a la parte dinámica.

De esta forma, la parte física del sensor de fuerza estaría completamente configurada.

En cuanto al sensor inductivo que incluye este sensor de fuerza, tiene la misma configuración que el resto de sensores, ejecutando el script ***Inductive Sensor*** y definiendo su rango de acción mediante un capsule collider a modo de trigger.

4.1.3 Programa de inicialización de modelos. Script Inicialization.

Una vez desarrollados todos los modelos analíticos que definen los elementos que conforman la plata, el script ***Initialization*** tiene la función de establecer el estado inicial de todos los elementos, antes de proceder al funcionamiento normal que estaría regido por el pograma de automatización.

A continuación se añade dicho script.

using **System**.**Collections**;

using **System**.**Collections**.**Generic**;

```
using UnityEngine;

//////////////////////////////////////////
// FUNCTIONS THAT INITIALIZE ALL COMPONENTS //
//////////////////////////////////////////
public class Initialization : MonoBehaviour
{
    // FUNCTIONS THAT INITIALIZE ALL MODELS
    public static void GlobalInitialization(Transform transform)
    {
        Models(transform);
        FilteredFiles(transform);
    }

    // FUNCTIONS THAT INITIALIZE ALL MODELS
    private static void FilteredFiles(Transform transform)
    {
        bool[] InitialValues = new bool[transform.GetComponentsInChildren<SensorComponent>().Lengt
h];
        for (int i = 0; i < InitialValues.Length; i++)
            InitialValues[i] = false;
        FilteringFunctions.FileInitialization(FilteringParameters.ReadSensorsPath, InitialValues);
        FilteringFunctions.FileInitialization(FilteringParameters.FilteredSensorsPath, InitialValues);
    }

    // FUNCTIONS THAT INITIALIZE ALL MODELS
    private static void Models(Transform transform)
    {
        InitializeChangePistonsDirection(transform);
    }

    // FUNCTION THAT INITIALIZE THE DIRECTION OF EACH CHANGE PISTONS
    private static void InitializeChangePistonsDirection(Transform transform)
    {
        int j = 1;
        foreach (Transform child in transform)
        {
            foreach (ChangePistonModule CPM in child.GetComponentsInChildren<ChangePistonModule>
())
```

```
            {
                CPM.sentidoCinta = j;
                j *= -1;
            }
        }
    }
}
```

5 SENSORIZACIÓN Y ALGORITMOS DE FILTRADO.

A la hora de hablar sobre un gemelo digital, una idea que destaca es que se trata de un sistema digital que trata de comportarse de la misma forma que el sistema real al que se asemeja. Para lograr este cometido, una vez que se ha creado todo el simulador virtual con todos los parámetros y modelos analíticos, es necesaria una primera fase de aprendizaje, en la cual dicho gemelo, a la vez que se comporta conforme a sus propias reglas y fórmulas, trata de contrastar esas decisiones con las dedl modelo real. Es por ello que necesita alimentarse con información real, que le llega a través de los sensores de la planta real. Con esta información, y mediante uso de redes neuronales, inicia un proceso de acomodación de los modelos analíticos iniciales a la realidad, para así lograr un funcionamiento idéntico. Tras mucho tiempo de obtención de datos procedentes del sistema real, el gemelo puede tratar de, mediante reglas probabilísticas, predecir comportamientos futuros, como posibles fallos, y evitarlos antes de que ocurran.

Es por ello que el aporte de información de los sensores reales es de vital importancia.

Sin embargo, una vez que cada sensor recoge información procedente del entorno, antes de llegar al gemelo, es necesario realizar un proceso de filtrado de dicha información, ya que, dependiendo del dispositivo que se esté empleando, existe una cierta precisión y un porcentaje de que el sensor ofrezca información errónea o falsa. Es importante recalcar que este aspecto depende mucho tanto de los dispositivos de medición que se usen, y por otro lado la aplicación concreta de dicha medición, que puede ser que no necesite demasiada precisión, y pueda despreciarse esta incertidumbre de fallo. De todas formas, el proceso de filtrado es importante para tratar de conseguir que toda o la mayor información que nutre al gemelo se pueda considerar correcta o fiel a la realidad.

En el caso que atañe a este proyecto, se trata de un ejemplo bastante básico, ya que todos los sensores empleados trabajan con información booleana que indica presencia o ausencia de ella. Es por ello que, en este apartado, se va a explicar cómo se ha llevado a cabo el proceso de recolección de la información de todos los sensores, y el traslado a lo que sería el programa de filtrado, para posteriormente enviar la salida al gemelo, para continuar con el proceso, salida que se correspondería con la información de los sensores filtrada. El programa de filtrado no se ha abordado en este trabajo, ya que está en proceso de investigación. En su caso, se ha creado un programa ejemplo en Python en el cual se ha conectado la entrada con la salida, dejando preparado el formato para añadir la rutina de filtrado cuando se finalice su desarrollo.

