



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

**Integración de gemelos digitales
dentro de un simulador de neumática
basado en realidad virtual**

MATERIAL DIDÁCTICO

Que para obtener el título de

Ingeniero Mecánico

P R E S E N T A

Inti López Lucero

ASESOR DE MATERIAL DIDÁCTICO

M.F. Gabriel Hurtado Chong



Ciudad Universitaria, Cd. Mx., 2024



**PROTESTA UNIVERSITARIA DE INTEGRIDAD Y
HONESTIDAD ACADÉMICA Y PROFESIONAL
(Titulación con trabajo escrito)**



De conformidad con lo dispuesto en los artículos 87, fracción V, del Estatuto General, 68, primer párrafo, del Reglamento General de Estudios Universitarios y 26, fracción I, y 35 del Reglamento General de Exámenes, me comprometo en todo tiempo a honrar a la institución y a cumplir con los principios establecidos en el Código de Ética de la Universidad Nacional Autónoma de México, especialmente con los de integridad y honestidad académica.

De acuerdo con lo anterior, manifiesto que el trabajo escrito titulado INTEGRACION DE GEMELOS DIGITALES DENTRO DE UN SIMULADOR DE NEUMATICA BASADO EN REALIDAD VIRTUAL que presenté para obtener el título de INGENIERO MECÁNICO es original, de mi autoría y lo realicé con el rigor metodológico exigido por mi Entidad Académica, citando las fuentes de ideas, textos, imágenes, gráficos u otro tipo de obras empleadas para su desarrollo.

En consecuencia, acepto que la falta de cumplimiento de las disposiciones reglamentarias y normativas de la Universidad, en particular las ya referidas en el Código de Ética, llevará a la nulidad de los actos de carácter académico administrativo del proceso de titulación.

INTI LOPEZ LUCERO
Número de cuenta: 304099317

Índice

1. Introducción	1
1.1. Objetivos	1
1.2. Alcances	1
1.3. Antecedentes	2
2. Marco Teórico	4
2.1. Realidad virtual en la educación	5
2.2. El simulador virtual NERV	7
2.3. El motor de videojuegos Unity como herramienta para la creación de ambientes virtuales de simulación	19
2.4. El software de modelado Blender como herramienta para agregar texturas a los modelos 3D	32
2.5. La plataforma VRChat como alojamiento del simulador NERV	42
3. Desarrollo de los gemelos digitales	43
3.1. Descripción de los sistemas neumáticos a desarrollar como gemelos digitales	43
3.1.1. Descripción de la Empacadora	43
3.1.2. Descripción de la Indentadora	46
3.1.3. Descripción de la Estampadora	48
3.2. Diseño CAD de los planos de situación neumáticos	50
3.2.1. Modelos CAD de los cilindros neumáticos y sus soportes	50
3.2.2. Modelos CAD de los ensambles de los gemelos digitales	53
3.3. Optimización de los modelos para un menor consumo de recursos computacionales . 81	
3.3.1. Simplificación de los modelos CAD	81
3.4. Adición de texturas a los gemelos digitales	90
4. Integración de los gemelos digitales al simulador virtual NERV	107
4.1. Adecuaciones al simulador virtual NERV	108
4.2. Correcciones al simulador virtual NERV	112
5. Programación de los gemelos digitales	118
5.1. Programación de la empacadora	118
5.2. Programación de la Indentadora	160
5.2.1. Simulación de la indentación con Decals	166
5.3. Programación de la estampadora	173

5.3.1. Animación de la deformación de la pieza de trabajo.....	185
6. Resultados.....	201
6.1. Pruebas de funcionamiento de la empacadora.....	203
6.2. Pruebas de funcionamiento de la indentadora.....	210
6.3. Pruebas de funcionamiento de la estampadora.....	215
7. Conclusiones.....	222
8. Bibliografía.....	223
9. Apéndice A. Planos del Gemelo Digital de la Empacadora.....	228
10. Apéndice B. Planos del Gemelo Digital de la Indentadora.....	249
11. Apéndice C. Planos del Gemelo Digital de la Estampadora.....	264
12. Apéndice D. Imágenes de las Texturas Utilizadas para los Paquetes del Gemelo Digital de la empacadora.....	286
13. Apéndice E. Código Original de la Mesas de Trabajo Neumáticas Proporcionado para la implementación de los Gemelos Digitales.....	296
14. Apéndice F. Código Modificado de la Mesas de Trabajo Neumáticas que controlan los gemelos digitales de la empacadora y la indentadora.....	333

Agradecimientos

Se agradece al Programa de Apoyo a Proyectos para Innovar y Mejorar la Educación, PAPIME con clave PE108323, por el apoyo brindado para la realización de este trabajo.

1. Introducción

Se presenta la integración de los gemelos digitales correspondientes a 3 planos de situación neumáticos: Una empacadora, una indentadora y una estampadora; dentro de un ambiente virtual de simulación de circuitos neumáticos (NERV, Neumática Educativa en Realidad Virtual) desarrollado por estudiantes y profesores de la facultad de ingeniería de la Universidad Nacional Autónoma de México. Los gemelos digitales son controlados a partir de la construcción de los circuitos neumáticos adecuados dentro de NERV, logrando así la visualización del comportamiento de los gemelos digitales (planos de situación) en un ambiente 3D interactivo. Los estudiantes se ven beneficiados, tanto de la práctica construyendo los circuitos neumáticos requeridos para el funcionamiento de los gemelos digitales, así como de la interpretación de las secuencias de movimiento que dan como resultado el adecuado o inadecuado funcionamiento de los gemelos digitales.

1.1. Objetivos

- Integrar los gemelos digitales de los planos de situación neumáticos de una empacadora, una indentadora y una estampadora, en el entorno virtual de simulación de neumática NERV.
- Ayudar a los estudiantes a interpretar de manera adecuada las secuencias de movimiento requeridas por los planos de situación de la empacadora, la indentadora y la estampadora.
- Ayudar a los estudiantes a entender las consecuencias de no implementar los circuitos neumáticos adecuados requeridos por los planos de situación y los diagramas de movimiento.

1.2. Alcances

Se espera que el resultado de este trabajo sirva para que los estudiantes de la facultad de ingeniería adquieran experiencia en la interpretación de las secuencias de movimiento neumáticas requeridas por diversos planos de situación encontrados comúnmente en la enseñanza de la automatización industrial. Además, se espera realizar a futuro más implementaciones de gemelos digitales, con el fin de tener un amplio catálogo de planos de situación y que así los estudiantes tengan un conocimiento sólido de la interpretación de este tipo de diagramas y de la construcción de los circuitos neumáticos requeridos para el funcionamiento de los modelos que representan. La ventaja de hacer esta implementación en realidad virtual es que los alumnos no requieren ingresar al laboratorio de Automatización industrial para adquirir experiencia. Esto resulta conveniente debido a la disponibilidad del laboratorio, pues un gran número de estudiantes hacen uso de las instalaciones delimitando los horarios de ingreso y empleo del equipo. A pesar de que estas

implementaciones están destinadas a su manejo mediante dispositivos de realidad virtual, los usuarios pueden utilizarlas a través de otros dispositivos como computadoras o smartphones, por lo que su uso no se ve limitado a aquellos que posean dispositivos de realidad virtual. Para acceder al “mundo” donde está alojada la presente implementación solo se requiere descargar el programa VRChat® a través de la plataforma de venta de videojuegos Steam® (el programa es gratuito). Muchos estudiantes están familiarizados con estas tecnologías por lo que no encuentran problemas para acceder al simulador virtual.

1.3. Antecedentes

Gemelo digital se refiere a la copia o modelo virtual de cualquier entidad física (gemelo físico), los cuales están interconectados a través del intercambio de datos en tiempo real. Conceptualmente un gemelo digital imita al estado de su gemelo físico en tiempo real y viceversa. La aplicación de los gemelos digitales incluye monitoreo en tiempo real, diseño/planificación, optimización, mantenimiento, acceso remoto, etc. Se espera que su implementación crezca exponencialmente en las próximas décadas [1].

Como se citó en [1], el nombre gemelo digital apareció por primera vez en la versión preliminar de la hoja de ruta tecnológica de la NASA en 2010 donde el gemelo digital se describió como “una simulación probabilística integrada multifísica, multiescala de un vehículo o sistema que utiliza los mejores modelos físicos disponibles, actualizaciones de sensores, historial de flota, etc., para reflejar la vida de su gemelo volador”. En la referencia [1] también mencionan que anteriormente la nasa ya había utilizado un concepto similar para los programas Apolo, en el que se construyeron dos vehículos espaciales idénticos para reflejarse.

En la referencia [2] mencionan que un Gemelo Digital o réplica digital de un producto, servicio o proceso permite adelantarnos al futuro para analizar qué puede pasar y tomar mejores decisiones. Sectores como el de la construcción, la energía, el transporte o el industrial ya están incorporando en su gestión estos gemelos digitales.

Existen un sin número de ejemplos de desarrollos de gemelos digitales. A continuación, se listan algunos ejemplos relevantes de gemelos digitales.

Factory I/O es un simulador comercial de una fábrica 3D, y un ejemplo de gemelo digital, para aprender tecnologías de automatización [3]. Dentro de la página de Factory I/O se describe que esta plataforma está diseñada para ser fácil de usar y permite construir rápidamente una fábrica virtual utilizando una selección de piezas industriales comunes. También incluye muchas escenas inspiradas en aplicaciones industriales típicas, que van desde niveles de dificultad principiante hasta avanzado. También se menciona que el escenario más común de uso para Factory I/O es como una plataforma de capacitación de PLC, ya que los PLC son los controladores más comunes que se encuentran en las aplicaciones industriales. Sin embargo, también se pueden utilizar microcontroladores, SoftPLC, Modbus, entre otras tecnologías. La figura 1.1 muestra una imagen del simulador virtual Factory I/O.



Figura 1.1. El simulador Factory I/O es un simulador comercial destinado al aprendizaje de las tecnologías de automatización y es un ejemplo de un gemelo Digital. Tomado de [3].

En la facultad de ingeniería también se ha implementado este tipo de tecnología. En la referencia [4] se muestra un gemelo digital de un proceso secuencial electroneumático. La figura 1.2 muestra la implementación física y su correspondiente gemelo digital. Para lograr el funcionamiento del gemelo físico y del gemelo digital, se requirió integrar el gemelo digital a la plataforma EmulPro junto con su modelo matemático, el cual también fue desarrollado por los autores. En la referencia [4] se señala que EmulPro es una herramienta, desarrollada en la facultad de ingeniería, capaz de resolver la dinámica de sistemas físicos en tiempo real, empleando un motor de resolución de ecuaciones diferenciales no lineales, también tiene la capacidad de conectarse mediante el protocolo de Comunicaciones de Plataforma Abierta (OPC, por sus siglas en inglés) a un controlador industrial para interactuar de manera bidireccional con el proceso físico, es un entorno que permite la implementación de gemelos digitales. En este trabajo los autores compararon el funcionamiento del gemelo digital y su contraparte física controlando ambos modelos mediante el controlador de automatización programable *ControlLogix* mediante el protocolo de comunicación OPC. También se conectó mediante el mismo protocolo de comunicación el simulador Mathlab Simulink® para validar sus resultados; los cuales mostraron que el gemelo digital funcionaba de manera realista, sin embargo, el protocolo de comunicación utilizado para la comunicación entre el gemelo físico y el gemelo digital, ocasionaba ciertos retrasos en su respuesta, esto porque dicho protocolo de comunicación no estaba optimizado para la plataforma Matlab – Simulink®. A decir de los autores de la referencia [4] este fue un primer paso para incursionar en la digitalización del laboratorio de automatización de la facultad de Ingeniería de la UNAM.

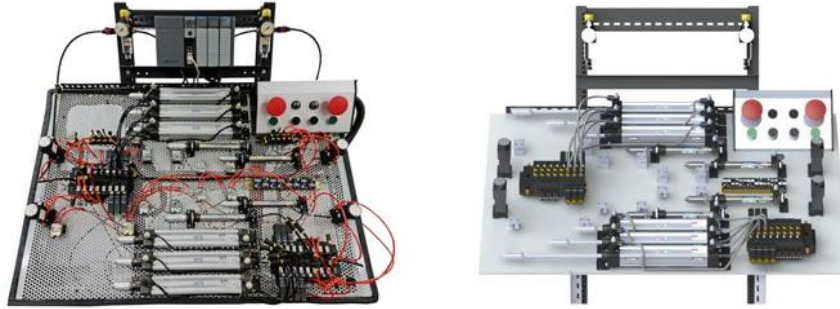


Figura 1.2. Implementación física y el correspondiente gemelo digital de un proceso secuencial electropneumático. Tomado de [4].

2. Marco Teórico

Se presenta a continuación la relevancia de la realidad virtual en la educación, debido a que la integración de los gemelos digitales en el entorno virtual de simulación NERV, tiene como objetivo brindar a los estudiantes una herramienta adicional a las que ya tienen disponibles para que pongan en práctica sus conocimientos y mejoren su comprensión y habilidades, todo desde un entorno de realidad virtual. Por supuesto los estudiantes que no cuenten que dispositivos de realidad virtual pueden acceder a la plataforma mediante computadoras y smartphones. En la actualidad existen muchos casos de estudio de la implementación de la realidad virtual en la educación, así que primero se exploran algunas propuestas interesantes y posteriormente se introduce el simulador virtual NERV, el cual es un entorno virtual de simulación para la creación de circuitos neumáticos creado en la facultad de ingeniería de la UNAM y es en este simulador donde se integraron los gemelos digitales desarrollados en el presente trabajo.

Después se presentan los programas Unity® y Blender®; utilizados en la integración de los gemelos digitales dentro del simulador NERV, y se describen las herramientas utilizadas en cada uno de estos programas para el desarrollo de los gemelos digitales, dejando para una explicación posterior cómo fueron utilizadas estas herramientas en dicho desarrollo.

Por último, se presenta VRChat la cual es una plataforma utilizada para acceder al metaverso y donde se aloja la presente implementación de los gemelos digitales.

2.1. Realidad virtual en la educación

La realidad virtual va más allá de videojuegos y entretenimiento. Un sinnúmero de aplicaciones de la realidad virtual pueden ser encontradas con una búsqueda simple en la web. Aplicaciones en medicina como neurorrehabilitación donde la realidad virtual, imágenes cerebrales y tecnologías de juego son utilizadas para volver a entrenar el cerebro y mejorar la movilidad de las extremidades superiores en pacientes que sufrieron accidentes cerebrovasculares [5]; o pruebas del diseño de productos en ingeniería automotriz, donde la realidad virtual evita el transporte de prototipos, mejora la interacción con los clientes y optimiza la línea de ensamble lo cual conlleva ahorro de tiempo y dinero [6]. En los últimos años el uso de la realidad virtual ha aumentado su auge también en el ámbito educativo, donde podemos encontrar una amplia variedad de casos como: el entrenamiento para el uso correcto de motosierras, donde el simulador busca entrenar a los taladores en la detección de situaciones de riesgo para evitar lesiones e incluso la muerte [7]; el entrenamiento para neurocirugía, donde los simuladores buscan mejorar el entendimiento conceptual de la anatomía compleja y mejorar las habilidades visuoespaciales [8] o el estudio de la implementación de transformadores en subestaciones de plantas de potencia, donde a partir de la realidad virtual se presenta al usuario el equipo y las funciones básicas de los equipos comúnmente encontrados en las subestaciones [9]. En la Universidad Nacional Autónoma de México también se ha implementado este tipo de tecnología enfocada a la educación. Entre los proyectos desarrollados en esta casa de estudios podemos mencionar al observatorio de visualización, Ixtli, el cual es una instalación de realidad virtual cuyo objetivo es introducir, a la comunidad académica de la universidad, los recursos y beneficios de la realidad virtual para la docencia [10]. Dentro de este observatorio se han realizado trabajos como: El proyecto FSAE® (fórmula SAE®), el cual es un proyecto desarrollado en la facultad de ingeniería que permite la integración de tópicos y proyectos del área automotriz en un entorno virtual de simulación [11]; y el proyecto de reconstrucción y visualización 3D del interior de la célula donde, a partir de observaciones experimentales, se presentan modelos 3D de la fagocitosis de eritrocitos por macrófagos [12].

En la referencia [13] se hace un recuento de las aplicaciones de la realidad virtual y se enumeran las ventajas y las desventajas del uso de esta tecnología en el ámbito educativo, Algunas de las ventajas descritas son:

- El docente no solo promueve el conocimiento, sino que lo facilita ya que ayuda a los estudiantes a explorar y aprender.
- Complementa fuertemente la teoría del aprendizaje constructivista porque los estudiantes se sienten empoderados y comprometidos ya que tienen el control sobre el proceso de aprendizaje.

- Los estudiantes pueden aprender experimentalmente y avanzar a su propio ritmo, ya que están explorando un entorno virtual, evitando situaciones en las que se retrasan durante la lección y pasan el resto de la clase tratando de ponerse al corriente.
- Ayuda a los estudiantes a aprender conceptos abstractos porque pueden experimentarlos y visualizarlos. A diferencia del proceso de aprendizaje tradicional, que generalmente se basa en el lenguaje y que es conceptual y abstracto, un entorno de aprendizaje de realidad virtual fomenta el aprendizaje activo lo que ayuda a los estudiantes a comprender los conocimientos abstractos. Los alumnos con poca capacidad espacial se benefician especialmente de la realidad virtual porque las visualizaciones ayudan a reducir la carga cognitiva superflua de los objetivos de aprendizaje.
- Permite a los usuarios la comprensión de sistemas u objetos en diferentes escalas.
- Las situaciones peligrosas pueden ser simuladas en realidad virtual permitiendo a los estudiantes un aprendizaje seguro.
- Prototipos digitales pueden ser copiados, modificados y probados sin los gastos y el tiempo de los que conlleva el uso de los prototipos reales. Esto permite a los estudiantes refinar y probar los diseños de una manera rápida y barata antes de crear un prototipo físico. La realidad virtual también permite probar de una manera fácil diferentes escenarios e hipótesis porque el entorno puede ser diseñado para prevenir variables extrañas que pudieran modificar los resultados y las variables experimentales pueden ser controladas de forma precisa.
- La naturaleza inmersiva de la realidad virtual puede ayudar a bloquear distractores de modo que los estudiantes pueden enfocarse en los objetivos de aprendizaje. La naturaleza inmersiva de la realidad virtual transforma a los estudiantes de aprendices pasivos a aprendices activos, mejorando la motivación y el sentido de control sobre su propio aprendizaje.

Y como limitaciones se tienen:

- Los dispositivos de realidad virtual pueden romperse o bloquearse y el riesgo de que ocurra un mal funcionamiento aumenta a medida que más estudiantes usan el dispositivo.
- Algunos estudiantes podrían tener náuseas, mareos o dolores de cabeza leves mientras usan los dispositivos.
- Se requiere tiempo para que los estudiantes y docentes aprendan a utilizar de manera correcta los dispositivos.
- Se requiere tiempo para construir mundos virtuales o simulaciones para las clases. Para la construcción y mantenimiento de los mundos virtuales se requieren conocimientos técnicos y muchos docentes no tienen el tiempo ni las habilidades técnicas para crear sus propias aplicaciones de realidad virtual por lo que se tendría que recurrir a terceros.
- La realidad virtual no reduce la importancia de la planificación de lecciones o el papel del maestro en la instrucción de la clase. La orientación del docente sigue siendo fundamental cuando se utilizan sistemas de realidad virtual.
- Hay algunos casos en los que la realidad virtual no es el mejor método para lograr un objetivo de aprendizaje, por lo que es esencial mirar el plan de estudios del curso y

determinar dónde puede ayudar la realidad virtual y dónde son más apropiados otros métodos de enseñanza.

- La integración de la realidad virtual en un plan de estudios puede ser difícil y muchos profesores pueden resistirse a usar la nueva tecnología.
- Si las herramientas de realidad virtual son difíciles de usar, puede desalentar a los docentes a implementarlas en el aula. Además, dado que es posible que muchos docentes no hayan estado expuestos a las capacidades o aplicaciones de la realidad virtual en el salón de clases, se debe usar una forma de educación profesional para que los docentes se sientan cómodos usando la tecnología en su salón de clases y explorando las nuevas posibilidades que abre la realidad virtual.

2.2. El simulador virtual NERV

Neumática Educativa en realidad virtual (NERV), es un proyecto de neumática en realidad virtual desarrollado por estudiantes y profesores pertenecientes a la facultad de ingeniería de la UNAM para su uso y aprovechamiento dentro del ámbito académico. Su objetivo es familiarizar a los estudiantes con los distintos elementos neumáticos, que comprendan su funcionamiento y que identifiquen correctamente cada una de sus conexiones. Este proyecto comienza en el año 2020 y toma auge como consecuencia de la contingencia sanitaria ocasionada por el virus SARS-CoV-2 y la necesidad de involucrar a los estudiantes en el estudio de la neumática en un ambiente interactivo y más apegado a la realidad que la proporcionada por los simuladores disponibles a la fecha, los cuales suelen ser bidimensionales y sus elementos no dejan de ser meras abstracciones de los componentes físicos reales [14]. La figura 2.1 muestra algunos de los símbolos neumáticos utilizados por uno de estos programas ampliamente extendido, FluidSim®, proporcionado por la empresa Festo®. A pesar de ser una herramienta valiosa para el diseño de circuitos neumáticos, no es de utilidad para los estudiantes novatos que buscan implementar físicamente dichos circuitos; ya que estos símbolos, muchas veces alejados del aspecto real de los componentes que representan, ocasionan en los estudiantes confusión lo que provoca que cometan errores cuando deben construir físicamente los circuitos neumáticos. En vista de esta problemática y de la poca disponibilidad y costo del software comercial que pudiera solventar esta necesidad es como surge el proyecto NERV, el cual hasta la fecha cuenta con tres etapas documentadas: SiNeRV1.0, SiNeRV2.0 y LavNeuM. La figura 2.2 muestra una estación de trabajo neumática del Laboratorio de Automatización Industrial de la Facultad de Ingeniería perteneciente a la Universidad Nacional Autónoma de México; la cual sirvió como base para el diseño y desarrollo de las mesas de trabajo neumáticas virtuales del proyecto NERV en todas sus etapas, cada una de las cuales fue desarrollada con el motor de videojuegos UNITY®. Los elementos mostrados en la mesa de trabajo neumática de la figura 2.2 incluye: la unidad de mantenimiento, cilindros neumáticos de doble efecto con sus racores, un cilindro de simple efecto, finales de carrera, válvulas 5/2 biestables de accionamiento neumático, botones pulsadores y enclavados, el piloto neumático, temporizadores (uno positivo y otro negativo), válvulas lógicas, etc.

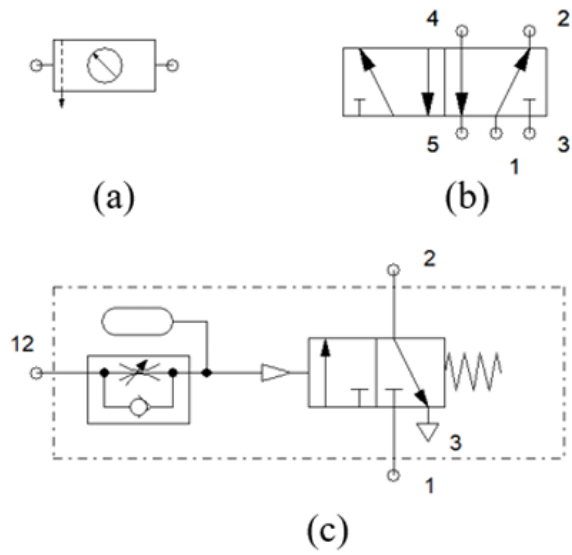


Figura 2.1. Símbolos neumáticos estandarizados y disponibles en el software de diseño de circuitos neumáticos FluidSim® de Festo®. a) unidad de mantenimiento, b) válvula 5/2, c) temporizador positivo. Tomado de [14].