Por otro lado, el programa de filtrado que se ha usado en este proyecto ha sido desarrollado en Python, generando un fichero externo a Unity3D y con un lenguaje de programación distinto. Esto se ha realizado para mostrar el porceso de traslado de información de una plataforma a otra, ya que esta posibilidad hace que se puedan aprovechar ventajas de cómputo de datos que otras plataformas pueden ofrecer, sin necesidad de perder información alguna.

5.1. Traspaso de la información. Archivos JSON

Para realizar el proceso de traslado de información entre plataformas como Unity y Python, despues de estudiar diferentes maneras, se llegó a la opción de los archivos JSON.

JSON (JavaScript Object Notation) se trata de un formato ligero de intercambio de datos [5]. JSON conforma un formato de texto cuya ventaja se basa en que es completamente independiente del lenguaje de programación que se emplee. Esto hace que hace que JSON sea una alternativa para el intercambio de datos.

El script ***Initialization*** explicado anteriormente también se encarga de inicializar los ficheros JSON dedicados al traslado de la información de los sensores.

5.1.1 Lectura y escritura de archivos JSON

El proceso de traslado de información de los sensores al programa de filtrado externo, y la devolución de los datos filtrados se realiza por medio de dos archivos JSON, uno con la información sin filtrar, y el otro archivo

de retorno con la información ya filtrada. Dichos archivos se han denominado SensorsReading.json y FilteredSensorsReading.json, respectivamente.

El script FilteringSupport se ha dedicado a todos los parámetros y funciones relacionadas con la gestión de la información procedente de los sensores y los archivos JSON. En este script se definen, por un lado, las rutas de los archivos JSON empleados para el envío de información, y por otro lado, los métodos para añadir una nueva lectura de un sensor a una lista de datos que serán posteriormente escritos en el archivo JSON (addSensorValue), escribir en el archivo JSON (writeJSON), y leer el archivo, guardando los datos en una lista con la que se pueda trabajar de nuevo en C# (readJSON).

Además se encuentra el método usado para inicializar los archivos JSON (FileInitialization). Este método crea nuevos archivos JSON si no existen ya de antes, y les define una longitud de datos determinada, definida en el parámetro MAXSIZE, que tiene un valor por de fecto de 50.

Todos estos métodos son definidos para ser usados por la rutina de automatización cuando lo requiera.

A continuación se muestra el código del script ***FilteringSupport().***

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Newtonsoft.Json;
using System.IO;
using RunPythonScriptFromCS;


//////////////////////////////////////////
// PARAMETERS OF ALL MODELS FOR AUTOMATION //
//////////////////////////////////////////
public class FilteringParameters : MonoBehaviour
{
   // JSON FILES PATH
   public const string ReadSensorsPath = @".\Assets\Scripts\Filtering\JSonFiles\SensorsReading.json";
   public const string FilteredSensorsPath = @".\Assets\Scripts\Filtering\JSonFiles\FilteredSensorsReading
.json";


}
//////////////////////////////////////////




//////////////////////////////////////////
// FUNCTIONS FOR SENSOR DATA MANAGEMENT //
//////////////////////////////////////////
public class FilteringFunctions : MonoBehaviour
{
   // Data capacity of JSON file
```

```
    static int MAXSIZE = 50;

  // FUNCTIONS THAT RETURN THE SENSOR COMPONENT
  public static void FileInitialization(string _path, bool[] InitialValues)
  {
    // Sensor reading file is created if it does not exist
    if(!File.Exists(_path))
    {
      List<SensorValue> ObjectList = new List<SensorValue>{};
      SensorValue Object = new SensorValue();
      Object.dateTime = System.DateTime.Now;
      Object.value = new bool[InitialValues.Length];
      for (int i = 0; i < InitialValues.Length; i++)
        Object.value[i] = InitialValues[i];
      ObjectList.Add(Object);
      writeJSON(ObjectList, _path);
    }
  }
  public static bool addSensorValue(bool actualValue, int numSensor, string _readingPath, string _filtere
dPath)
  {
    int size;
    List<SensorValue> SensorList = new List<SensorValue>();
    SensorList = readJSON(_readingPath);
    size = SensorList.Count;

    SensorValue newValue = new SensorValue();
    newValue.value = new bool[SensorList[size-1].value.Length];

    // Values of the last reading is assigned to the rest of the sensors
    for (int i=0 ; i < SensorList[size-1].value.Length ; i++)
      newValue.value[i] = SensorList[size - 1].value[i];

    // New value of indicated sensor is added to the vector
    newValue.value[numSensor] = actualValue;

    // Current date is assigned
    newValue.dateTime = System.DateTime.Now;
```

```
        // If the list reach the size limit, the oldest value is removed
        if(SensorList.Count == MAXSIZE)
            SensorList.RemoveAt(0);

        // The new data vector is added to the list
        SensorList.Add(newValue);

        // File with no filtered data is updated
        writeJSON(SensorList, _readingPath);

        // Filtering routine made in Python is executed
        PythonAndCS.FilteringProcess(_readingPath, _filteredPath);

        // File of already filtered values is read
        List<SensorValue> FilteredSensor = readJSON(_filteredPath);

        // The function returns the filtered value of the sensor
        return FilteredSensor[FilteredSensor.Count-1].value[numSensor];

    }
    public static void writeJSON(List<SensorValue> SensorList, string _path)
    {
        string SensorListArray = Newtonsoft.Json.JsonConvert.SerializeObject(SensorList.ToArray(), Form
atting.Indented);
        File.WriteAllText(_path, SensorListArray);
    }

    static List<SensorValue> readJSON(string _path)
    {
        string SensorListArray;
        using (var reader = new StreamReader(_path))
        {
            SensorListArray = reader.ReadToEnd();
        }

        List<SensorValue> SensorList = Newtonsoft.Json.JsonConvert.DeserializeObject<List<SensorValue
>>(SensorListArray);

        return SensorList;
```

```
    }
}
//////////////////////////////////////////

//////////////////////////////////////////
// CLASS FOR EACH SENSOR READING REGISTER //
//////////////////////////////////////////
public class SensorValue
{
    public bool[] value;

    public System.DateTime dateTime;

}
```

5.2. Ejecución del programa de filtrado

En el momento en el que se completa la escritura del archivo JSON y se deja listo para ser enviado, es el momento de ejecutar el script ***PythonProgramExecute().***

Este script se encarga de, teniendo ambos ficheros JSONlistos, es decir, el archivo con los datos a filtrar, y el segundo archivo JSON preparado para recibir la información filtrada, ejecutar el programa externo, en este caso desarrollado en Python.

Dicho script determina la ruta concreta en la que se ubica el programa externo, y le pasa todos los argumentos necesarios, que en este caso son los dos archivos JSON con la información a filtrar.

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

namespace RunPythonScriptFromCS
{
    ////////////////////////////////////////////////////////
    // FUNCTION FOR CREATE THE CONECTION BETWEEN C# AND PYTHON //
    ////////////////////////////////////////////////////////
    public class PythonAndCS
    {
```

```
public static void FilteringProcess(string _readingPath, string _filteredPath)
{
    // Create process info
    var psi = new ProcessStartInfo();

    psi.FileName = @"\python.exe";

    // Provide script and arguments
    var script = @".\Assets\Scripts\Filtering\FilteringProcess.py";

    // Argument inicialization
    psi.Arguments = $"\"{script}\" \"{_readingPath}\" \"{_filteredPath}\"";

    // Process configuration
    psi.UseShellExecute = false;
    psi.CreateNoWindow = false;
    psi.RedirectStandardOutput = true;
    psi.RedirectStandardError = true;

    // Execute process and get output
    var errors = "";
    var results = "";

    using(var process = Process.Start(psi))
    {
        errors = process.StandardError.ReadToEnd();
        results = process.StandardOutput.ReadToEnd();
    }

    // Display output
    if (errors != "")
    {
        UnityEngine.Debug.Log("ERRORS: ");
        UnityEngine.Debug.Log(errors);
    }
    else
    {
        UnityEngine.Debug.Log("Results");
```

```
            UnityEngine.Debug.Log(results);
        }


    }
  }
}
```

Una vez finalizada la ejecución de este script, el archivo JSON dedicado a la información filtrada obtendría todos los valores filtrados de los sensores. Sin embargo, como se ha comentado, el programa de filtrado se encuentra todavía en desarrollo por investigación. En su caso, se ha desarrollado un pequeño programa de ejemplo en Python, llamado ***FilteringProcess***, que, en primer lugar, transforma el contenido del fichero de entrada en datos procesables en Python, y transfiere dichos datos al fichero de salida, el cual es devuelto a C#. A continuación se muestra el código del programa ***FilteringProcess***.

```
import json
import sys

_path1 = sys.argv[1]
_path2 = sys.argv[2]

# Lectura de valores de sensores sin filtrar
with open(_path1, 'r') as archivo:
  lectura_sensores = json.load(archivo)

##############################################################################
# Filtering algorythm (in this case it is bridged, since the development of the filter is not within the scope of this project).
for i in range(len(lectura_sensores)):
  for j in range(len(lectura_sensores[i]['value'])):
    lectura_sensores[i]['value'][j] = lectura_sensores[i]['value'][j]

##############################################################################

# Save filtered values in filter file, overwriting previous values

with open(_path2, 'w') as archivo:
  json.dump(lectura_sensores, archivo, indent = 4)
```

6 CONCLUSIÓN

Este trabajo desarrolla en profundidad el modelado visual y analítico de varios elementos industriales, mostrando, en primer lugar, la gran versatilidad y calidad del software Cinema 4D a la hora de modelar elementos en 3D.