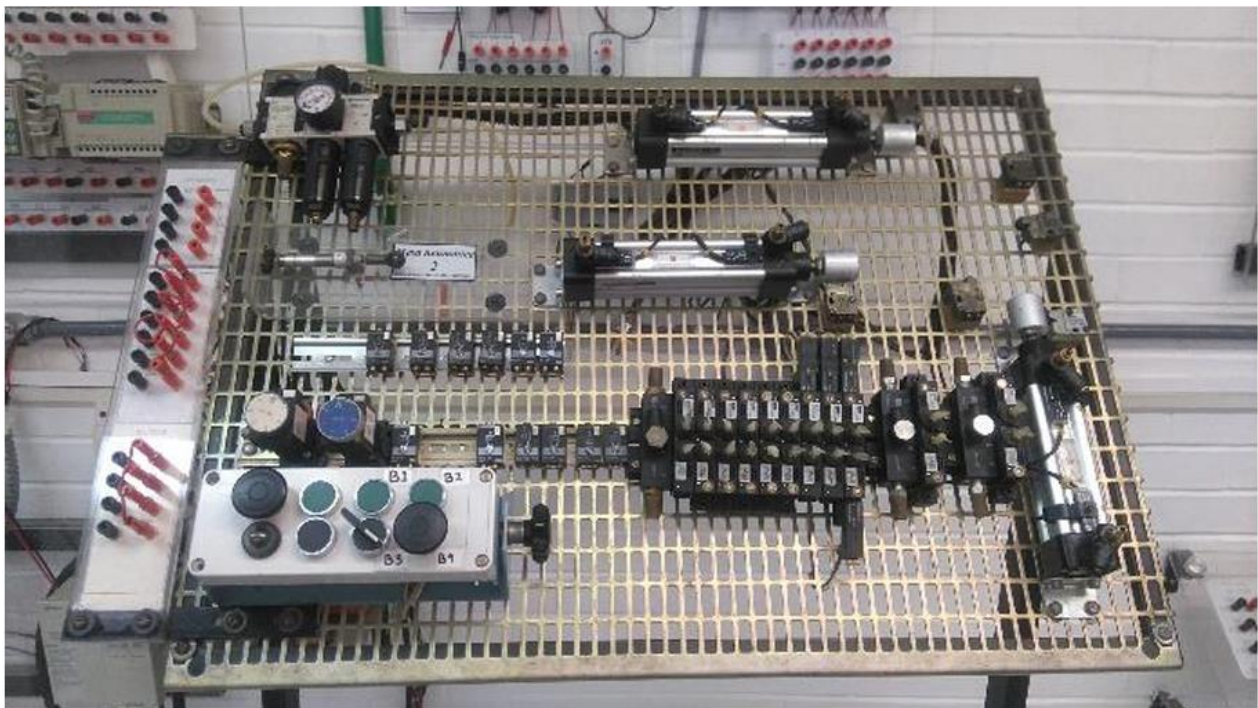
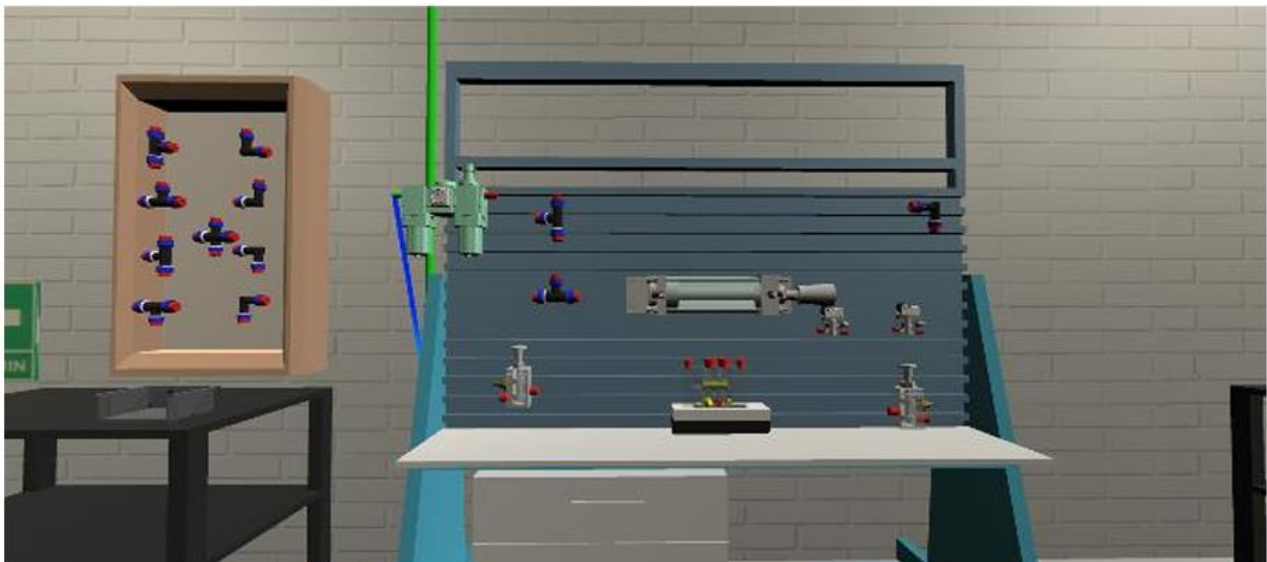


Figura 2.2. Estación de trabajo neumática del laboratorio de automatización industrial de la facultad de Ingeniería de la UNAM. Tomado de [14].

Para la primera etapa del proyecto NERV, el SiNeRV1.0, se desarrollaron estaciones de trabajo neumáticas como la mostrada en la figura 2.3, donde podemos observar los siguientes elementos: La unidad de mantenimiento, dos válvulas 3/2 normalmente cerradas de accionamiento por botón pulsador, dos válvulas 3/2 normalmente cerradas de accionamiento por rodillo o finales de carrera, un cilindro de doble efecto con sus racores, una válvula 5/2 biestable de accionamiento neumático y diversos conectores en T, codo y cruz.

La figura 2.4 muestra todos los modelos virtuales desarrollados para esta primera implementación de las estaciones de trabajo neumáticas, que incluyen: un cilindro neumático de simple efecto, un cilindro neumático de doble efecto, válvulas 3/2 normalmente cerradas accionadas por botón pulsador, válvulas 3/2 normalmente cerradas accionada por rodillo (final de carrera), una válvula 5/2 biestable de accionamiento neumático, f) la unidad de mantenimiento y g) conectores en T, codo y cruz.

La figura 2.5 muestra una implementación simple de esta mesa de trabajo, en donde por medio de botones pulsadores se controla la posición extendida y retraída del cilindro de doble efecto.



**Figura 2.3. Estación de trabajo neumática en realidad virtual desarrollada para el SiNeRV1.0.
Tomado de [14].**

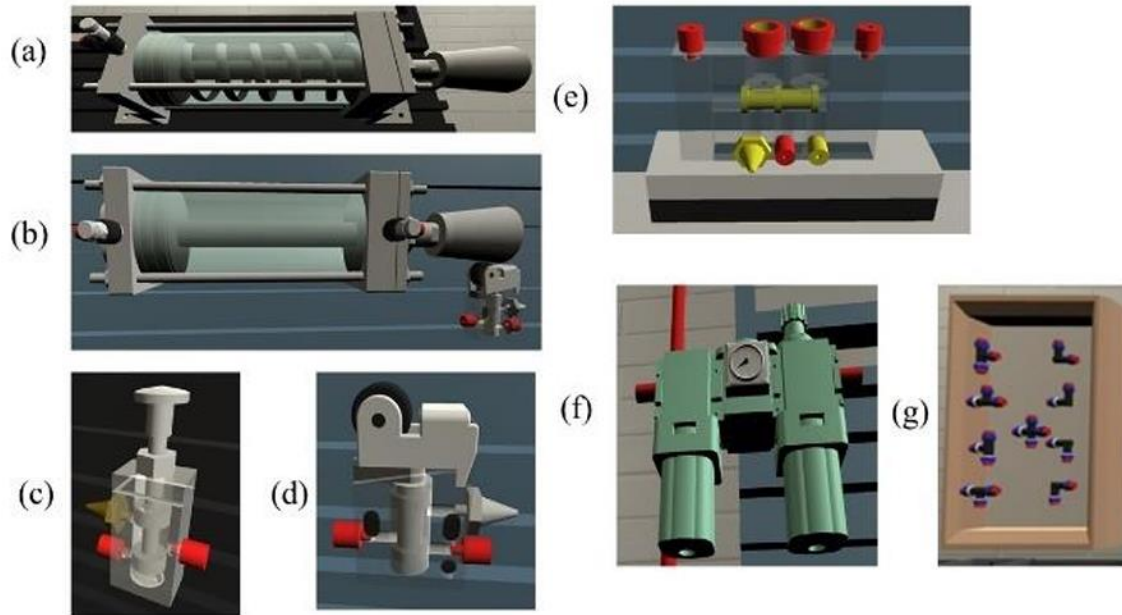


Figura 2.4. Elementos creados para las estaciones de trabajo neumáticas virtuales. a) Cilindro de simple efecto, b) cilindro de doble efecto, c) botón pulsador, d) final de carrera, e) válvula 5-2 de accionamiento neumático, f) unidad de mantenimiento y g) conectores en T, codo y cruz.
Tomado de [14].

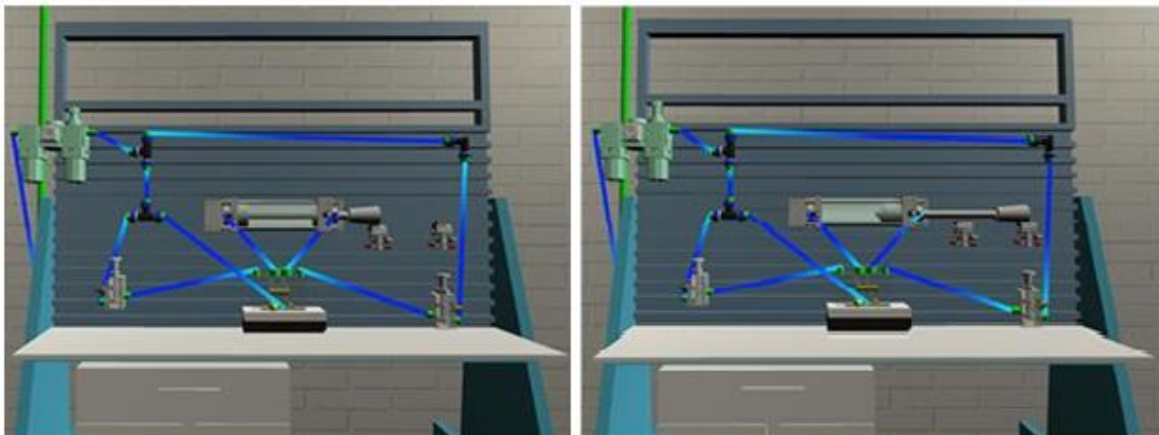


Figura 2.5. A la izquierda se muestra el cilindro neumático en su posición de reposo y a la derecha en su posición de trabajo. El movimiento del vástago se controla con los botones pulsadores. Tomado de [14].

Esta primera implementación del simulador de neumática virtual ofrecía un gran potencial en diversos aspectos como:

- Favorecer el aprendizaje de los estudiantes de una forma más apegada a la realidad que lo permitido por los simuladores neumáticos comerciales que se basan en la simbología neumática estandarizada y que son bidimensionales
- Eliminar la necesidad de contar con infraestructura física, lo cual facilita el aprendizaje fuera de los horarios de clase
- Reducción de costos al eliminar el uso de electricidad, aire comprimido y el desgaste del equipo, además de contribución al medio ambiente al reducir la adquisición, reemplazo y desecho del equipo
- Ampliar el catálogo de componentes virtuales adicionales a las disponibles en el laboratorio
- Generar simulaciones orientadas a la solución de problemas específicos de automatización
- Eliminación del ruido ocasionado por los elementos neumáticos al poder regular el volumen a través de la computadora.

Debido a la suspensión de actividades en la universidad ocasionada por la pandemia de Covid-19, SiNerV1.0 no tuvo pruebas de funcionamiento con los estudiantes y profesores y su acceso a través de internet fue pospuesto, por lo que solo se podía acceder a él de manera local. Estos aspectos debieron corregirse en etapas posteriores del proyecto.

La segunda etapa de desarrollo del proyecto NERV, la cual fue nombrada SiNerV2.0 e inscrita en el Registro Público de Derechos de Autor, tuvo como propósito superar las limitaciones encontradas en la primera etapa. Al simulador SiNerV2.0 podía accederse a través de internet, con lo cual se superó una de las limitaciones del SiNerV1.0, al cual solo podía accederse de manera local en las computadoras donde estaba instalada la aplicación. Para lograr que estudiantes y profesores pudieran acceder al SiNerV2.0 desde internet; se realizó una copia de la aplicación con el estándar WebGL® (desde el mismo UNITY®), la cual fue alojada en la plataforma Itch.io®, que está destinada a la publicación de diversos contenidos digitales creados por desarrolladores independientes [15]. Además, en el SiNerV1.0 únicamente podían desarrollarse dos de un total de las siete prácticas de neumática correspondientes al laboratorio de automatización industrial impartido en la facultad de ingeniería, lo cual fue resuelto en el SiNerV2.0 agregando los componentes faltantes requeridos. La figura 2.6 muestra una de las mesas de trabajo neumáticas virtuales desarrolladas para el SiNerV2.0 donde se pueden observar varios elementos que se tomaron del SiNerV1.0 y los nuevos elementos desarrollados para el SiNerV2.0. Los componentes que se pueden observar en la mesa de trabajo neumática virtual de la figura 2.6 son listados a continuación a partir de la numeración mostrada en la figura: 1) unidad de mantenimiento, 2) Un cilindro de simple efecto, 3) dos cilindros de doble efecto, 4) cuatro finales de carrera, 5) una válvula 3/2 biestable de accionamiento manual (botón rojo), 6) dos válvulas 3/2 monoestables de accionamiento manual (botón azul), 7) cuatro

válvulas 5/2 biestables de accionamiento neumático, 8) una válvula 5/2 monoestable de accionamiento neumático, 9) un temporizador neumático positivo (rojo), 10) un temporizador neumático negativo (azul), 11) una válvula lógica O (OR) y 12) una válvula lógica Y (AND). Los componentes nuevos que fueron agregados para el SiNerV2.0 respecto del SiNerV1.0 son: la válvula 5/2 monoestable de accionamiento neumático, los temporizadores positivo y negativo y las válvulas lógicas Y (AND) y O (OR). También fue agregado un nuevo conector (tipo manifold o distribuidor) a los ya existentes en el SiNerV1.0 el cual se muestra en la figura 2.7. Adicionalmente a la mesa de trabajo neumática mostrada en la figura 2.6, se agregó la mesa de trabajo mostrada en la figura 2.8, cuyo objetivo era la elaboración de circuitos neumáticos empleando el método cascada y, para lo cual se dispusieron los elementos de forma especial lo que facilitaba la elaboración de circuitos neumáticos mediante la aplicación de esta técnica. La mesa de trabajo mostrada en la figura 2.8 no cuenta con elementos nuevos respecto a la mesa de trabajo mostrada en la figura 2.6, pero fueron agregados algunos elementos adicionales: un cilindro de doble efecto, dos finales de carrera, dos válvulas 5/2 biestables de accionamiento neumático, cinco válvulas OR y cinco válvulas AND. Con la adición de esta mesa fue posible la realización de todas las prácticas del laboratorio de automatización industrial, con lo cual se solucionó otra limitación respecto del SiNerV1.0.

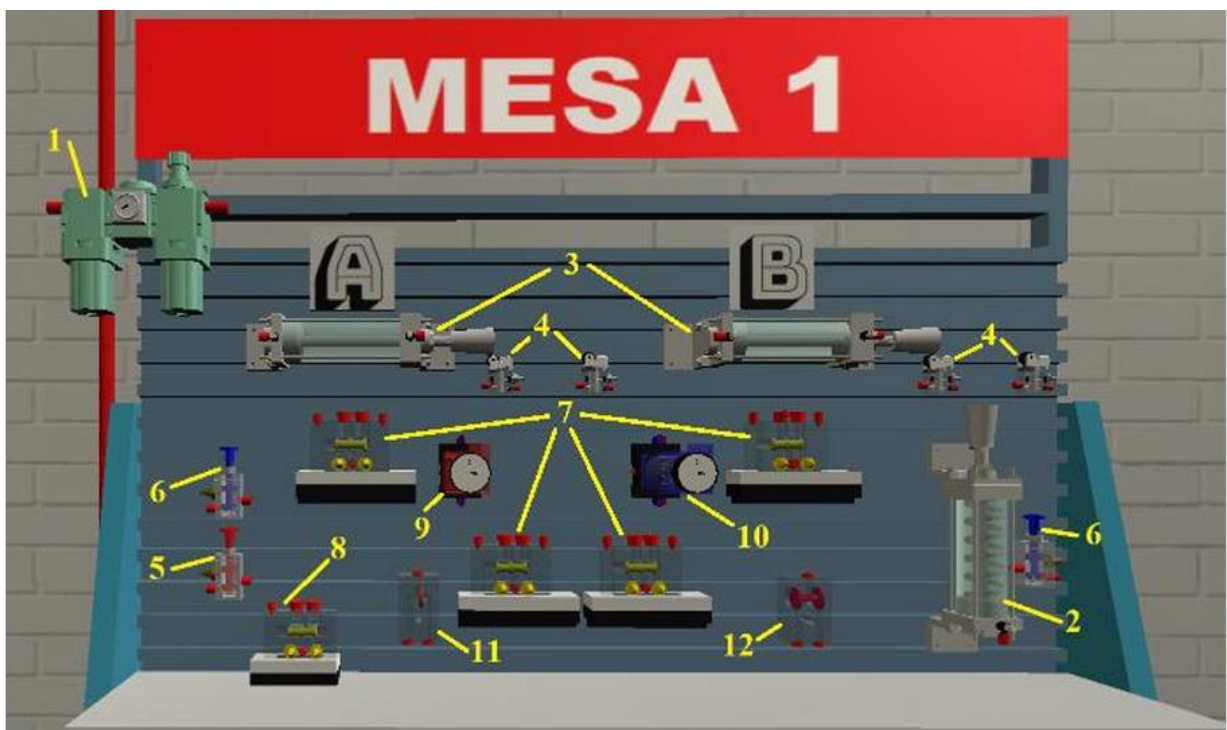


Figura 2.6. Mesa de trabajo neumática desarrollada para el SiNerV2.0. Tomado de [15].

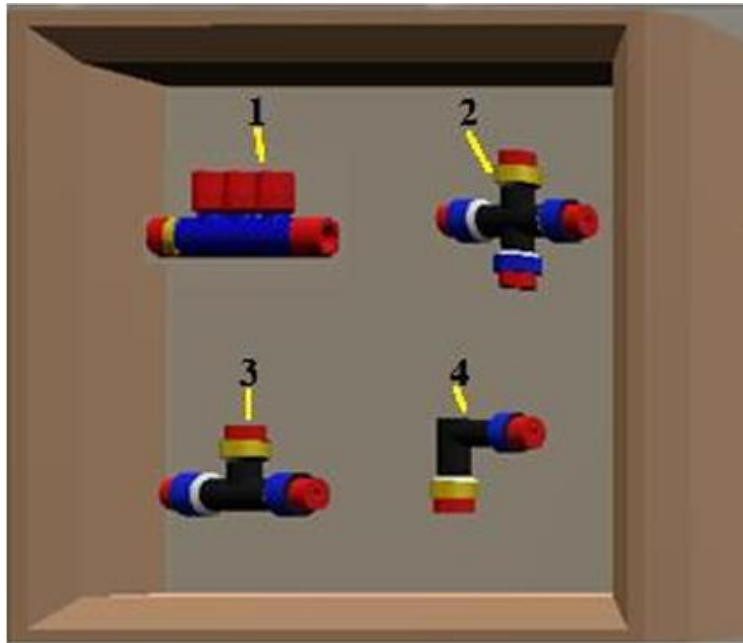


Figura 2.7. Conectores disponibles para el SiNerV2.0. 1) Conector manifold o distribuidor, 2) conector de cruz, 3) conector en T, 4) conector de codo a 90°. Tomado de [15].



Figura 2.8. Mesa de trabajo neumática para la elaboración de circuitos neumáticos implementando el método cascada. Tomado de [15].

SiNeRV2.0 fue probado por un grupo de control formado por estudiantes y docentes teniendo buena aceptación entre los participantes, los cuales consideraron al SiNeRV2.0 como una herramienta didáctica que realmente favorecía el proceso de aprendizaje. Su uso a través de internet permitía acceder a él en cualquier horario, lo cual permitía a los estudiantes utilizarlo en el momento que tuvieran disponible y desde cualquier lugar con acceso a internet. La semejanza con un videojuego le dio un carácter lúdico y entretenido, por lo que los estudiantes lo consideraron como una herramienta motivante. Además, al ser una herramienta desarrollada dentro de la misma facultad, dio un sentido de pertenencia a los alumnos de la facultad haciéndolos sentir orgullosos de pertenecer a la institución. La evaluación general de la facilidad de uso del SiNeRV2.0 por parte de los alumnos fue de 3.7 mientras que la de los profesores fue de 4, en una escala de 0 a 5 donde 0 era muy difícil y 5 muy fácil [15].

Por otro lado, todavía había algunos puntos con los cuales seguir trabajando. Entre los cuales estaban la facilidad de uso del programa, ya que muchos de los estudiantes consideraron que era un tanto complicado hacer y borrar las conexiones neumáticas. Los conectores neumáticos solo podían conectarse de un determinado modo para que funcionaran de manera correcta (un puerto era exclusivamente de entrada), por lo que se sugería que pudieran ser alimentados indistintamente por cualquiera de sus puertos de conexión. Resultaba conveniente reducir los requerimientos necesarios para la ejecución del programa ya que algunos alumnos tuvieron problemas para conseguir un rendimiento adecuado en sus equipos. Era necesario agregar más formas de asistencia al usuario: hacer las instrucciones más claras, incluir ejemplos, tutoriales y ayuda contextual. La mayoría de las funciones deberían ser controladas con el uso exclusivo del ratón para tratar de limitar o eliminar el uso del teclado. Era deseable que los usuarios pudieran armar una mesa o estación de trabajo a su gusto, en vez de tenerlas prediseñadas para que cada usuario pudiera agregar, mover o eliminar componentes de acuerdo a sus necesidades.

La tercera etapa de desarrollo del NERV fue nombrada Laboratorio Virtual de Neumática en el Metaverso (LaVNeuM). Uno de principales objetivos de esta etapa de desarrollo fue brindar a los estudiantes una experiencia más cercana a la realidad del armado físico de los circuitos neumáticos, permitiéndoles asociar los símbolos con los componentes reales y ayudarles a identificar plenamente cada una de las vías de conexión. A diferencia de las primeras etapas del proyecto donde las aplicaciones eran monousuario, LaVNeuM ofrecía la ventaja de ser multiusuario lo cual permitía al profesor interactuar con los estudiantes en tiempo real. Para lograr esto, LaVNeuM fue implementado en la plataforma social 3D VRChat®, la cual ofrecía varias ventajas para la interacción entre los estudiantes y profesores como: hablar, desplazarse, dibujar, sujetar objetos, abrir y cerrar puertas, reproducir música y videos, etc., todo a través de avatares. La decisión de utilizar VRChat® estuvo fuertemente influenciada por las grandes ventajas que ofrecen las herramientas que incorpora esta plataforma para la programación de interacciones a través de un kit de desarrollo de software (SDK) compatible con Unity®. Debido a esta compatibilidad, los elementos creados para las primeras etapas de desarrollo pudieron ser reutilizados en esta nueva implementación. La figura 2.9 muestra una mesa de trabajo neumática implementada para el LaVNeuM, donde se incluyen los siguientes componentes: Unidad de

mantenimiento, indicador de presión (piloto neumático), un cilindro de simple efecto, un cilindro de doble efecto, dos válvulas 3/2 monoestables accionadas por rodillo (finales de carrera), cuatro conectores en T, dos válvulas 3/2 biestables de accionamiento manual, una válvula 5/2 biestable de accionamiento neumático y una válvula 5/2 monoestable de accionamiento neumático. Las figuras 2.10 y 2.11 hacen un acercamiento a las mesas de trabajo neumáticas para poder visualizar de mejor manera los elementos neumáticos de las mesas. En la figura 2.10, siguiendo la numeración mostrada, se tienen: 1) Indicador de presión, 2) conectores en T, 3) cilindro de simple efecto, 4) cilindro de doble efecto, 5) finales de carrera. En la figura 2.11 se pueden observar las válvulas 5/2 de accionamiento neumático, una monoestable y la otra biestable.



Figura 2.9. Mesa de trabajo neumática implementada para el desarrollo del LaVNeuM. Tomado de [16].

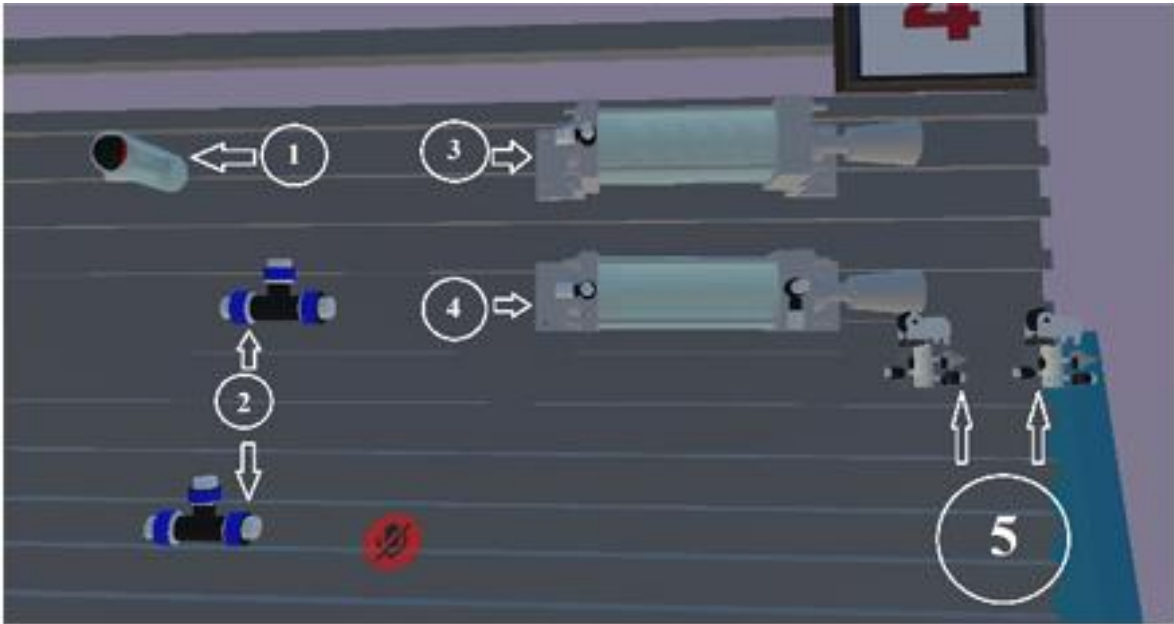


Figura 2.10. Acercamiento a la mesa de trabajo de la figura 2.7 para visualizar mejor sus elementos. Tomado de [16].



Figura 2.11. Acercamientos a las válvulas 5/2 virtuales del LaVNeuM. Válvula 5/2 biestable de accionamiento neumático (izquierda) y válvula 5/2 monoestable de accionamiento neumático (derecha). Tomado de [16].

El LaVNeuM tuvo mejoras notables con respecto las primeras etapas de desarrollo. Una de las más significativas fue la interactividad entre los usuarios y con el entorno de trabajo, ya que VRChat ofrecía de manera nativa estas características. De este modo los estudiantes podían comunicarse entre sí durante el uso de la aplicación para externar dudas y los profesores podían brindar asesoría. Esto era imposible en las primeras etapas de desarrollo donde las aplicaciones eran monousuario y sólo era posible la comunicación entre los usuarios si estos mantenían contacto por otros medios. La experiencia de usuario tuvo una mejora sustancial al permitir a los usuarios conocer qué objetos eran susceptibles de interacción por medio de mensajes emergentes que aparecían conforme el usuario se acercaba a ellos (Fig. 2.12). Otra característica que se implementó en esta etapa de desarrollo fue el cambio de color de las mangueras cuando el flujo de aire pasaba a través de ellas. En la misma figura 2.12 es posible ver este efecto. La imagen izquierda de la figura 2.12 representa a la unidad de mantenimiento cuando no está activada la válvula de paso. En este caso podemos observar que la manguera es de un color azul claro lo cual representa que no ocurre flujo de aire a través de la conexión. Por otro lado, la imagen de la derecha de la misma figura representa a la unidad de mantenimiento activada, por lo que hay paso de aire y la manguera se torna de color azul oscuro, lo cual representa flujo de aire a través de la manguera. Se agregaron además indicadores visuales y sonoros para la localización de fugas de aire cuando existían conexiones abiertas. La figura 2.13 muestra esta característica, donde se puede apreciar el cambio de color de la conexión abierta de blanco a rojo. Esta indicación visual va acompañada de un sonido de fuga de aire, lo que hace más evidente que no existe conexión en esa vía.

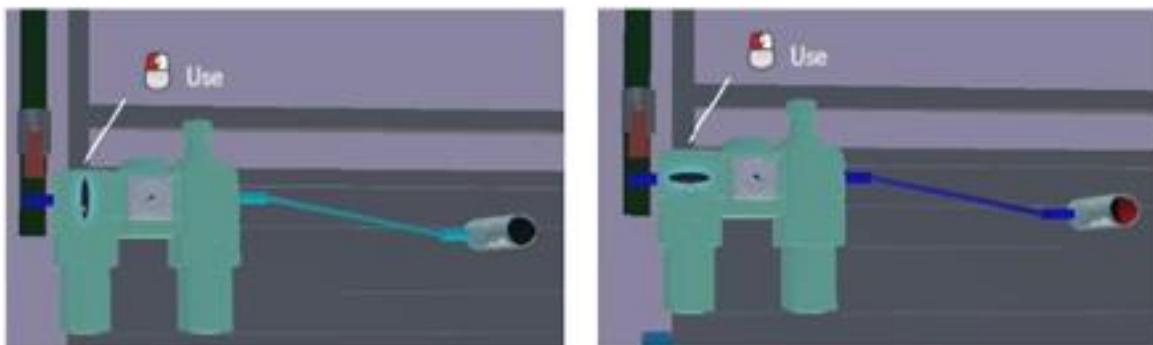


Figura 2.12. Mensajes emergentes de ayuda contextual, donde se indica que el objeto es susceptible de interacción con el usuario, en este caso activar y/o desactivar la unidad de mantenimiento. Puede observarse el cambio de color de la manguera lo que representa flujo de aire a través de ella. Tomado de [16].

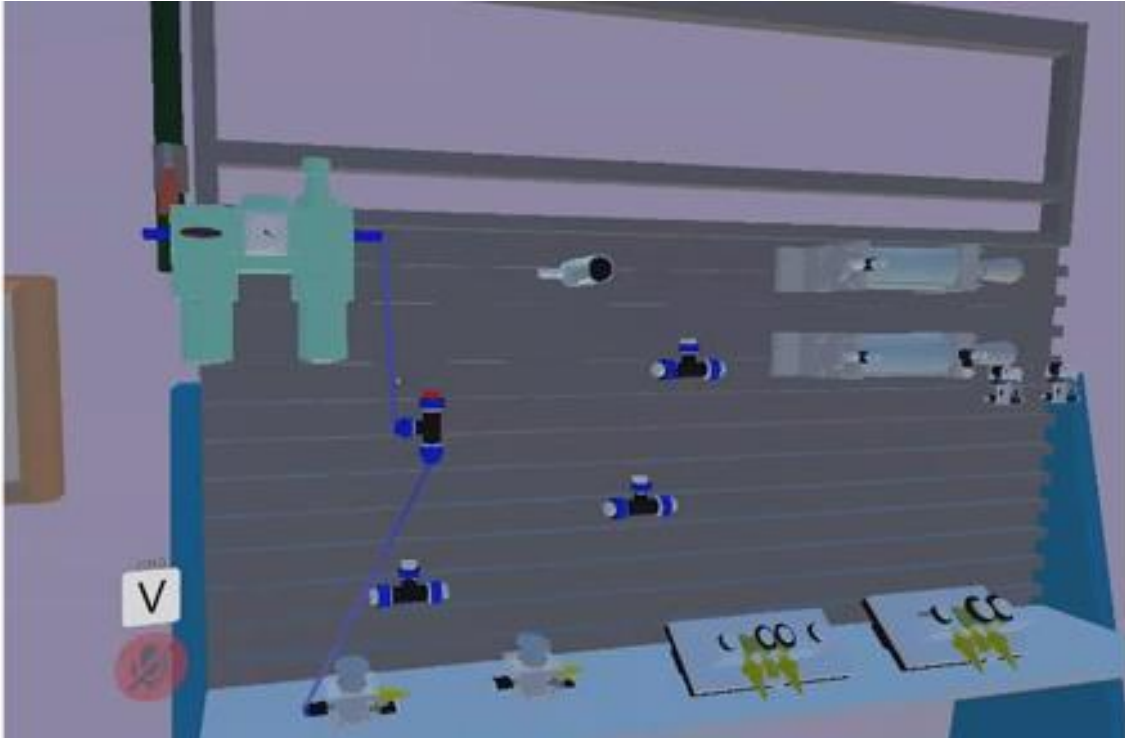


Figura 2.13. Circuito neumático donde se ha dejado intencionalmente una conexión abierta, del conector en T, para ejemplificar la fuga de aire. La vía se torna de color rojo, y ocurre un sonido de fuga a la par. Tomado de [16].

Para la evaluación del simulador LaVNeuM, se realizaron pruebas demostrativas con un total de 16 alumnos que se encontraban cursando en línea la asignatura de automatización industrial, los cuales nunca habían visto ni utilizado el equipo real. A todos los alumnos se les proporcionó una encuesta de evaluación del simulador, sin embargo, solo 5 de ellos pudieron utilizarlo debido a que los restantes no contaban con el equipo adecuado para ejecutar la aplicación. Los alumnos que no pudieron probar la aplicación fueron espectadores a través de videoconferencia. De los 5 alumnos que pudieron probar el simulador uno se negó a contestar la encuesta de opinión. Los cuatro alumnos que aceptaron responder la encuesta evaluaron de bueno a muy bueno el funcionamiento general; así como la facilidad de uso del software, por su parte más del 90% de los espectadores tuvieron la misma apreciación [16].

A la par que se desarrollaba LaVNeuM se realizó la ampliación y el mejoramiento de SiNerV, con lo cual se pudieron superar y resolver problemas propios del LaVNeuM. Aún falta incorporar elementos desde el SiNerV al LaVNeuM, por lo que dista de estar completo, pero una vez que pueda estar completo el proyecto se espera que este espacio en el metaverso coadyuve para que tanto estudiantes y profesores superen algunas limitantes físicas a las que muchas veces se enfrentan en el laboratorio presencial, como: la falta de equipo, espacio limitado y los horarios de acceso restringido.

Cabe resaltar que la creación de este tipo de espacios virtuales como el SiNerV y el LaVNeuM de ninguna forma plantea suprimir el uso de los simuladores tradiciones como FluidSim® o

del armado físico de los circuitos neumáticos en el laboratorio, no obstante, se espera que el realismo alcanzado complemente favorablemente el proceso de formación de los estudiantes y que facilite su primer acercamiento al manejo del equipo neumático real.

En suma, el Laboratorio Virtual de Neumática en el Metaverso (LaVNeuM) aprovecha las ventajas que ofrece el metaverso en el campo de la educación para la enseñanza de la neumática en las carreras de ingeniería. Ofrece una experiencia inmersiva visual y auditiva que permite a los usuarios trabajar con réplicas 3D de los equipos neumáticos disponibles físicamente en el laboratorio. Permite llevar a cabo interacciones entre profesores y alumnos gracias a la funcionalidad multiusuario. Todas estas características resultaron muy valiosas durante la pandemia pues los estudiantes pudieron tener una experiencia cercana a la realidad [16].

2.3. El motor de videojuegos Unity como herramienta para la creación de ambientes virtuales de simulación

Como se ha mencionado, el ambiente virtual de simulación NERV fue construido mediante el motor de videojuegos UNITY®. Para la implementación de gemelos digitales también fue utilizada esta herramienta por lo que se hace una pequeña semblanza de los orígenes de UNITY® y una breve revisión de los principales elementos que fueron utilizados en la presente implementación.

Unity® es un motor de videojuegos y un ambiente integrado de desarrollo (IDE, por sus siglas en inglés) para la creación de medios interactivos, típicamente videojuegos. La primera versión de Unity® fue creada por David Helgason, Joachim Ante y Nicholas Francis en Dinamarca y su principal objetivo fue crear un motor de videojuegos asequible con herramientas profesionales para los desarrolladores de videojuegos aficionados [17].

Una de las características que hacen de Unity® un referente en la industria de videojuegos es que permite la importación de muchos formatos 3D como 3ds, Cinema4D, Blender, FBX, y también importar recursos de tipo gráfico, visual y de audio, todo lo cual puede ser optimizado posteriormente en la misma plataforma. Unity® permite construir juegos o ambientes virtuales de simulación mediante su editor y un lenguaje de programación, que permite al usuario mediante scripts crear interacción. El usuario puede escoger entre Java Script o C# como lenguajes de programación. Los juegos en Unity® se crean mediante escenas que representarán un nuevo nivel o un lugar distinto dentro del juego. Las escenas se crean con el editor de terrenos de Unity o importando modelos propios [18].

La figura 2.14 muestra la interfaz de usuario que aparece por defecto en Unity® [18]. Es importante tener una idea general del ambiente de trabajo debido a que se hará referencia a las diversas ventanas en la descripción de la presente implementación. A continuación, se describen las ventanas de la interfaz de usuario a partir de la numeración mostrada en la figura.

- 1) Es la ventana escena (Scene); en donde se construyen, mediante objetos, las escenas del ambiente virtual. Dentro de esta ventana hay dos pestañas que en la figura se señalan por flechas; las cuales corresponden a dos ventanas anidadas a este espacio. Las ventanas se pueden anidar en distintos espacios.
- 2) Es la ventana Jerarquía (Hierarchy), esta ventana contiene todos los elementos de la escena actual y también organiza los objetos por escenas.
- 3) La ventana Proyecto (Project), es una ventana que se divide a su vez en dos partes. La parte de la izquierda contiene una estructura jerárquica de carpetas, en donde se pueden crear nuevas carpetas y se pueden organizar todos los archivos que se necesitan para el proyecto; como las escenas, modelos, materiales, texturas, scripts, audios, etc. En la parte derecha se muestra el contenido de las carpetas.
- 4) La ventana Inspector (Inspector) es la ventana encargada de albergar todos los parámetros de los objetos que se seleccionan. También permite añadir nuevos parámetros y configurar algunos otros mediante scripts.

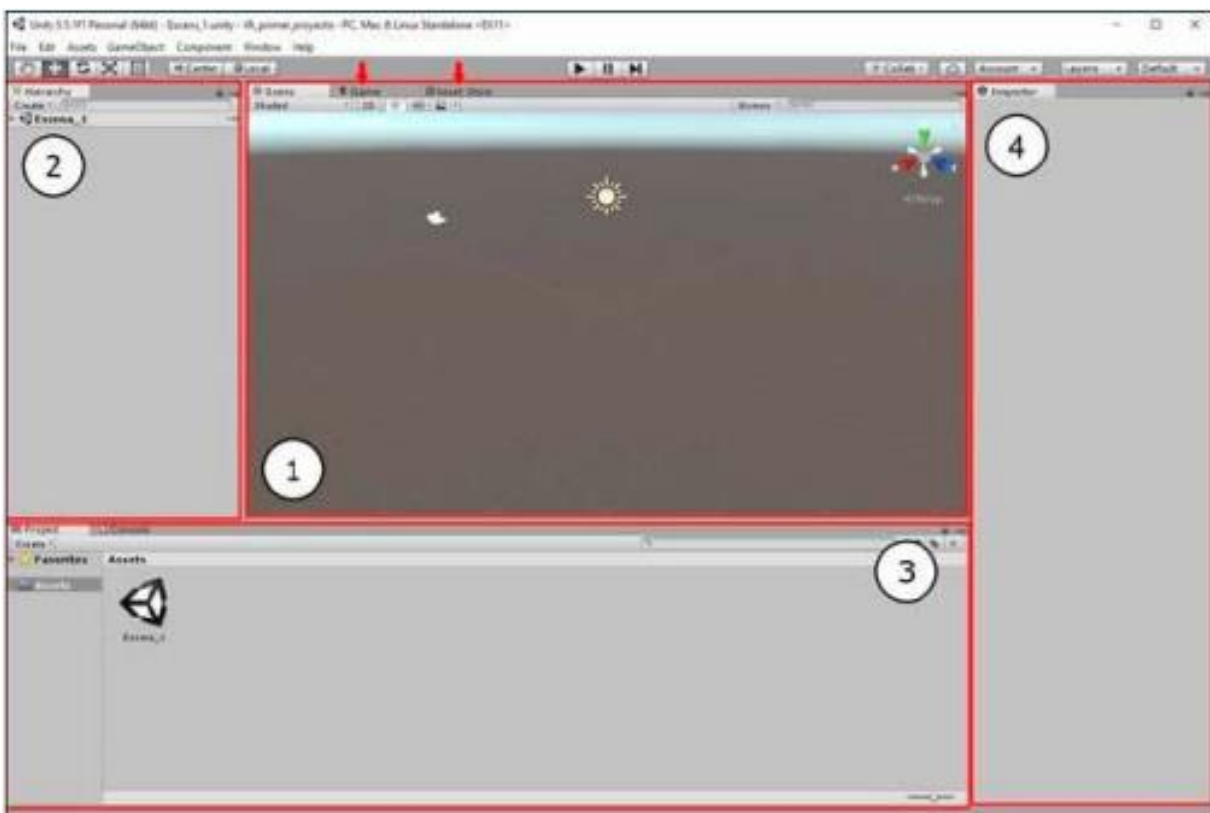


Figura 2.14. Interfaz de usuario por defecto en Unity®. Tomado de [18].

Assets

Para trabajar con los modelos 3D, y crear un ambiente virtual de simulación en Unity, primero deben ser creados en algún software de modelado 3D. En la actualidad hay una gran variedad de software de diseño asistido por computadora (CAD) que facilita la creación de modelos para una gran variedad de aplicaciones. Una vez realizados los modelos 3D pueden ser importados a Unity como Assets.

Los Assets son una representación de cualquier item que puede ser utilizado en el juego o proyecto de simulación. Un asset podría venir de un archivo creado fuera de unity, tal como un modelo 3D, un archivo de audio, una imagen, o cualquiera de los otros tipos de archivos que Unity soporta (Figura 2.15). También hay otro tipos de assets que pueden ser creados dentro de Unity, tal como un Animator Controller, un Audio Mixer o una Render Texture [20].

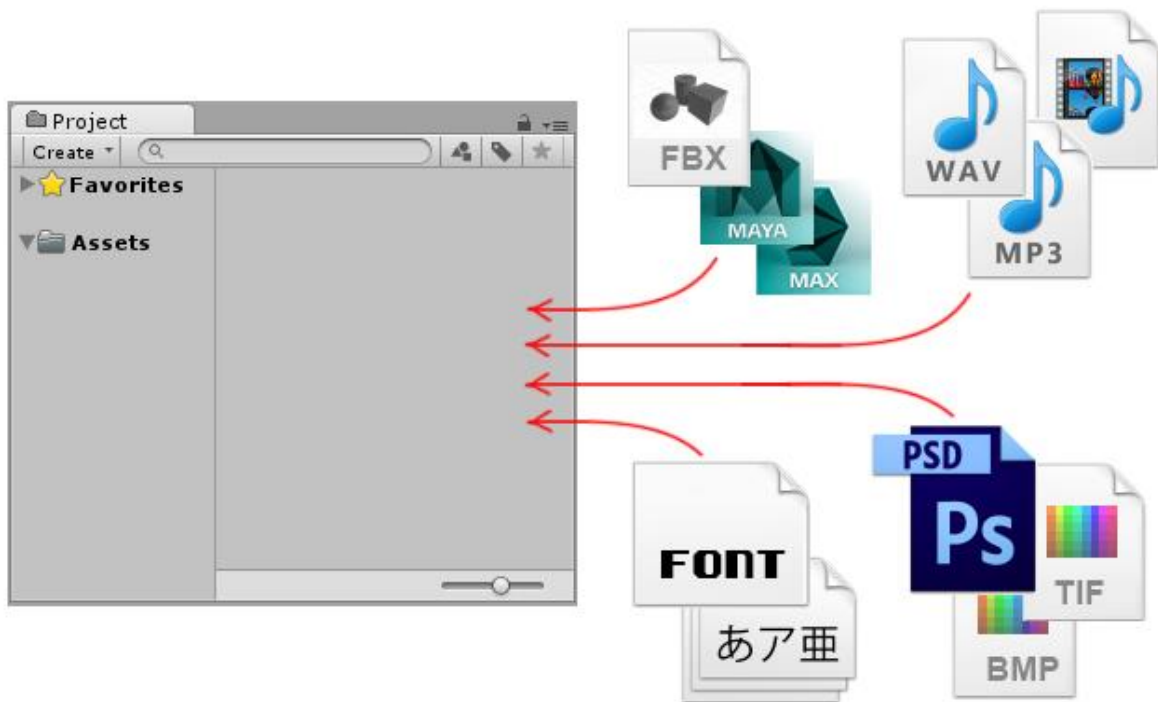


Figura 2.15. Representación de los Assets en Unity®. Tomado de [16].

Un Asset puede ser importado a Unity con solo arrastrar el archivo requerido hasta la carpeta assets desde la ventana que lo contiene. Los modelos 3D pueden ser visualizados entonces en la ventana escena arrastrando los modelos desde la carpeta Assets, en la ventana de proyecto, a la ventana escena. Todo lo que contiene una escena son GameObjects [18]. Los GameObjects son objetos fundamentales en Unity que representan

personajes, accesorios y escenarios. Estos no logran nada por sí mismos, pero funcionan como contenedores de los componentes (Components) que implementan la verdadera funcionalidad [20]. Por ejemplo, si agregamos un componente Rigidbody (Cuerpo Rígido) a un GameObject este se verá afectado por la física que se haya implementado en el ambiente virtual de simulación. Los componentes de un GameObject pueden ser visualizados, agregados o borrados a través de la ventana Inspector (ver Figura 2.14). La figura 2.16 muestra la ventana Inspector de un GameObject llamado Cube donde se pueden observar los siguientes componentes: el componente Transform, el cual determina la posición, orientación y escala del GameObject; el componente Mesh Filter y el componente Mesh Renderer, los cuales dibujan la superficie del Cubo; y el componente Box Collider el cual representa, en términos físicos, un volumen sólido.

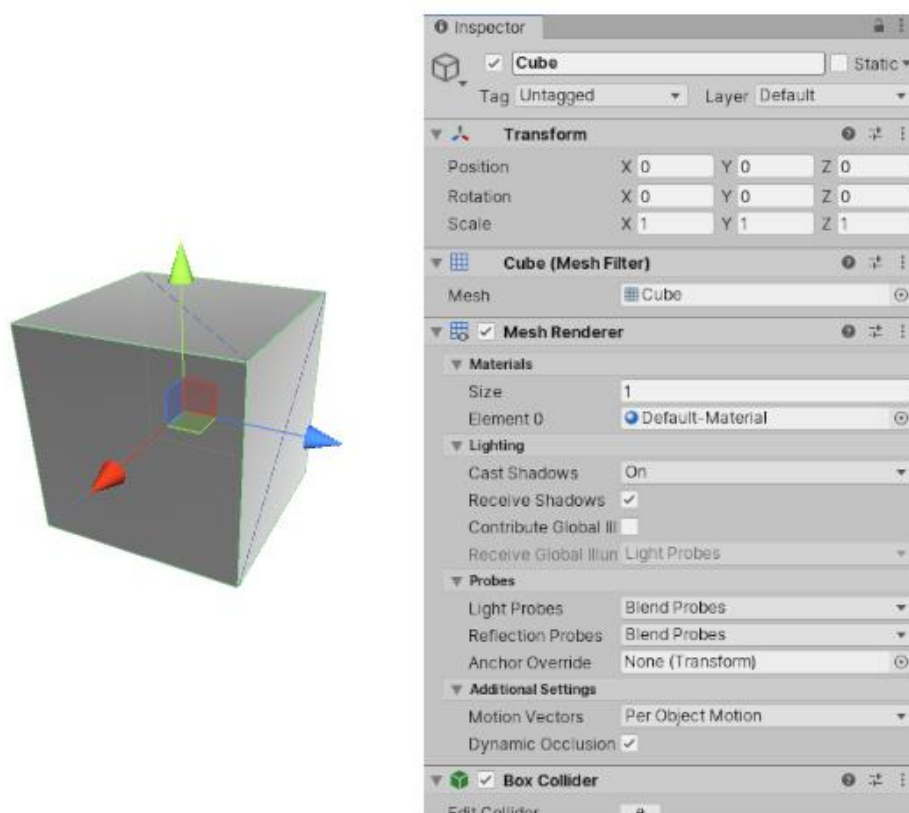


Figura 2.16. Ventana Inspector del GameObject Cube, donde se observan los componentes: Transform, Mesh Filter, Mesh Renderer y Box Collider. Tomado de [21].