Por otro lado, desarrolla la implementación de dichos modelos en Unity 3D, software bastante Amplio que permite la total programación de los mismos.

Este trabajo también muestra ejemplos de programación a la hora de desarrollar el comportamiento de los elementos industriales diseñados anteriormente.

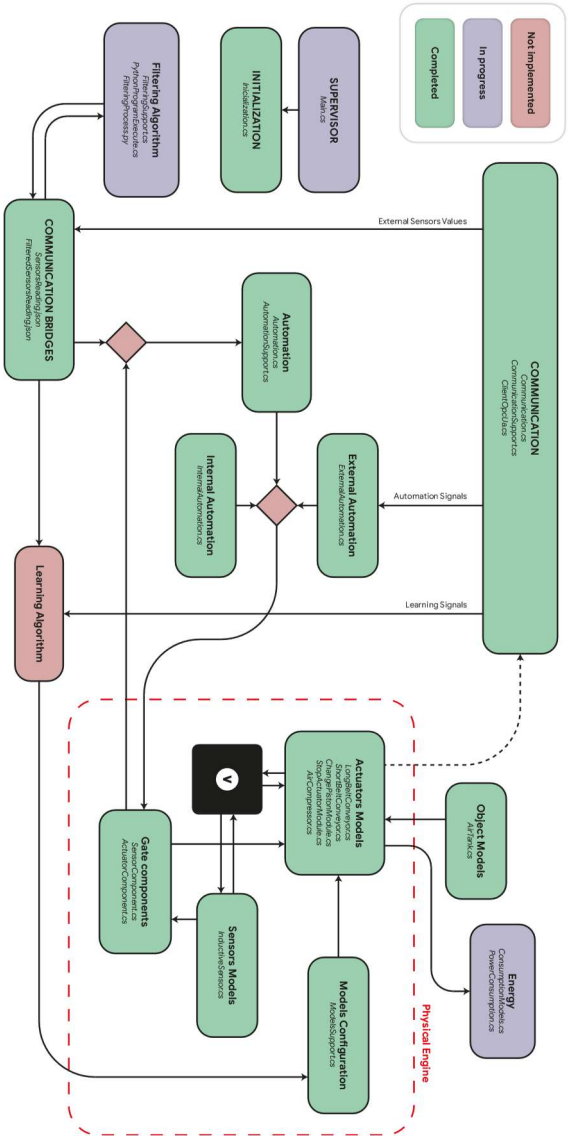
En cuanto a la programación desarrollada, se ofrece la característica de que es modular, es decir, que está dividida por módulos que dividen el programa complete. Esto ofrece multitude de ventajas en cuanto a fututros cambios y adición de nuevos elementos. De hecho, como se ha comentado en alguna occasion, existen numerosas partes todavía en porceso de investigación, que deberán añadirse al Proyecto, y debido a la estructuración del programa no serán necesarios excesivos cambios.

En resumen, este trabajo ofrece un desarrollo relacionado con un método de creación de gemelos digitales, método el cual está en proceso de desarrollo, y muestra bastantes posibilidades en el campo de los gemelos digitales, un terreno con muchos entornos y conceptos nuevos por descubrir.

7 ANEXO 1

Este anexo está formado por el esquema general de la arquitectura interna del gemelo digital implementado.

Nota aclaratoria: hay dos símbolos que no se muestra su significado. El primero es un rombo (no implementado = relleno rojo) que consiste en un análogo de un multiplexador en electrónica. El segundo es el cuadrado con un **V** en el centro: representa el visor donde se encuentran todos los modelos visuales.



8 BIBLIOGRAFÍA

[1] Real Academia Española

URL: https://dle.rae.es/gemelo

[2] Equipo de expertos de la Universidad Internacional de Valencia, 2021. ¿Qué es un gemelo digital?
URL: https://www.universidadviu.com/es/actualidad/nuestros-expertos/gemelo-digital-definicion-funcionamiento-y-ejemplos

[3] Comisión Europea, Cordis, 2020. Proyecto DENiM.

URL: https://cordis.europa.eu/project/id/958339/es

[4] Texturas 3D. Texturas PBR y modelos 3D desde fotogrametría.

URL: https://www.texturas3d.com/fotogrametria/texturas-pbr/

[5] Página oficial de JSON

URL: https://www.json.org/json-es.html