Un GameObject siempre tiene atado el componente *Transform* y no es posible quitarlo. Los otros componentes que le dan al objeto su funcionalidad pueden ser agregados del menú *Component* del editor o mediante scripts. También hay muchos objetos útiles preconstruidos (cubos, esferas, cilindros, cámaras, etc.) disponibles en Unity [21]. A continuación, se describen algunos componentes importantes para la implementación de gemelos digitales.

El Componente Transform

Como ya se mencionó el componente *Transform* siempre va asociado a un *GameObject* y es imposible eliminarlo. Este componente define la posición, orientación y escala del objeto en el mundo del ambiente virtual (Figura 2.17). Los valores de posición rotación y escala del componente *Transform* se miden con relación al padre de la transformación. Si el *Transform* no tiene padre las propiedades son medidas en el espacio del mundo [22].

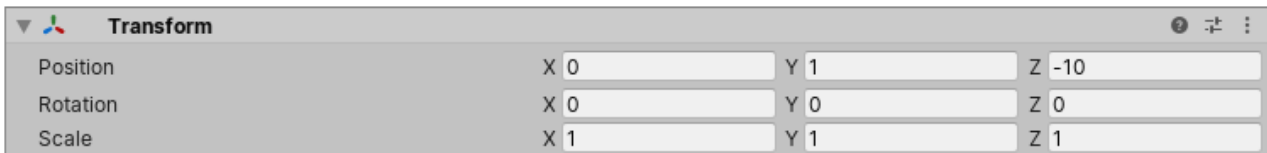


Figura 2.17. Componente Transform donde se observan las propiedades de posición, rotación y escala. Tomado de [22].

El componente Rigidbody

Es el componente principal que permite el comportamiento físico para un objeto. Con un *Rigidbody* adjunto, el objeto inmediatamente responderá a la gravedad. Si uno o más componentes *Collider* son también agregados, entonces el objeto será movido por colisiones entrantes [23]. La figura 2.18 muestra el componente *Rigidbody* como es visto desde la ventana del Inspector, sus propiedades son:

- Masa. Es la masa del objeto, por defecto en kilogramos.
- Drag. Es la resistencia al aire cuando se mueve debido a fuerzas, 0 significa que no hay resistencia e infinito hace que se detenga inmediatamente.
- Angular Drag. La resistencia del aire cuando rota por la aplicación de una torca, 0 significa que no hay resistencia al aire, pero infinito no provocará que el objeto deje de rotar.
- Use Gravity. Si está activada el objeto se verá afectado por la gravedad
- Is Kinematic. Si está activado el objeto no será controlado por el motor de físicas y solo podrá ser manipulado por su Transform.
- Interpolate. Sirve para regular las sacudidas (Jerk) que pudieran estar afectando al objeto.
 1. None. No se aplica ninguna interpolación.
 2. Interpolate. La transformación se suaviza en función de la transformación del fotograma anterior
 3. Extrapolar. La transformación se suaviza en función de la transformación estimada del siguiente fotograma.
- Collision Detection. Se usa para evitar que los objetos que se mueven rápidamente pasen a través de otros objetos sin detectar colisiones.

1. Discrete. Es el valor predeterminado. Se usa contra todos los otros Colliders en la escena. Otros colisionadores utilizarán esta detección de colisión cuando prueben la colisión contra él.
 2. Continuous. Se usa detección Discreta contra Colliders dinámicos (con Rigidbody) y detección continua contra Colliders estáticos (sin Rigidbody).
 3. Continuous Dynamic. Utilizado para objetos que se mueven rápidamente
 4. Continuous Speculative. Se utiliza cuando el GameObject se establece como is kinematic.
- Constrains. Restringe selectivamente la movilidad lineal y/o angular del GameObject respecto a los ejes coordenados del mundo.

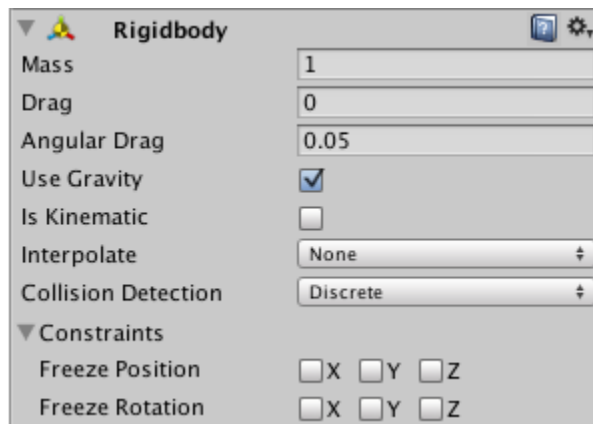


Figura 2.18. Componente Rigidbody con sus propiedades características. Tomado de [23].

El componente Collider

Los componentes *Collider* (colisionador) definen la forma de un *GameObject* para efectos de colisiones físicas. Un colisionador, que es invisible, no necesita tener exactamente la misma forma que la malla del *GameObject*. Una aproximación de la malla suele ser más eficiente e indistinguible en el juego o simulador virtual. Los colisionadores más simples (y menos intensivos para el procesador) son tipos de colisionadores primitivos, como: *Box Collider*, *Sphere Collider*, *Capsule Collider*. Se puede agregar cualquier cantidad de estos a un solo *GameObject* para crear colisionadores compuestos. Los colisionadores compuestos se aproximan a la forma de un *GameObject* mientras mantienen una sobrecarga de procesador baja. Para obtener mayor flexibilidad se pueden agregar colisionadores adicionales en *GameObjects* secundarios. Cuando se crea un colisionador compuesto, solo se debe usar un componente *Rigidbody* colocado en el *GameObject* raíz en la jerarquía [24]. La figura 2.19 muestra las propiedades del componente *BoxCollider*, el cual es un colisionador (*Collider*) en forma de cubo.

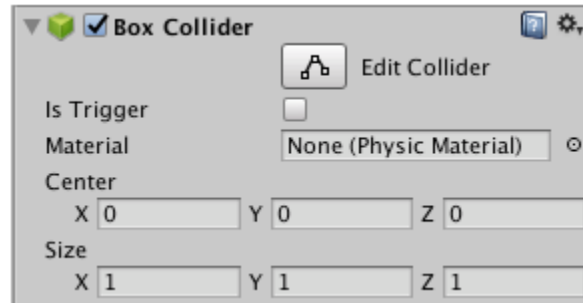


Figura 2.19. Componente *Box Collider*. Tomado de [25].

Para la implementación de gemelos digitales solo fue requerido el tipo de colisionador *Box Collider*. Las propiedades de los colisionadores son similares a las representadas en la figura 2.19 para el *BoxCollider*, las cuales son:

- *Is Trigger*. Si está habilitada, el *Collider* será usado para eventos de *triggering* (Detección), y será ignorado por el motor de física.
- *Material*. Se refiere al *Physics Material* que determina en qué forma el *Collider* interactúa con otros *Colliders*.
- *Center*. La posición del *Collider* en el espacio local del objeto.
- *Size*. El tamaño del *Collider* en las direcciones X, Y, y Z.

Dentro de estas, la propiedad *Material* es de suma importancia y fue ampliamente utilizada en la implementación de gemelos digitales. Como se mencionó anteriormente, la propiedad *Material* hace referencia a *Physics Material* que se describe a continuación.

La Propiedad *Physics Material* del Componente *Collider*

El material físico (*Physics Material*) ajusta la fricción y los efectos de rebote de la colisión [26]. *Physics Material* a su vez cuenta con otras propiedades que pueden ser configuradas (Fig. 2.20):

- Fricción dinámica (*Dynamic Friction*). Es utilizada cuando el cuerpo ya se está moviendo. Por lo general tiene un valor entre 0 a 1. Un valor de cero se siente como hielo, un valor de 1 hará que se detenga muy rápidamente a menos que mucha fuerza o gravedad empujen el objeto.
- Fricción estática (*Static Friction*). La fricción que se usa cuando un objeto está quieto sobre la superficie. Por lo general, un valor de 0 a 1. Un valor de cero se siente como hielo, un valor de 1 hará que sea muy difícil mover el objeto.
- Rebote (*Bounciness*). Qué tan bien rebota la superficie. Un valor de 0 no rebotará. Un valor de 1 rebotará sin ninguna pérdida de energía, aunque se esperan ciertas aproximaciones que podrían agregar pequeñas cantidades de energía a la simulación.
- Combinación de fricción (*Friction Combine*). Cómo se combina la fricción de dos objetos que chocan.

- Promedio. Los dos valores de fricción se promedian
 - Mínimo. Se utiliza el menor de los dos valores.
 - Máximo. Se utiliza el mayor de los dos valores.
- Combinación de rebote (*Bounce Combine*). Cómo se combina el rebote de dos objetos que chocan, Tiene los mismos modos que el modo combinación de fricción.

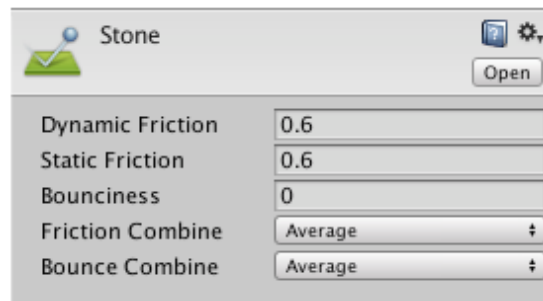


Figura 2.20. Propiedad *Material (Physics Material)* del componente *Collider*. Tomado de [26].

Cuando dos cuerpos están en contacto, los modos de *bounciness* (rebote) y *friction* (fricción) son aplicados individualmente a cada agente. Entonces, cuando el cuerpo A tiene un modo *Average* y un cuerpo B tiene un modo *Multiply*, el cuerpo A se va a comportar de acuerdo a los parámetros promedio (*average*) y B de acuerdo a los parámetros que multiplican (*multiplied*) [26].

El detector *Raycast*

El *Raycast* es un rayo invisible que choca con los colisionadores (*Colliders*) y nos devuelve una información. Para los *Raycast* necesitamos siempre:

- Un punto de origen, y
- Un punto final o una distancia

El rayo es disparado desde el punto origen y este atraviesa la escena hasta encontrar el punto final o recorrer la distancia (fig. 2.21). Durante el trayecto el rayo puede colisionar con otros objetos que lleven colisionadores, cuando sucede esto devuelve información del objeto con el que ha colisionado [18].

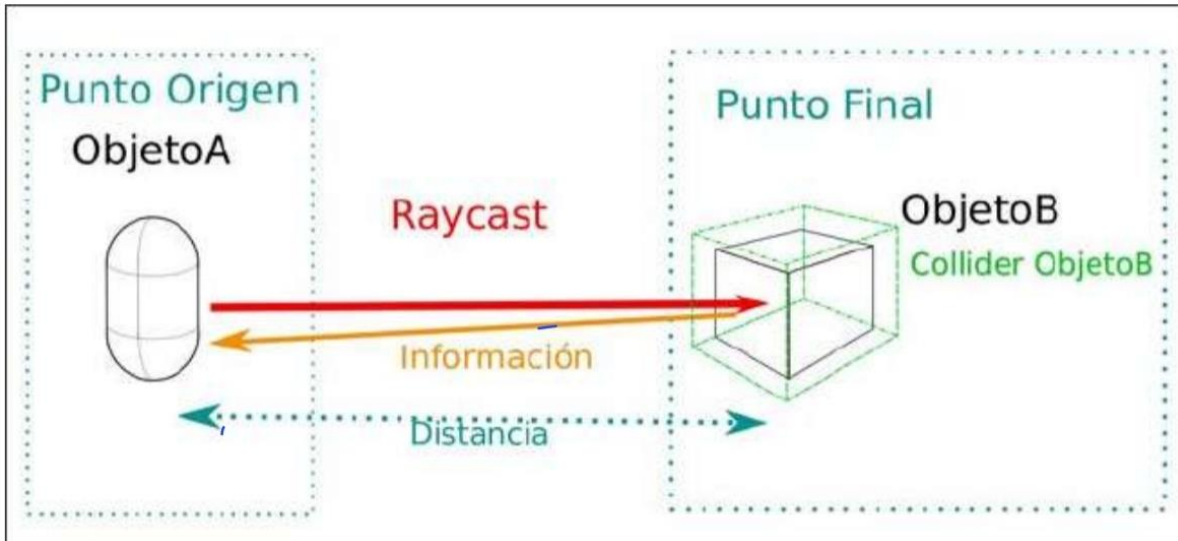


Figura 2.21. Se ilustra el funcionamiento del Raycast. El Raycast parte del origen hasta colisionar con un *Collider* y se devuelve información del objeto con el que colisionó. Tomado de [18].

El Componente Animator

Unity dispone de un sistema de animación bastante sofisticado el cual contiene varias características como [27]:

- El flujo de trabajo configurable de animación para todos los elementos de Unity, incluidos objetos, personajes, y propiedades.
- Clips de animación que pueden realizarse dentro de Unity. También soporta la importación de clips de animación creados en otros programas.
- Se pueden reorientar las animaciones de tipo humanoide, es decir se puede transferir una animación de un modelo de personaje a otro.
- Flujo de trabajo simplificado para alinear clips de animación.
- Conveniente vista previa de clips de animación, transiciones e interacciones entre ellos. Esto permite a los animadores trabajar de manera más independiente de los programadores, crear prototipos y obtener una vista previa de las animaciones antes de que se conecte el código del juego.
- Gestión de interacciones complejas entre animaciones con una herramienta de programación visual.
- Animado de diferentes partes del cuerpo con diferente lógica.
- Funciones de capas y máscaras.

Flujo de trabajo de animación

El sistema de animación de Unity se basa en el concepto de *Clip de animación*, que contiene información sobre cómo ciertos objetos deberían cambiar su posición, rotación u otras propiedades a lo largo del tiempo. Cada clip se puede considerar como una única grabación lineal. Los clips de animación de fuentes externas son creados por artistas o animadores con herramientas de terceros o provienen de estudios de captura de movimiento u otras fuentes.

Luego los clips de animación se organizan en un sistema estructurado similar a un diagrama de flujo llamado *Animator Controller* (Figura 2.22). El controlador de animación actúa como una máquina de estados que realiza un seguimiento de qué clip se debe reproducir actualmente y cuándo las animaciones deben cambiar o combinarse.

Un *Animator Controller* muy simple puede contener solo uno o dos clips, por ejemplo, para animar una puerta que se abre y se cierra en el momento correcto. Un controlador de animación más avanzado podría contener docenas de animaciones humanoides para todas las acciones del personaje principal, y podría combinar varios clips al mismo tiempo para proporcionar un movimiento fluido a medida que el jugador se mueve por el escenario [27].

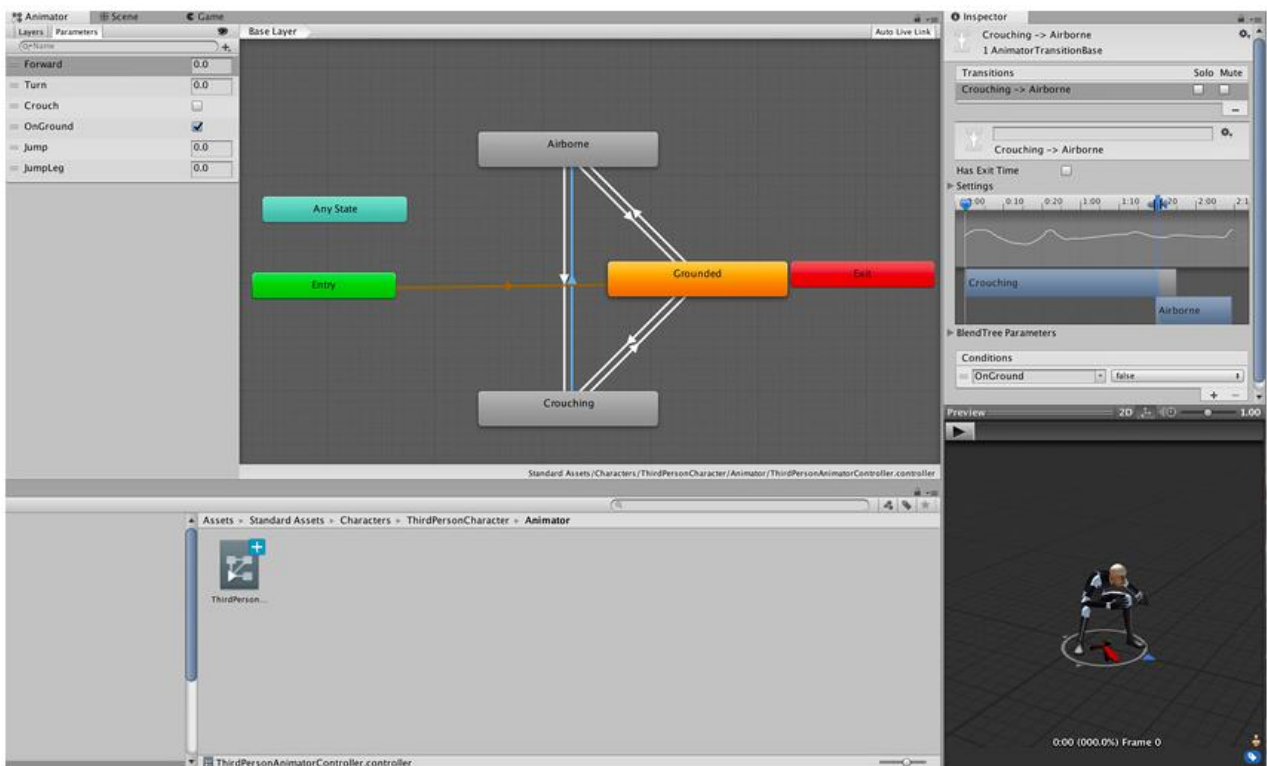


Figura. 2.22. Vista típica de una máquina de estado de animación en la ventana de Animator.
Tomado de [27].

Agregado de texturas PBR en Unity

Unity permite agregar materiales y texturas directamente desde la ventana de proyecto. Para modelos simples, agregar materiales y texturas directamente desde Unity puede ahorrar tiempo y esfuerzo. En modelos más complejos es preferible el uso de otro tipo de herramientas para asignar materiales y texturas. En la implementación de gemelos digitales la mayor parte de los modelos son figuras geométricas simples y la asignación de materiales y texturas se puede hacer directamente desde Unity. Se agrega un nuevo material desde la ventana de Proyecto (Project) y nos aparece una nueva ventana de material. En la nueva ventana podemos asignar colores y variar los parámetros de rugosidad y metálico de los materiales (ver siguiente sección), así como ingresar imágenes PBR para su uso como texturas. Las texturas PBR dan información sobre el nivel de detalle (mapa de normales), el color del material (Base Color/Albedo/Difuse), el desplazamiento de los polígonos (Height), la cantidad de reflexión (Specular), el detalle de la superficie (Roughness) y otro tipo de información como transparencia, refracción, curvatura, posición de los polígonos, etc. Según el tipo de material y su aplicación, necesitaremos más mapas o menos [29]. Las texturas PBR están disponibles en forma gratuita y de pago en internet. Hay una gran variedad de texturas para elegir: desde madera hasta metales y cerámicos. En la figura 2.23. se muestra un ejemplo de este tipo de texturas donde podemos observar las texturas de Color, Normal Maps, Height, y Rugosidad. Para agregar las imágenes PBR a algún determinado material primero tienen que ser importadas a Unity (como assets) y luego pueden ser arrastradas a cada uno de los parámetros del material (albedo, metallic, Normal map, etc.) como se muestra en la figura 2.24, donde se han colocado cada una de las texturas PBR a los respectivos parámetros del material, dando como resultado la textura de ladrillos que se puede observar en la parte inferior de la figura. Además de este tipo de texturas también fue requerido crear texturas propias, con ayuda del software de Diseño Blender®.

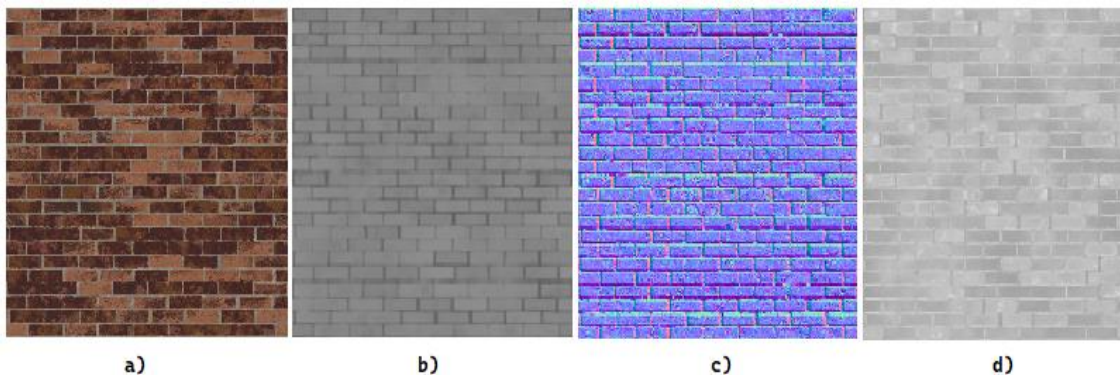


Figura 2.23. Imágenes PBR donde podemos observar las distintas imágenes que dan como resultado una textura, en este caso de ladrillos. a) Imagen de Color, b) Imagen de Height, c) Imagen de Normal Maps, d) Imagen de Rugosidad. Las imágenes PBR se han descargado de la referencia [30].

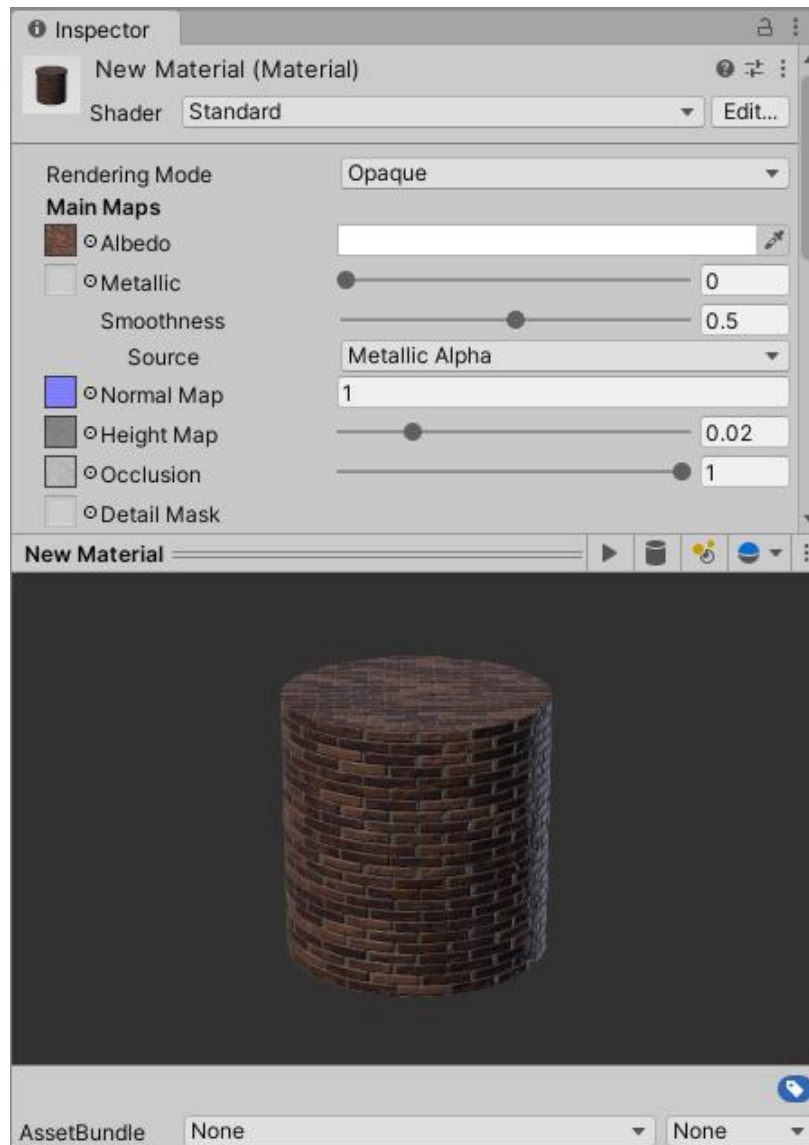


Figura 2.24. Se agregaron las respectivas texturas PBR mostradas en la figura 2.23 a los parámetros albedo, Normal map, Height map y oclusión. El resultado es la textura de ladrillos mostrada en la parte inferior de la figura.

También es posible crear distintos tipos de materiales mediante la combinación de las propiedades del material, esto se verá en mayor profundidad en la siguiente sección, pero también se crearon algunos materiales de este modo en Unity. En particular, para la implementación de los gemelos digitales solo se requirieron variar los parámetros metálico y rugosidad de los materiales, pero hubo un caso especial donde se utilizó una textura translúcida. Esta textura se creó desde Unity® pues al hacerla translúcida se perdía el efecto de cualquier otro parámetro, por lo que solo fue requerido poner un color determinado de acuerdo con la paleta de colores elegida para los modelos. Para hacer un material translucido

se requiere crear un nuevo material y en el parámetro Rendering Mode (figura 2.25) elegimos transparente (Transparent). En la propiedad color elegimos un valor de A (Alpha) apropiado, según se adecúe a lo requerido. Un valor más alto de A dará por resultado un material más opaco y uno bajo dará un material más traslúcido [28].



Figura 2.25. Creación de un material traslúcido.

2.4. El software de modelado Blender como herramienta para agregar texturas a los modelos 3D

Blender® es una suite de creación 3D gratuita y de código abierto totalmente integrada que ofrece una amplia gama de herramientas esenciales, que incluyen modelado, renderizado, animación y rigging, edición de video, VFX, composición, texturizado y muchos tipos de simulaciones [32]. A pesar de sus grandes capacidades, para la implementación de gemelos digitales, solo fueron requeridas las herramientas de material y texturizado debido a que los modelos 3D de las piezas y los ensambles de los gemelos digitales fueron desarrollados con la paquetería Inventor®. La interfaz de usuario de Blender es bastante personalizable; por defecto está dividida en 6 distintas partes como se muestra en la figura 2.26.

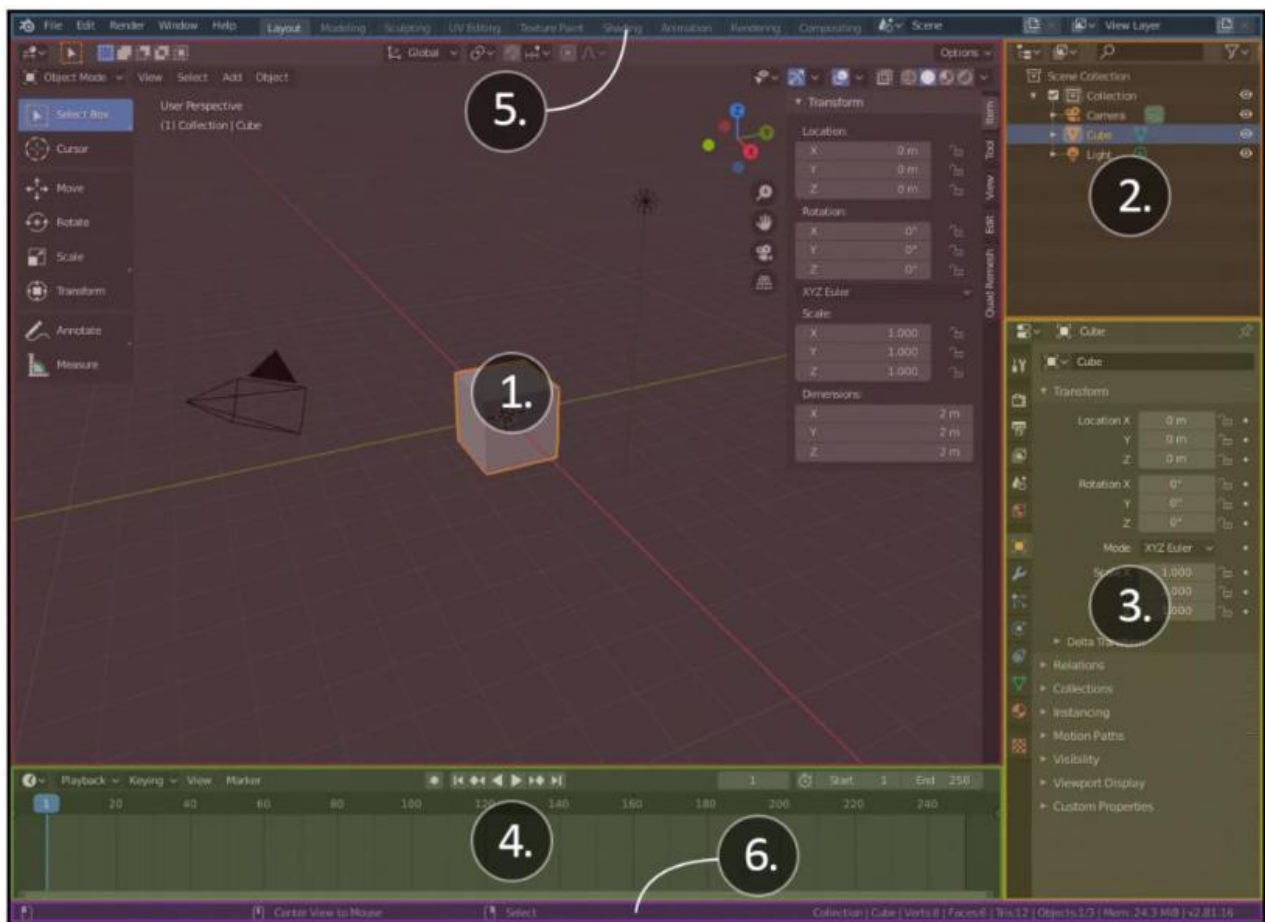


Figura. 2.26. Interfaz de usuario de Blender la cual por defecto está dividida en seis distintas partes. Tomado de [31].

Las cuatro áreas en el centro de la interfaz de usuario son llamadas editores. Cada editor nos presenta una forma distinta de ver los proyectos 3D. Hay muchos tipos de editores, pero por defecto se presentan los siguientes en el espacio de trabajo (ver fig.2.26):

1. Ventana de visualización 3D (*3D Viewport*). Es donde se trabaja la mayor parte del tiempo. Es la ventana en las escenas 3D. Casi todo el modelado 3D es hecho en esta ventana.
2. Listado (*Outliner*). En este editor se listan todos los objetos en el proyecto y ayuda a organizar las escenas.
3. Propiedades (*Properties*). El panel de propiedades contiene los ajustes de renderizado y nos permite agregar modificadores avanzados, restricciones, partículas, física y materiales a los modelos 3D.
4. Línea de Tiempo (*Timeline*). La línea de tiempo es útil cuando empezamos a animar. Realiza un seguimiento de las opciones de reproducción y los fotogramas clave.
5. Barra superior (*Top bar*). La barra superior incluye el típico menú de opciones, como: archivo, editar, etc. Además, presenta pestañas para acceder a los diversos editores disponibles en Blender como: modelar (*modeling*), esculpir (*sculpting*), editor UV (*UV editing*), etc.
6. Barra de estatus (*Status bar*). Está situada en la parte inferior de la interfaz de usuario. Incluye útiles recordatorios de atajos de teclado, operaciones de herramientas, contador de polígonos, y otra información sobre el archivo actual.

Materiales y Texturas

Podemos agregar color a los modelos 3D con una mezcla de materiales y texturas. Los materiales son usados para determinar cómo se comporta la luz cuando interactúa con la superficie de un objeto (vidrio, metal, cerámica). Las texturas son imágenes 2D que se envuelven en un modelo 3D algo así como una envoltura de dulce. Para hacer que las texturas se ajusten al modelo primero debemos desenvolver el modelo 3D, lo que da como resultado una representación 2D del modelo llamado mapa UV. La figura 2.27 muestra el resultado de desenvolver un modelo 3D y el mapa UV resultante [31].

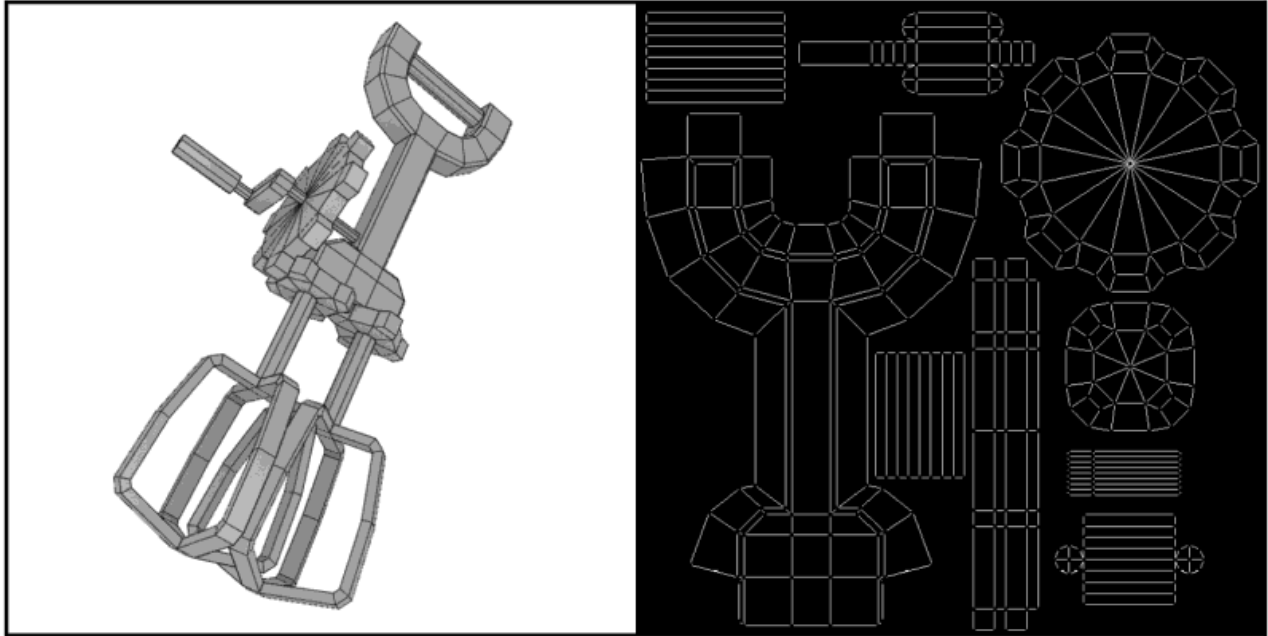


Figura 2.27. Proceso de desdoblamiento de un modelo 3D (izquierda) para obtener un mapa UV (derecha). Tomado de [31].

Al proceso de agregar materiales y texturas a los modelos 3D mediante su transformación a modelos 2D, se le conoce como renderizado (Rendering) [33]. Blender cuenta con varios motores de renderizado como Eevee y Cycles. En general Cycles utiliza más recursos computacionales que Eevee ya que Cycles es un motor de renderizado basado en el trazado de rayos de luz. Por su parte Eevee, en vez de calcular cada rayo, utiliza un proceso nombrado rasterización (*rasterization*). La rasterización estima la forma en que la luz interactúa con objetos y materiales utilizando numerosos algoritmos. Dependiendo de la aplicación se puede optar por Eevee o Cycles como motores de renderizado teniendo en cuenta que Cycles dará renderizaciones físicamente más realistas (fotorrealistas) con un mayor gasto de recursos computacionales, mientras que Eevee será más eficiente, pero los resultados pueden no ser los esperados [34]. Para el desarrollo de las texturas de los gemelos digitales usamos el motor de renderizado Eevee.

El material BSDF principista

Para todos los colores y texturas usados en la implementación de gemelos digitales y creados desde Blender® se utilizó el material BSDF principista. BSDF son las siglas en inglés de Bidirectional Scatter Distribution Falloff (Caída de distribución de dispersión bidireccional). Cuando una emisión de luz choca con una superficie, hay un número limitado de efectos físicos plausibles. La emisión del rayo de luz puede reflejarse (*Reflection*) en la

superficie, rebotando la energía lumínica. Puede dispersarse (*Scatter*) sobre la superficie, parcialmente siendo absorbida y parcialmente rebotada como rayos difusos (también llamado albedo). Cuando la luz puede dispersarse más profundamente y salir de una superficie, se denomina dispersión del subsuelo. Pero es posible que el rayo nunca se refleje, teniendo transmisión a través de la superficie, pasando directamente y continuando por el otro lado. Esta transmisión a través del sombreador puede ralentizar el rayo de luz, cambiando su ángulo, lo que se conoce como refracción (ver figura 2.28) (Baechler & Greer, 2020).

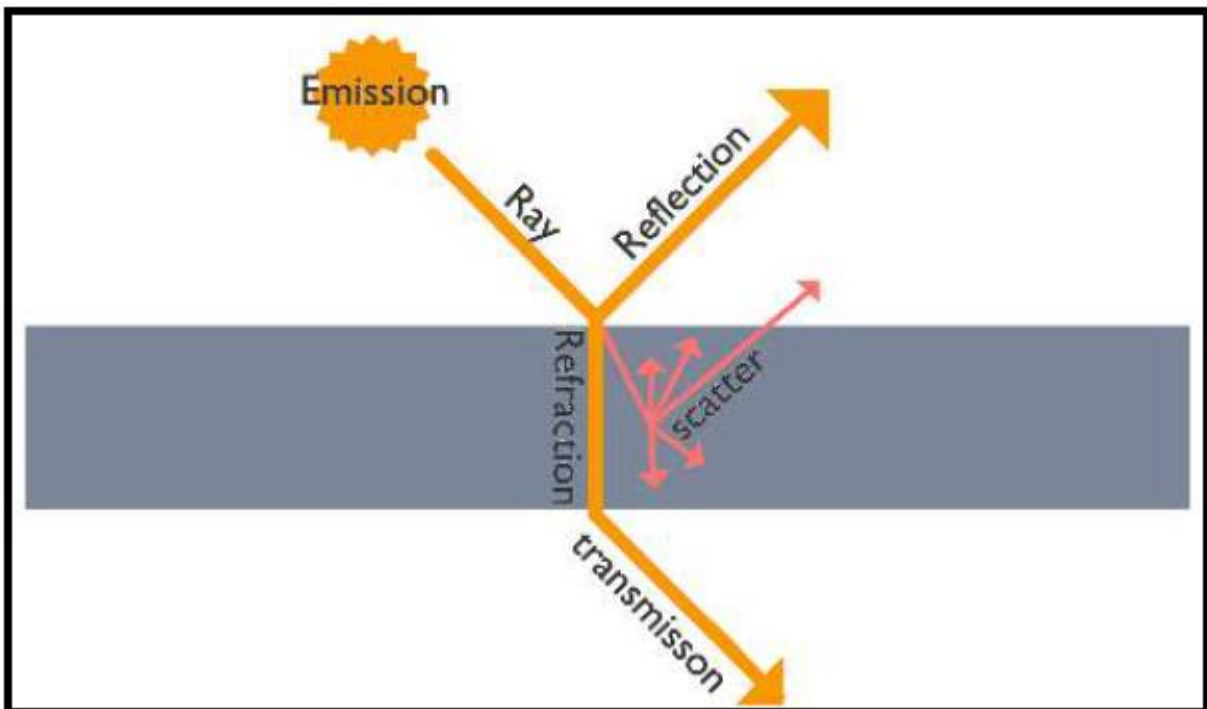


Figura 2.28. Posibles efectos físicos de la incidencia de un rayo de luz sobre una superficie. Tomado de [31].

La rugosidad (*Roughness*) complica el modelo mostrado en la figura 2.28. En un espejo perfecto, el rayo rebota directamente fuera de la superficie en una dirección, pero la mayoría de las superficies tienen cavidades, grietas o detalles microscópicos (Rugosidad) que cambian el ángulo de reflexión como se muestra en la figura 2.29. Las reflexiones pueden ser desglosadas en dos tipos, Difuso (*Diffuse*) y Especular (*Specular*), las cuales tienen influencia en dos categorías de materiales: metales y no metales. La reflexión especular rebota directamente sobre la superficie de incidencia. Aunque la rugosidad puede dispersar los rayos, rebotan en un ángulo de incidencia predecible. Por su parte los rayos difusos se dispersarán dentro y fuera de la superficie y limitarán su color a la energía que no fue absorbida en esa dispersión (ver figura 2.30).

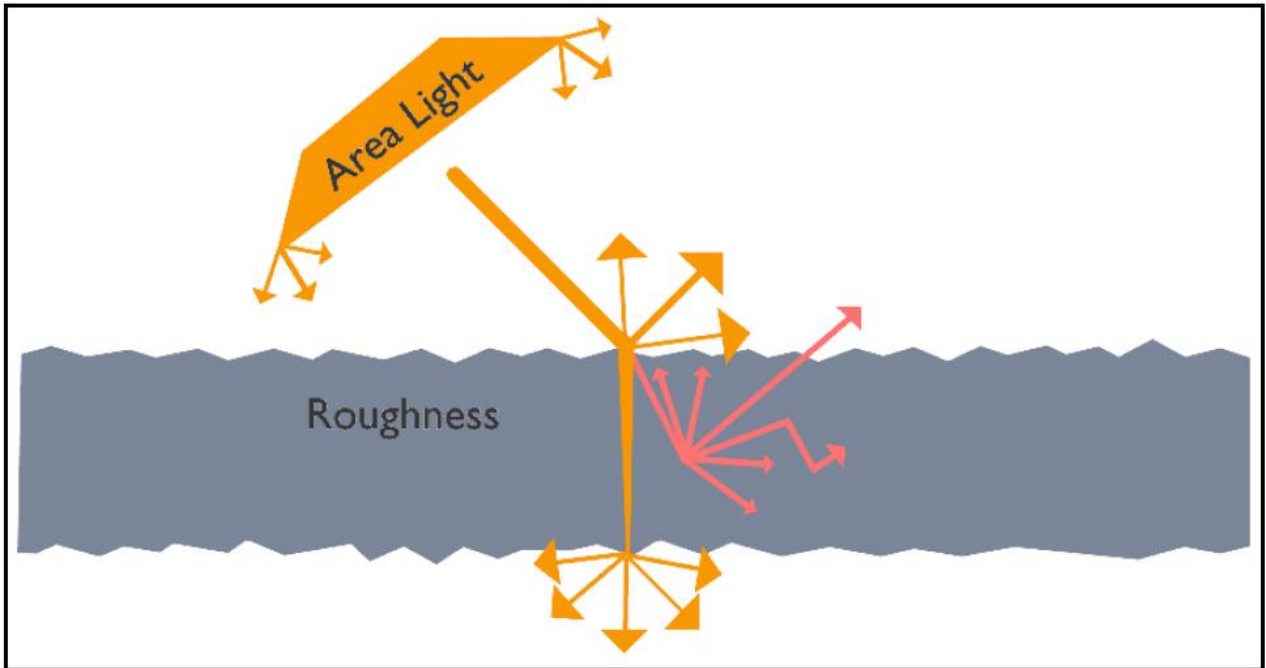


Figura 2.29. Efecto de la rugosidad en la incidencia de un rayo de luz sobre la superficie. Tomado de [31].

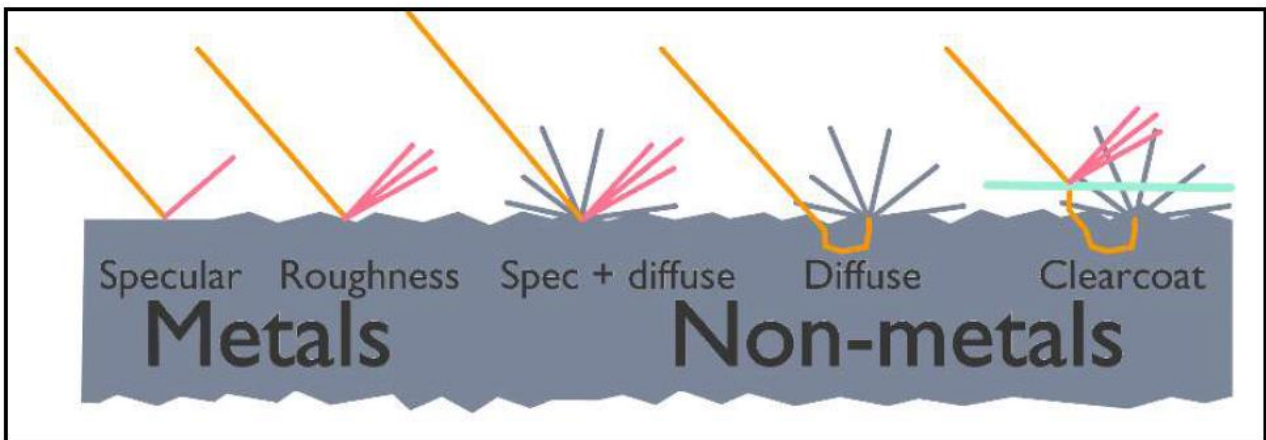


Figura 2.30. Efecto de la incidencia de un rayo de luz sobre las superficies especulares y difusivas. Tomado de [31].

Un material en Blender debería ser explícitamente un metal o un no metal. Hay excepciones, tales como superconductores o el pixel suavizado entre metales y no metales en un mapa de textura. Los metales deberán tener su propiedad Metálica (Metallic) establecida en 1. La propiedad de color base (Base Color) del sombreador (shader) puede combinarse con la propiedad metálica para darle color al metal [31].

Los no metales, también llamados dieléctricos, muestran su reflexión difusa vía el color base. Ellos pueden tener también reflexión especular al mismo tiempo, y la mayoría de los materiales serán una combinación de los dos [31].

Los materiales transmisivos (Transmissive) permiten el paso de luz a través de ellos. La propiedad de transmisión (Transmission) debería ser establecida entre 0 y 1. [31].

Mezclando cada una de las propiedades anteriores en el material BSDF principista, podemos crear todo tipo de combinaciones, como las mostradas en la figura 2.31.

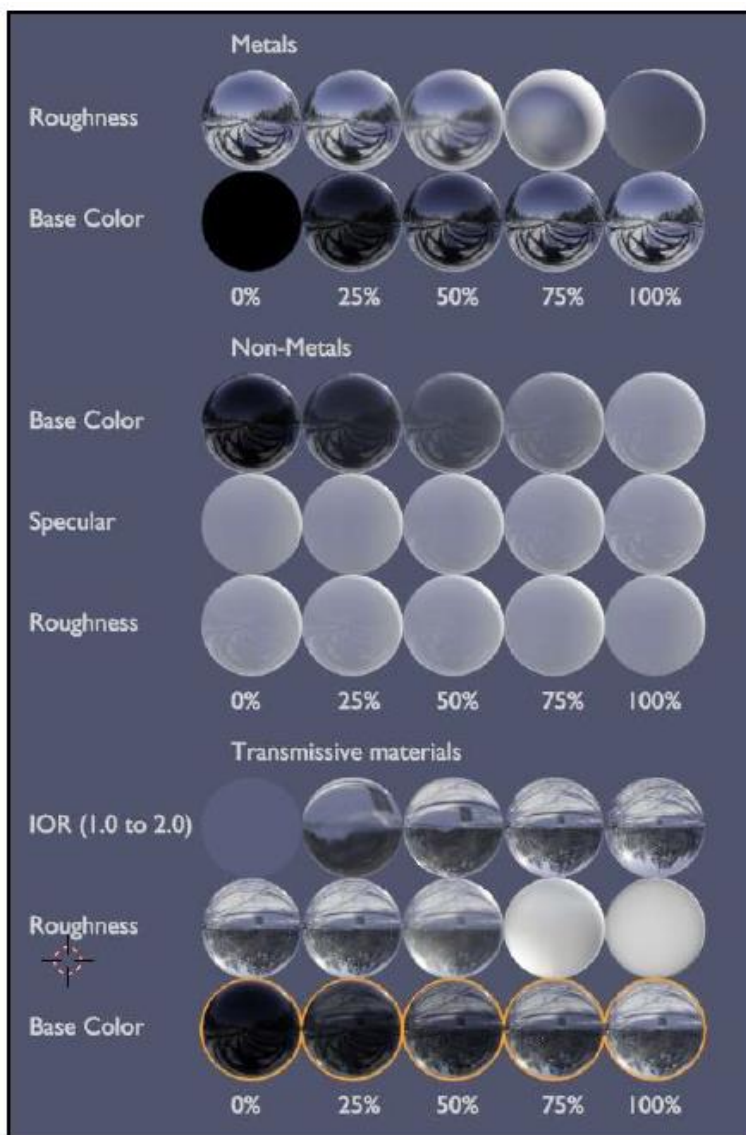


Figura 2.31. Se pueden obtener una gran variedad de materiales mediante la combinación de las diversas propiedades del material BSDF principista. Tomado de [31].

Para crear un nuevo material accedemos al panel de propiedades y presionamos la pestaña de materiales mostrada en la figura 2.32. Dentro del panel aparecerá una nueva sección (ver figura 2.33). En esta ventana se muestra por defecto el material BSDF principista y las propiedades que podemos modificar para conseguir una gran variedad de materiales. Para la implementación de gemelos digitales solo se requirió modificar las propiedades: color base, metálico y rugosidad ya que se consideró que los gemelos digitales serían metálicos.

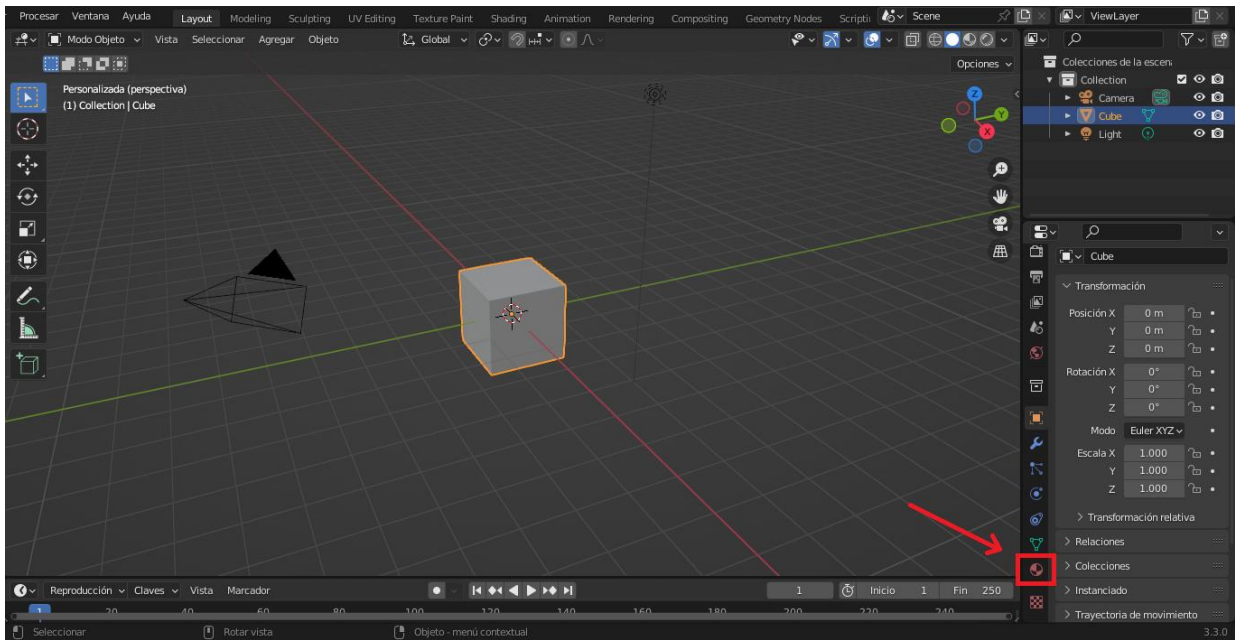


Figura 2.32. Se crea un nuevo material desde la pestaña de materiales (recuadro rojo) del panel de materiales.

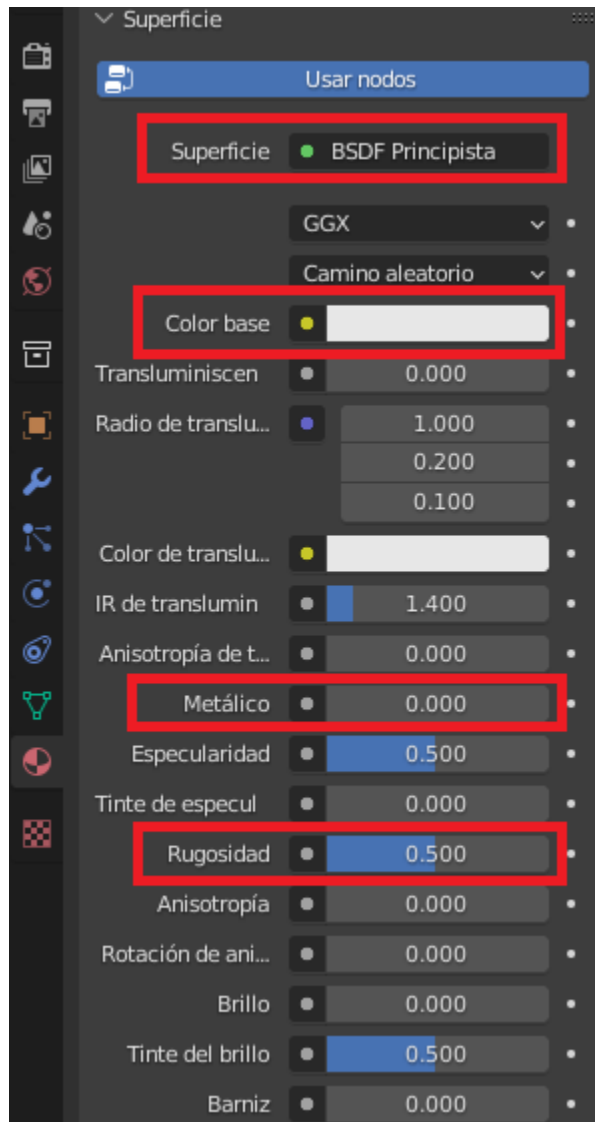


Figura. 2.33. Ventana de propiedades del material. En los recuadros rojos se muestra el material BSDF principista, y las propiedades de color base, metálico y rugosidad.

Para agregar texturas con base en una imagen simplemente agregamos la imagen desde la propiedad Color base. Se nos aparecerá una ventana con opciones en donde elegimos en la sección textura la opción imagen (Fig. 2.34). Buscamos la imagen en nuestro ordenador y con esto podremos añadirla a nuestro modelo a partir de su manipulación por medio de mapas UV.

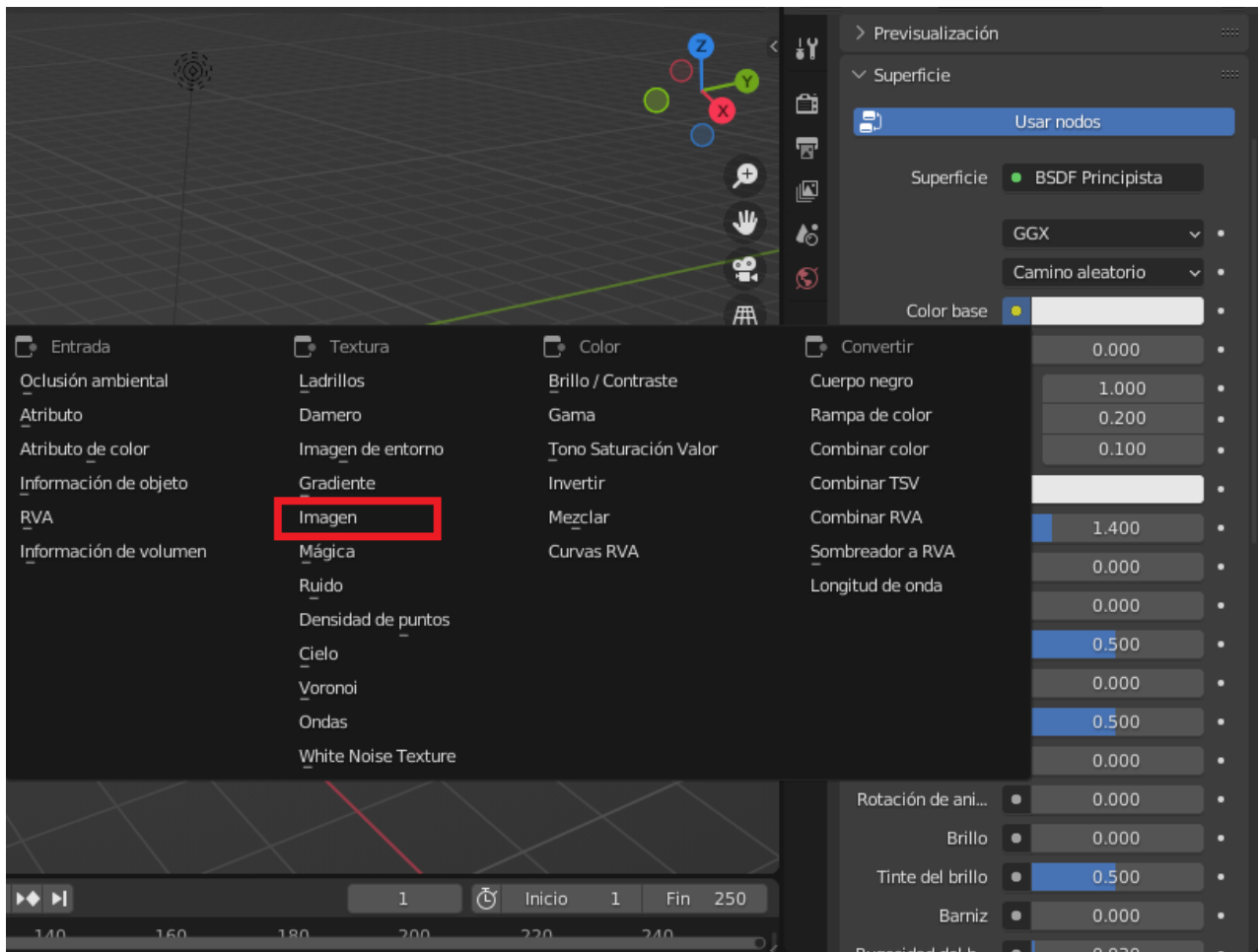


Figura 2.34. Para agregar una textura con base en una imagen seleccionamos la textura imagen desde la propiedad Color base.

Para agregar una imagen como textura a nuestros modelos 3D se debe crear un mapa UV de nuestro modelo. Para ello accedemos al editor UV Editing (Editor UV), desde la barra superior como se muestra en la figura 2.35. Se nos aparecerá un nuevo editor como el mostrado en la figura 2.36, donde podemos observar cómo se ha desdoblado el cubo en el plano UV. Luego podemos manipular el tamaño de la imagen y/o el tamaño de la proyección UV para acomodar la imagen según nuestra preferencia. Es posible agregar las proyecciones UV de las caras del cubo y de las caras de cualquier modelo una por una, para tener un control adecuado de la posición de la imagen que se quiere añadir como textura, en el modelo 3D. Para ello solo es requerido seleccionar las caras que se quieren proyectar en el mapa UV una por una, en vez de seleccionar todo el modelo 3D.

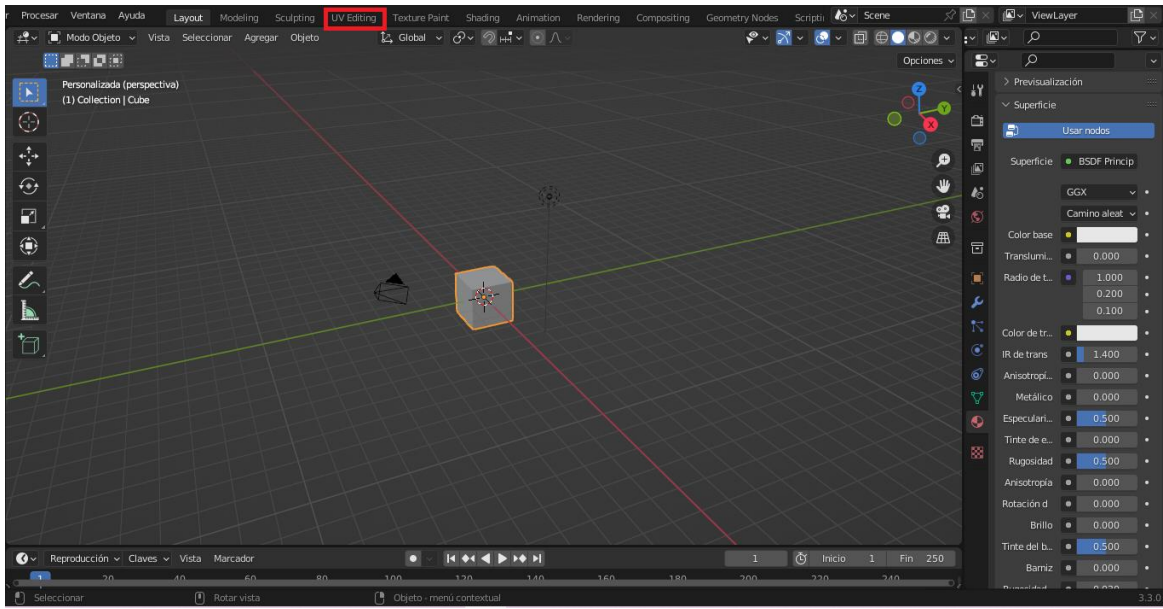


Figura 2.35. Accedemos al UV editor desde la barra superior de la interfaz de usuario de Blender.

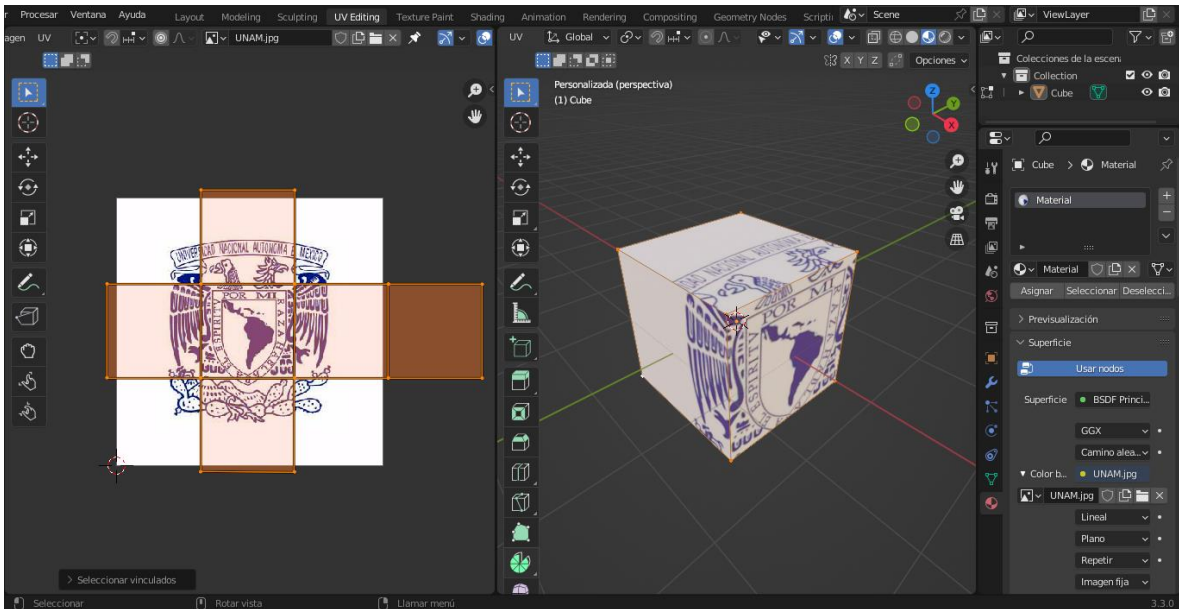


Figura 2.36. Editor UV donde podemos observar cómo se desdobra el cubo mostrado del lado derecho en el plano UV del lado izquierdo. Se ha cargado una imagen de textura con el escudo de la UNAM.

Agregado de Texturas PBR desde Blender

Se ha mencionado que se obtienen los mismos resultados colocando texturas PBR ya sea desde Unity® o desde Blender®. Esto es cierto para modelos no complejos, pero para los modelos que sí lo son, es preferible utilizar Blender para acomodar bien las texturas en sus modelos. Para agregar texturas PBR desde Blender se crea igualmente un material como anteriormente se hizo. Las texturas pueden irse agregando una por una en las distintas propiedades del material; por ejemplo, la textura “*color*” se puede agregar como una imagen a la propiedad color, la textura *metallic* se puede agregar como una imagen a la propiedad metálico, *roughness* a la propiedad rugosidad, etc. Puede que las texturas no queden bien colocadas sobre el modelo al solo agregar texturas de este modo. Para arreglarlo se recurre de nuevo a los mapas UV. En el editor de mapas UV se abre alguna de las texturas PBR que se han colocado en las propiedades del material (el mismo editor permite abrir las imágenes que ya se han cargado en las propiedades) y sobre la imagen se puede ir acomodando el modelo desplegado para acomodar las texturas de manera correcta. Solo es necesario hacerlo con una imagen PBR no se requiere hacer esto para cada una de las imágenes con las que se cuenta.

2.5. La plataforma VRChat como alojamiento del simulador NERV

VRChat® es la plataforma social de realidad virtual de mayor crecimiento y una de las más populares. En VRChat los usuarios se conectan a un centro virtual desde el cual pueden acceder a una infinidad de mundos virtuales donde pueden socializar con los usuarios presentes en los mundos que visiten; cada usuario es representado por un avatar de su elección [35]. Al ser una plataforma de realidad virtual está pensada para su uso con un casco de realidad virtual, pero también existe una versión de computadora y otra para smartphones. VRChat permite a los usuarios cargar mundos personalizados construidos con Unity® mediante el uso de un SDK de VRChat. El SDK (Kit de Desarrollo de Software, por sus siglas en inglés) permite, mediante su lenguaje de programación, el uso de métodos que facilitan la programación de las interacciones características de VRChat. Para programar las interacciones y el contenido del mundo se utiliza el lenguaje de programación Udon, el cual permite la programación en dos formas: gráfica y tradicional. Para programar tradicionalmente se requiere agregar (dentro del SDK) el compilador Udon Sharp (U#), el cual está basado en C# y permite utilizar muchas las de bibliotecas disponibles en Unity®.

La última versión de NERV (LaVNeuM) fue alojada en VRChat por lo que la integración de los gemelos digitales junto con NERV también se colocó en esta plataforma. Para la implementación de los gemelos digitales dentro del simulador NERV se utilizó la programación tradicional (mediante código) con U#.

3. Desarrollo de los gemelos digitales

Se desarrollaron tres gemelos digitales a partir de tres planos de situación neumáticos. Estos planos de situación son utilizados en la asignatura de automatización industrial, junto con otros planos de situación, para que los estudiantes enlacen los movimientos de dos o más cilindros neumáticos.

3.1. Descripción de los sistemas neumáticos a desarrollar como gemelos digitales

3.1.1. Descripción de la Empacadora

El plano de situación de la empacadora se muestra en la figura 3.1. Consiste en dos cilindros neumáticos y un almacén vertical donde pueden irse acumulando las piezas en espera de ser empacadas. Los cilindros neumáticos tienen como objetivo seguir cierta secuencia de movimiento de modo que un conjunto de piezas pueda ser empacado en un contenedor. Existen al menos dos secuencias de movimiento que pueden ser implementadas para el correcto funcionamiento de la empacadora. En este caso se ha utilizado la secuencia de movimiento: A+ A- B+ B-, como puede apreciarse en la figura 3.2 que corresponde al diagrama de movimiento para el plano de situación de la empacadora. El uso de esta ecuación es únicamente con fines didácticos ya que el uso de esta secuencia de movimiento involucra estados repetidos (estados 1 y 3, además del 1 y 5 que completan el ciclo) por lo que, para lograr el movimiento requerido, se deben emplear memorias neumáticas ya sea mediante el diseño intuitivo o el empleo del método cascada. Para este caso específico la secuencia de movimiento A+ B+ A- B-, la cual no requiere el uso de memorias y es una secuencia más sencilla de implementar, funcionaría igualmente bien que la anterior. Las figuras 3.3 y 3.4 muestran los circuitos neumáticos para lograr la secuencia de movimiento requerida, mostrada en la figura 3.2, desarrollados con el software de simulación de circuitos neumáticos FluidSim®. En la figura 3.3 observamos una válvula neumática 5/2 y un temporizador negativo utilizados como memorias neumáticas para lograr el funcionamiento requerido (Elementos 2.4 y 1.6 respectivamente). En este caso la válvula 5/2 (elemento 2.4) fue transformada en una 3/2 con el simple hecho de tapar una de sus utilidades (la utilización 2). El temporizador neumático tiene como objetivo frenar el flujo de aire a través del pilotaje 14 de la válvula de potencia 1.1 una vez que esta se ha movido a su posición de trabajo; ya que, si no estuviera y se dejara enclavado el botón accionador, el flujo de aire en el pilotaje 14 permanecería durante el movimiento de retorno del cilindro A, ocasionando que la válvula de potencia se bloqueara y evitando que el cilindro A no regrese a su posición retraída. En la figura 3.4 se ha utilizado el método cascada para lograr el funcionamiento requerido por los pistones. Podemos observar que la secuencia de movimiento de la empacadora tiene 3 grupos, por lo que el método cascada requiere 3 líneas de presión y 2 válvulas de memoria 5/2.

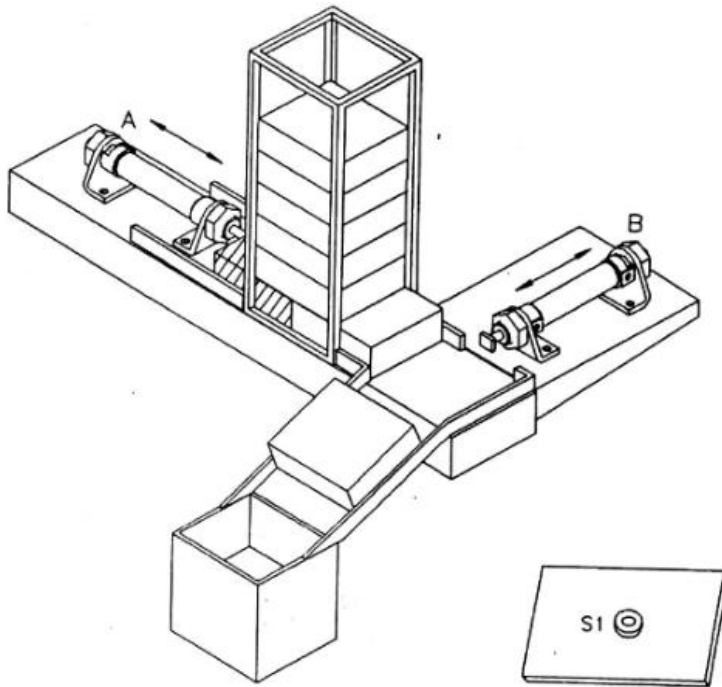


Figura 3.1. Plano de situación de la empacadora.

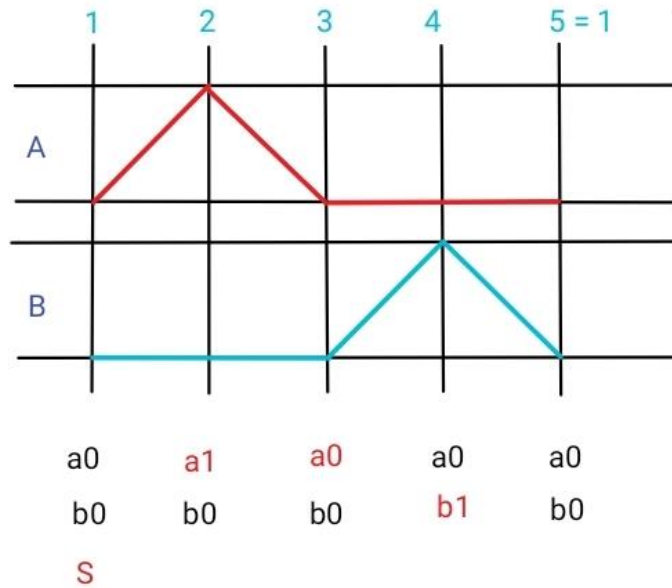


Figura 3.2. Diagrama de movimiento de la empacadora.

Empacadora

Ecuación de Movimiento

A+ A- B+ B-

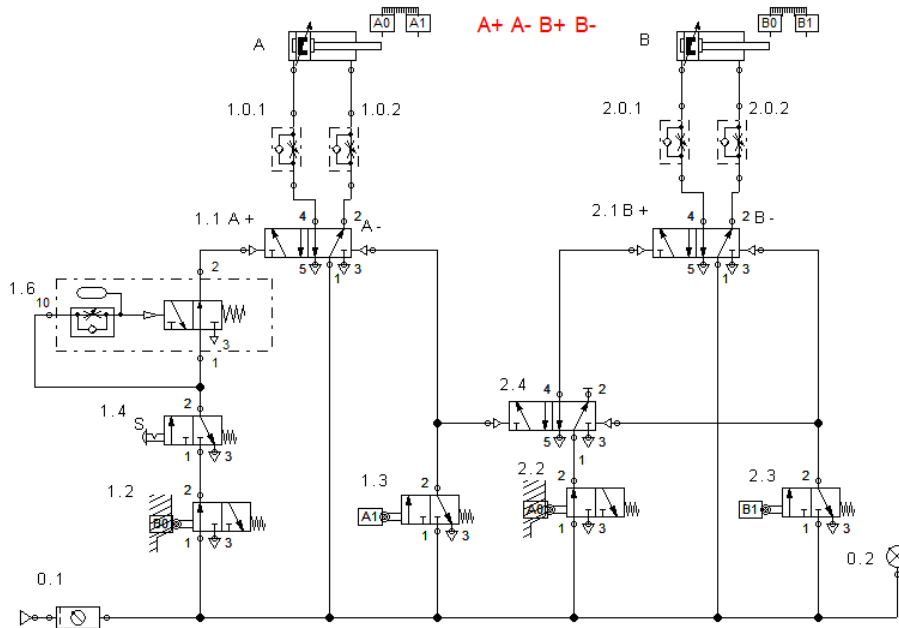


Figura 3.3. Circuito neumático de la empacadora. Se utilizaron: un temporizador negativo (elemento 1.6) y una válvula 5/2 (elemento 2.4) como memorias neumáticas.

Empacadora

Ecuación de movimiento

A+/A- B+/B-

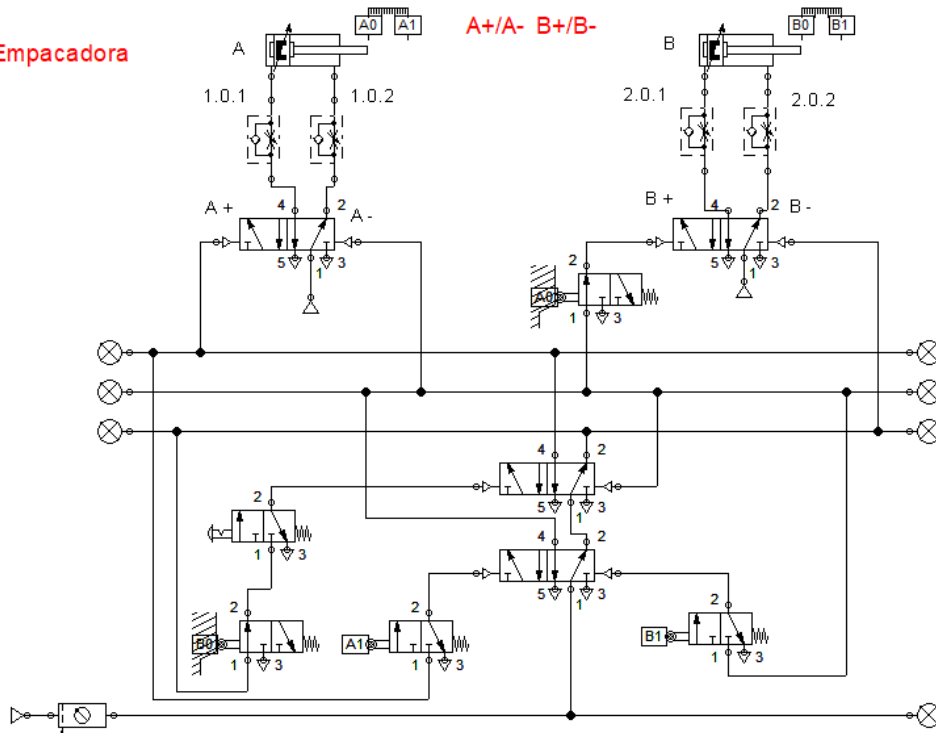


Figura 3.4. Circuito neumático de la empacadora. Se utilizó el método cascada con 2 válvulas de memoria 5/2.

3.1.2. Descripción de la Indentadora

La figura 3.5 muestra el plano de situación de la indentadora. Como puede observarse, este plano consiste de dos cilindros neumáticos, cuyo objetivo es posicionar la pieza cúbica de manera adecuada para lograr que el cabezal del cilindro B logre la indentación de la pieza. En este caso la secuencia de movimiento es importante ya que una secuencia errónea provocaría que la pieza se mueva evitando una indentación adecuada. Ambos cilindros deben iniciar retraídos. El inicio del ciclo lo comienza el cilindro A con su extensión. Este cilindro deberá permanecer en esta posición durante la indentación para evitar que la pieza se mueva durante la indentación. La indentación la realiza el cilindro B con su extensión. En este punto podría colocarse un temporizador para permitir la correcta indentación de la pieza (en este caso no se ha colocado el temporizador, pero un ejemplo con temporizador se muestra en la sección 3.1.2). Posterior a la indentación se retrae el cilindro B y luego el cilindro A, completándose el ciclo. Como puede observarse de la figura 3.6, la secuencia de movimiento correcta es entonces: A+ B+ B- A-. Si se llegara a optar por la secuencia A+ A- B+ B-, que también luce como una ecuación de movimiento que cumpliría con la indentación, la pieza podría moverse de la posición requerida cuando el cilindro A regrese a su posición retraída. Otras secuencias de movimiento no cumplirán con la indentación de la pieza. Como puede observarse de la figura 3.6. se repiten los estados 2 y 4, además del 1 y 5 que completan el ciclo de movimiento, por lo que es requerido el uso de memorias neumáticas. Aquí también se puede optar por los métodos de diseño intuitivo o el método cascada. La figura 3.7, muestra el diseño del circuito neumático empleando el método intuitivo. Podemos observar el uso de las dos memorias neumáticas 5/2 (elementos 1.5 y 2.4). La figura 3.8 muestra el diseño del circuito neumático empleando el método cascada. En este caso debido a que hay dos grupos de movimiento se requieren dos líneas de presión y únicamente una válvula de memoria.

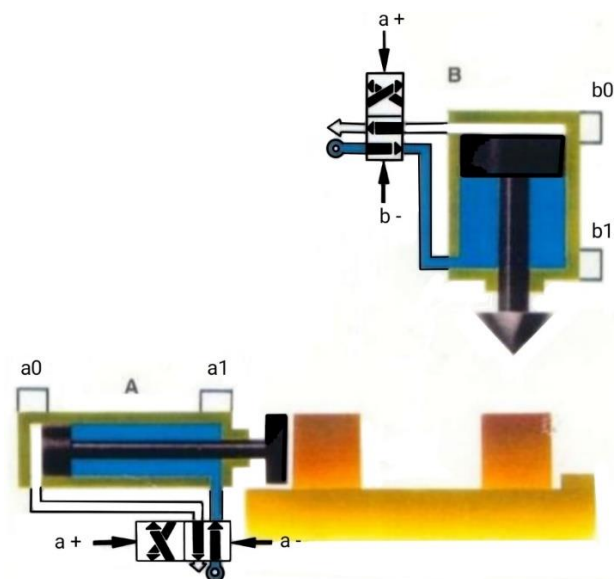


Figura 3.5. Plano de situación de la indentadora.

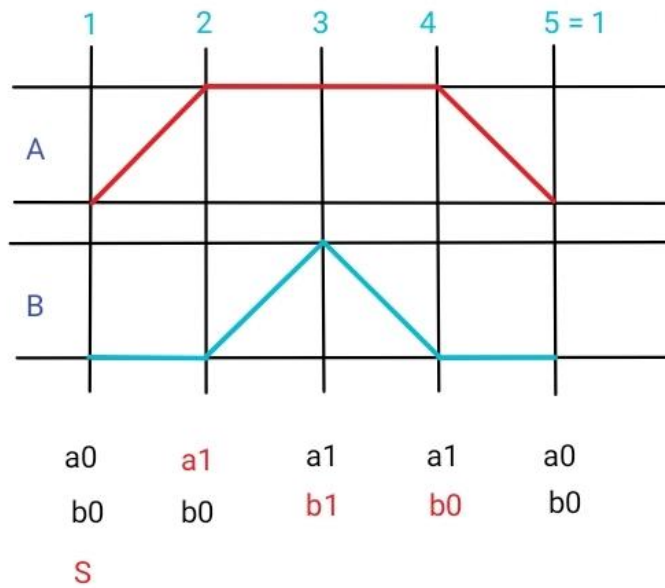


Figura 3.6. Diagrama de movimiento de la indentadora.

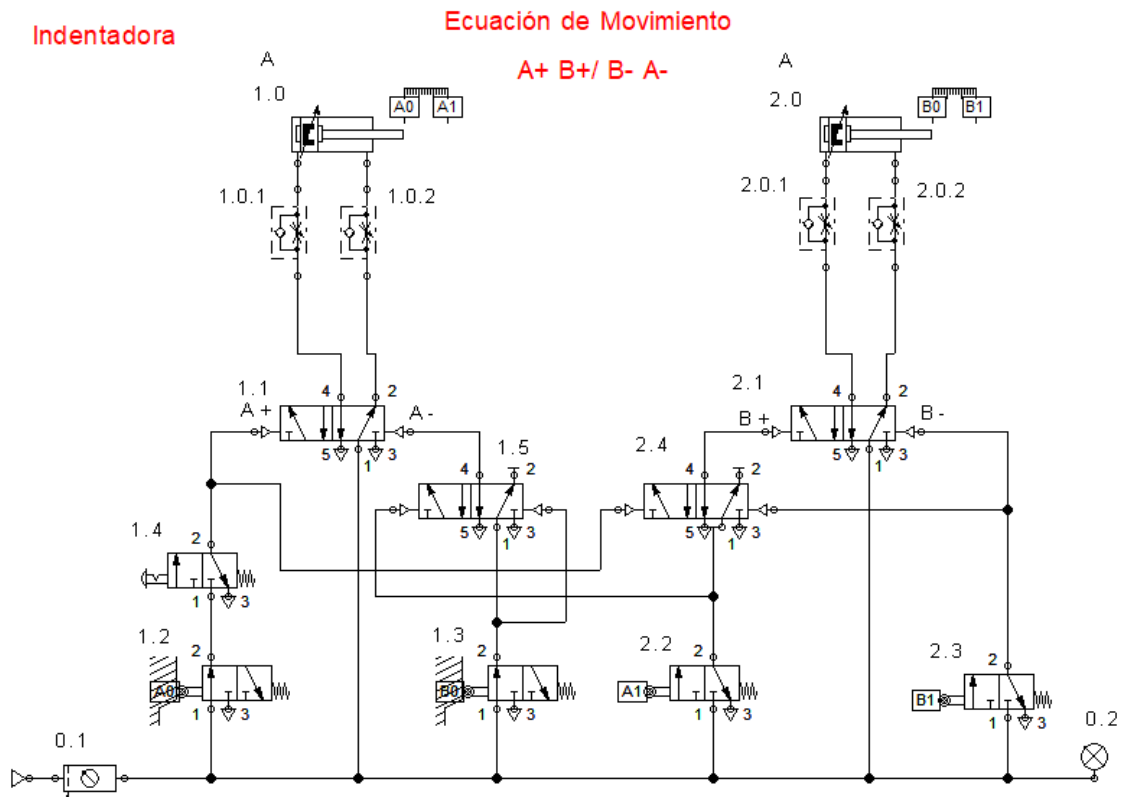


Figura 3.7. Circuito neumático de la indentadora con dos válvulas de memoria 5/2 (elementos 1.5 y 2.4).

Indentadora

Ecuación de Movimiento

A+B/-B-A

I / II

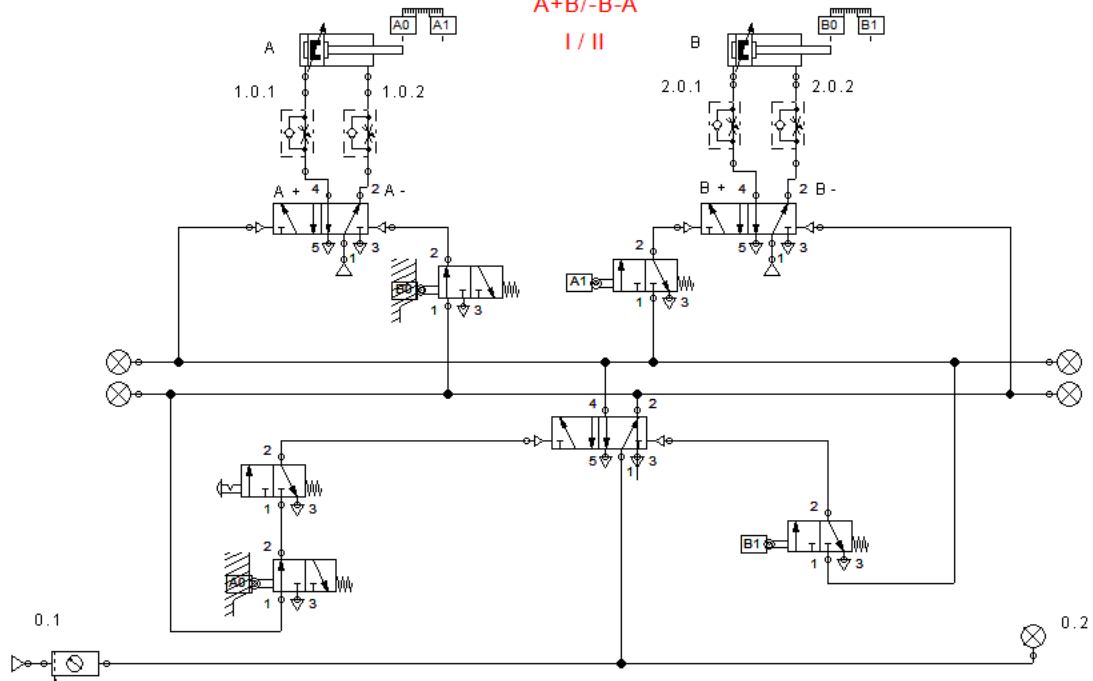


Figura 3.8. Circuito neumático de la indentadora empleando el método cascada. Se observa que solo se requiere una válvula de memoria.

3.1.3. Descripción de la Estampadora

El plano de situación de la estampadora se muestra en la figura 3.9. En este caso se tienen tres cilindros neumáticos y al igual que en el caso de la empaquetadora se tiene un almacén vertical que permite que las piezas se acomoden en espera de ser estampadas. Los tres cilindros inician retraídos. El ciclo de movimiento comienza con la extensión del cilindro A. Después inicia la extensión del cilindro B con lo cual se realiza el estampado de la pieza. Aquí puede optarse por colocar un elemento temporizador para permitir un proceso adecuado de estampado. Luego de realizado el estampado el cilindro B se retrae y el cilindro C puede expulsar la pieza con la extensión y retracción del cilindro C. Para finalizar el ciclo el cilindro A se retrae con lo que puede dar inicio un nuevo ciclo de estampado. A partir de lo descrito se espera que la secuencia de movimiento sea: A+ B+ B- C+ C- A-, como puede observarse del diagrama de movimiento de la figura 3.10. La figura 3.11 muestra el circuito neumático implementado, con el método cascada, para la estampadora. Se puede apreciar que debido a que hay tres grupos se tiene igual número de líneas de presión y dos válvulas de memoria 5/2. También es posible apreciar el uso de un temporizador normalmente cerrado (positivo) para retrasar el cambio al grupo 2. Esto ayuda a que el estampado de la pieza se realice de manera correcta pues el cilindro 2 permanece extendido una cierta cantidad de tiempo.

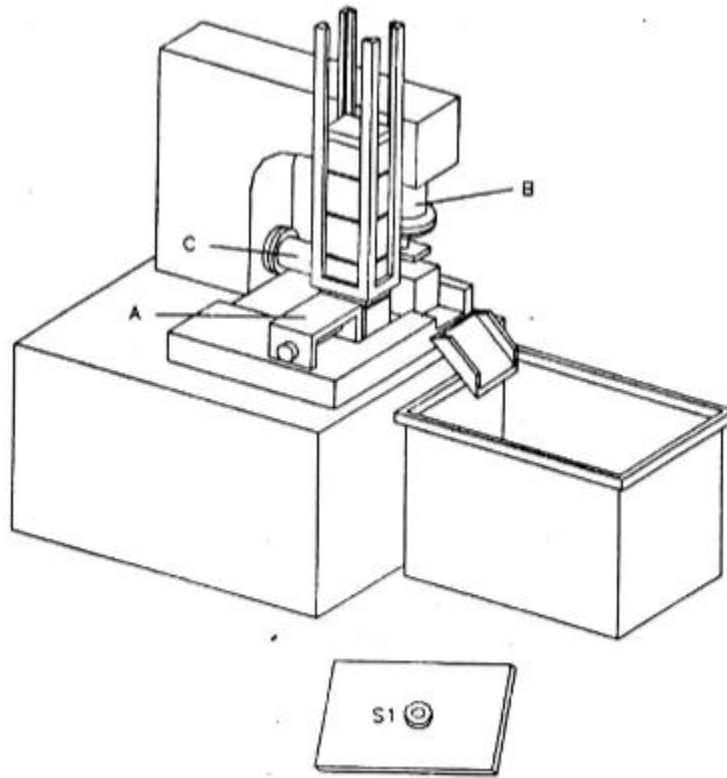


Figura 3.9. Plano de situación de la estampadora.

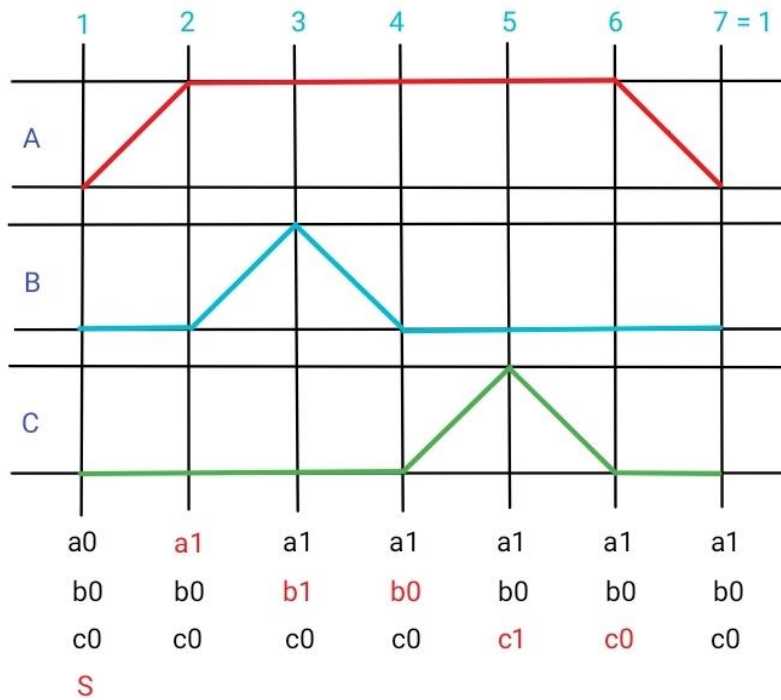


Figura 3.10. Diagrama de movimiento de la estampadora.

Ecuación de Movimiento

$$A + B + B - C + C - A -$$

I / II / III

Estampadora

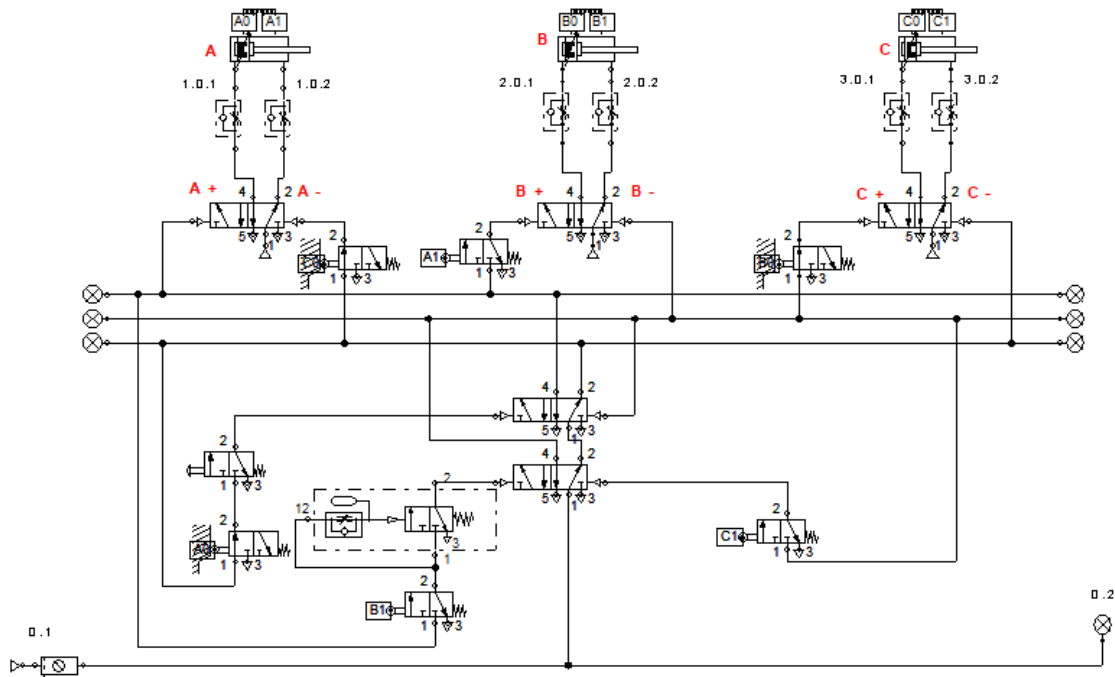


Figura 3.11. Circuito neumático de la estampadora elaborado con el método cascada. Se utilizó un temporizador para postergar el cambio al grupo 2.

3.2. Diseño CAD de los planos de situación neumáticos

A partir de los planos de situación de la empacadora, la indentadora y la estampadora descritos en la sección 3.1. es posible construir modelos CAD de los planos para su integración como gemelos digitales en el simulador NERV. Los modelos CAD fueron realizados utilizando la paquetería Autodesk Inventor®. Cada modelo fue diseñado de tal modo que cupiese en un volumen de 1 m^3 , sin considerar la mesa de soporte.

3.2.1. Modelos CAD de los cilindros neumáticos y sus soportes

Todos los planos de situación vistos en la sección 3.1 tienen en común el uso de cilindros neumáticos. En la actualidad hay un sin número de recursos CAD disponibles en la web para uso de los diseñadores con lo cual se facilita el diseño de ensambles más complejos. En vista de esto se buscaron en la Web cilindros neumáticos lo más parecidos a los utilizados en el laboratorio de automatización industrial (ver figura 1.1). Se encontró que la empresa Festo® diseña cilindros bastante parecidos a los utilizados en el laboratorio y además proporciona

los modelos en CAD a través de una plataforma intuitiva y fácil de usar. La página puede ser consultada en la referencia [36]. Los cilindros neumáticos utilizados en el desarrollo de los gemelos digitales, son entonces, similares y solo cambian por el hecho de tener distintas carreras. La figura 3.12 muestra uno de los cilindros utilizados en el desarrollo de los gemelos digitales. El modelo CAD del cilindro consta de un vástago y una tuerca, además del cuerpo del cilindro (ver figura 3.13). La tuerca no fue requerida en la construcción de los gemelos digitales.



Figura 3.12. Modelo CAD de los cilindros utilizados en la implementación de gemelos digitales y proporcionados por Festo® [36]. El cilindro consta de un vástago y el cuerpo del cilindro. Además, posee una tuerca situada en el vástago la cual no fue requerida.

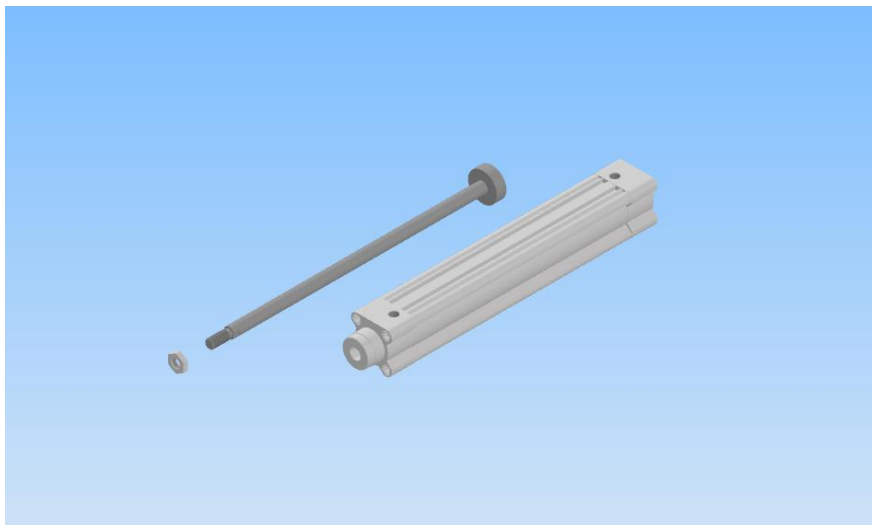


Figura. 3.13. El cilindro mostrado en la figura 3.14 consta de un vástago y una tuerca, además del cuerpo del cilindro. El cilindro fue obtenido de la referencia [36].

Para la sujeción de los cilindros se diseñó un soporte común para los 3 gemelos digitales, el cual se muestra en la figura 3.14. Por cada cilindro se requiere un par de estos soportes para su sujeción. El soporte requiere además 4 pernos M5 para lograr la correcta sujeción del cilindro (8 pernos en total si se consideran los dos soportes por cilindro). Para el perno también se recurrió a modelos CAD prefabricados. En este caso se utilizó la plataforma Traceparts [37] la cual cuenta con una amplia biblioteca de modelos 3D incluidos pernos de marcas comerciales. En realidad, el diseño de los tres gemelos digitales solo considera un tipo de perno y en algunos casos sólo varía su longitud.

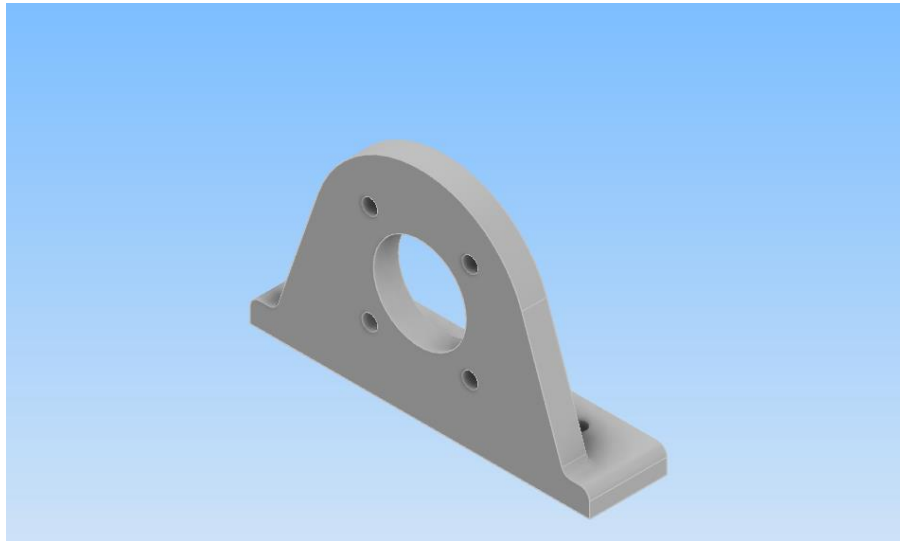


Figura 3.14. Soporte de los cilindros neumáticos. Por cada cilindro se requiere un par de soportes.



Figura 3.15. Perno M5 utilizado para la sujeción de los cilindros neumáticos Obtenido de [37].

3.2.2. Modelos CAD de los ensambles de los gemelos digitales

Ensamble de la Empacadora

El diseño de la empacadora se basó en el plano de situación mostrado en la figura 3.1 y en el empaqueo de una pieza de dimensiones mostradas en el anexo A y cuyo modelo CAD se muestra en la figura 3.16.

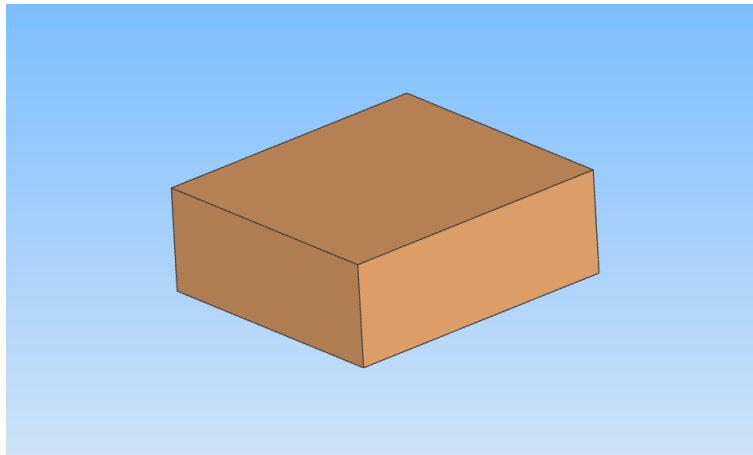


Figura 3.16. Modelo CAD de la pieza con la cual se diseñaron las dimensiones de la empacadora.

Para el movimiento de las cajas y la fijación de los cilindros neumáticos se diseñó la base mostrada en la figura 3.17. En esta base se fijan los soportes de los cilindros y también los elementos que sirven de guía para el correcto desplazamiento de las cajas.

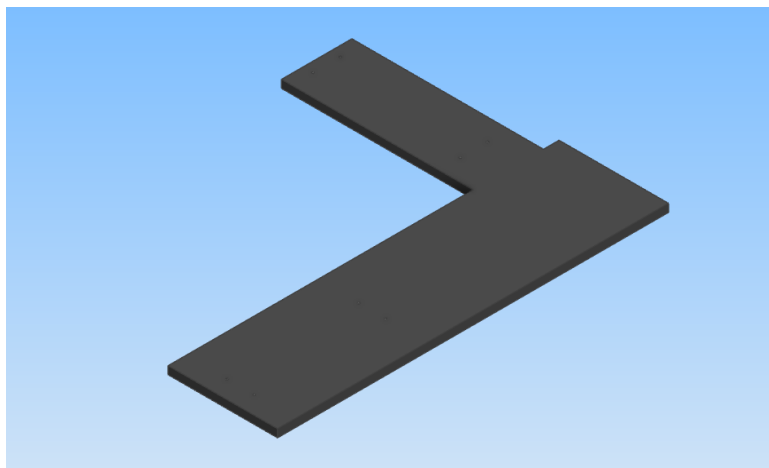


Figura 3.17. Base de la empacadora para el movimiento de los paquetes y la fijación de los cilindros neumáticos y las guías.

El almacén vertical que permite que las piezas puedan esperar a ser empacadas se muestra en la figura 3.18. Este elemento se construyó mediante el ensamble de las piezas mostradas en la vista explosionada de la figura 3.19. Dicho elemento fue diseñado por partes para que a cada uno de sus componentes se le adaptara un colisionador *Box Collider*. Los colisionadores *Box Collider* se adaptan bien a las geometrías prismáticas rectangulares o geometrías cercanas a esta forma, cubriéndolas por completo; por lo que diseñar el elemento de almacenamiento por partes ahorra el trabajo de modificaciones adicionales del colisionador. Si se optara por hacer una sola pieza para el soporte de almacenamiento, al agregarle un *Box Collider*, este cubriría por completo la pieza, por lo que se requerirían modificaciones adicionales del colisionador para lograr el espacio vacío necesario para que se acumulen las piezas. En realidad, las partes que conforman el elemento de almacenamiento no son prismáticas rectangulares debido al hueco en su interior, sin embargo, Unity pasa por alto esto y asigna los colisionadores como si no tuvieran este hueco (esto se entenderá de mejor manera cuando se asignen los colisionadores en la sección de la programación de la empacadora). Para la fijación de las piezas del elemento de almacenamiento no se consideró ningún otro elemento de unión mecánico como pernos.

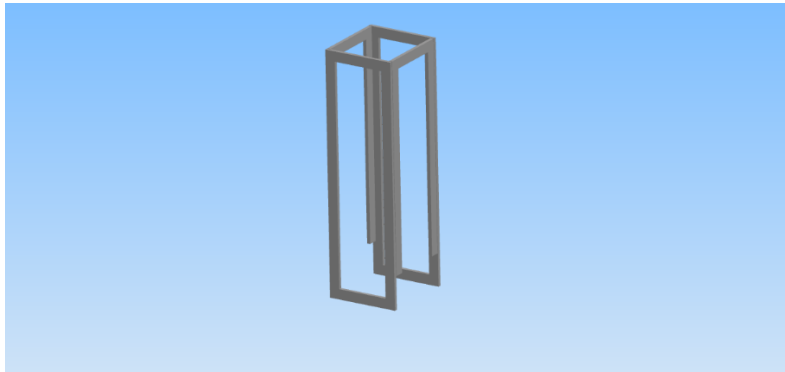


Figura 3.18. Almacén vertical de la empacadora donde se depositan las piezas en espera de ser empacadas.

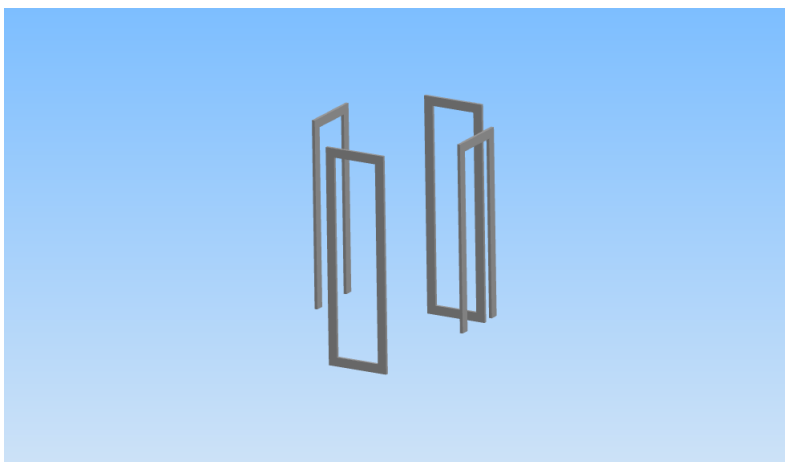


Figura 3.19. Elementos que conforman el almacén vertical.

Las elementos guía que permiten el correcto posicionamiento y deslizamiento de las piezas se muestran en la figura 3.20 en un ensamble conjunto con la base para indicar la posición relativa de estos componentes respecto de la base. En la figura 3.21 se muestra una vista explosionada del mismo ensamble, para observar que las guías fueron igualmente diseñadas en figuras prismáticas rectangulares para poder agregarles de manera sencilla sus colisionadores (*colliders*) una vez exportados a Unity (todos los elementos que de aquí en adelante se nombren como elementos guías sirven para el correcto posicionamiento y deslizamiento de las piezas). Para la fijación de los elementos guías tampoco fue considerado ningún medio de fijación mecánico.

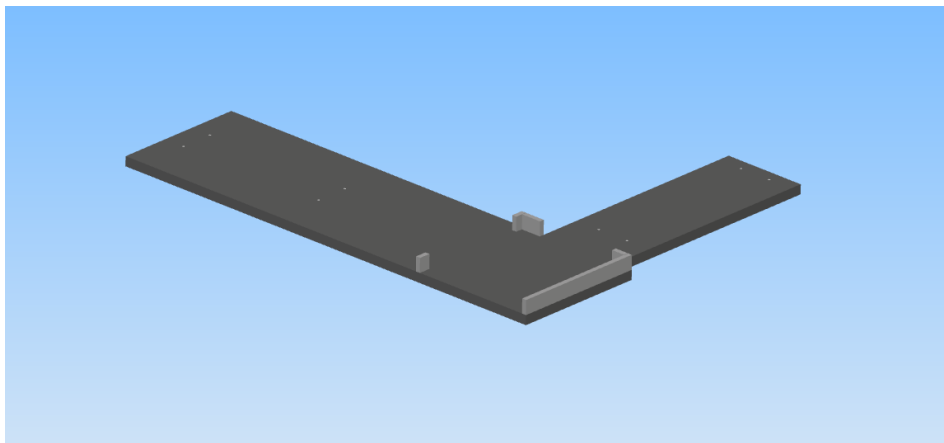


Figura 3.20. Ensamble de la base y las guías de la empacadora que mantienen los paquetes en una posición adecuada.

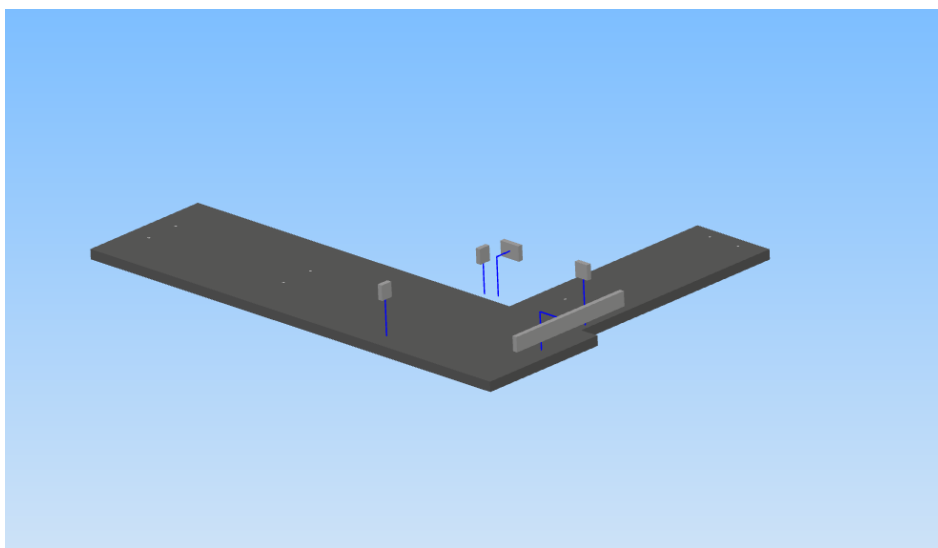


Figura 3.21. Vista explosionada del ensamble base-guías mostrado en la figura 3.20.

La figura 3.22 muestra el ensamble de la rampa junto con sus elementos guía. Este elemento es por donde las cajas se deslizan antes de ser empacadas. La figura 3.23 presenta una vista explosionada del ensamble mostrado en la figura 3.22. Como estos componentes interactúan directamente con las piezas requieren colisionadores que se adapten bien a su geometría, por lo que también fue requerido hacer este elemento en partes más pequeñas y simples.

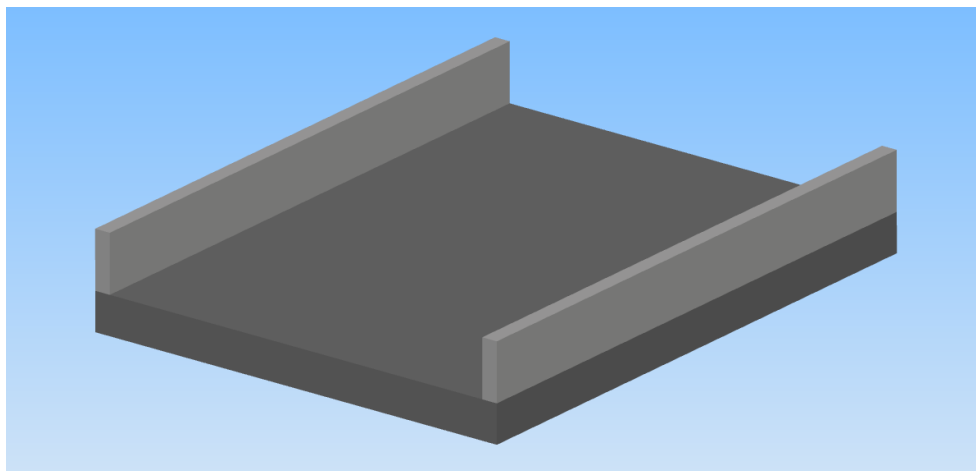


Figura 3.22. Ensamble de la Rampa y las guías de pieza en la empacadora.

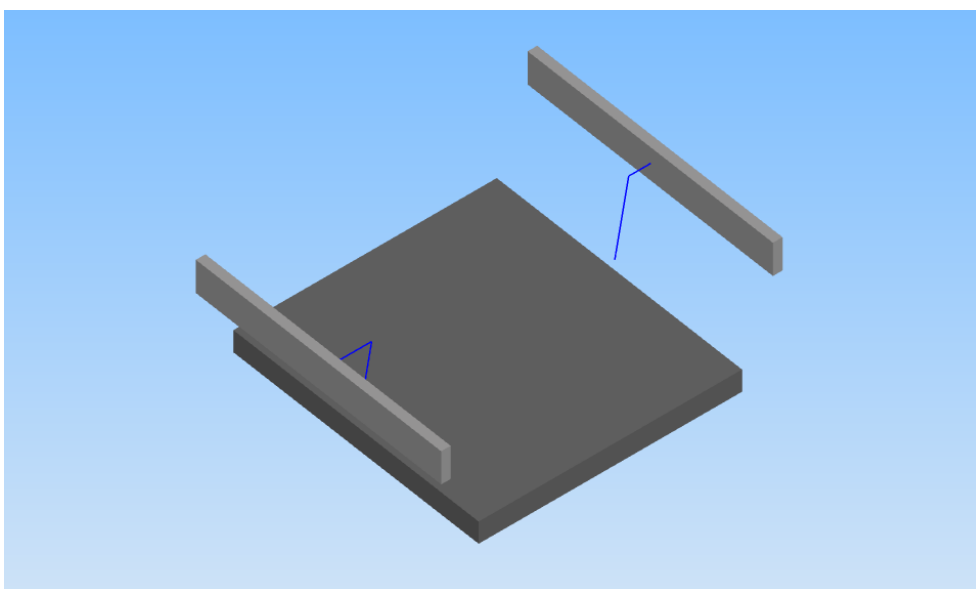


Figura 3.23. Vista explosionada del ensamble mostrado en la figura 3.22.

Para los cilindros A y B se consideraron carreras de 260 y 300 *mm* respectivamente. Como ya se mencionó los cilindros son iguales a los mostrados en la figura 3.12 y solo varía el largo de sus carreras. El diseño del cabezal del cilindro A se muestra en la figura 3.24 en ensamble con su vástago correspondiente. La figura 3.25 muestra la vista explosionada del ensamble de la figura 3.24. Cabe mencionar que el cabezal del cilindro A, además de empujar la pieza impide que las piezas restantes caigan a la base de la empacadora cuando el cilindro A está extendido, de ahí que la forma de este cabezal tenga el grosor mostrado.

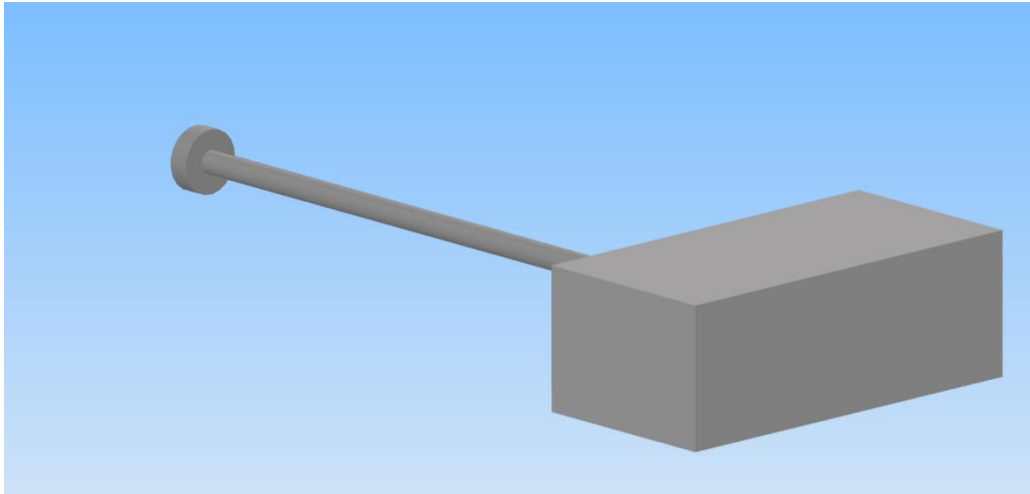


Figura 3.24. Ensamble cabezal - vástago del cilindro A de la empacadora.

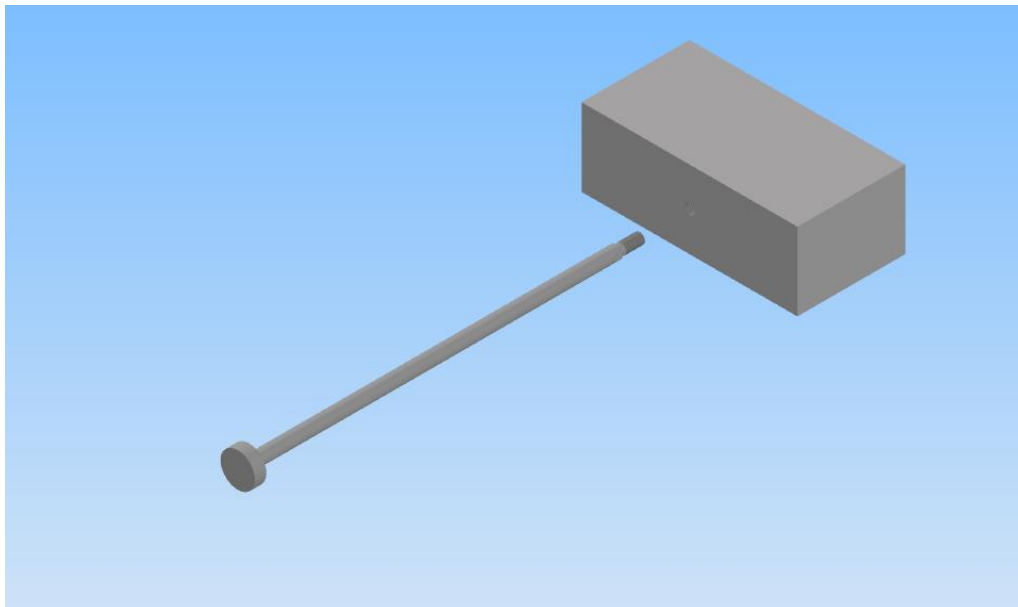


Figura 3.24. Vista explosionada del vástago y cabezal del cilindro A de la empacadora.

El diseño del cabezal del cilindro B se muestra en la figura 3.25 en ensamble con su vástago, mientras que la figura 3.36 muestra el despiece de este ensamble. Este cabezal tiene la función de empujar la pieza para ser empacada. La empacadora se construyó de tal forma que hubiera cierta holgura entre sus elementos (entre las piezas, las guías y el soporte de almacenamiento), por lo que es importante que este cabezal tenga forma rectangular para dirigir de manera correcta la pieza hacia su destino. Otras formas de cabezal podrían dirigir la pieza de manera incorrecta provocando que se atore con el almacén vertical.

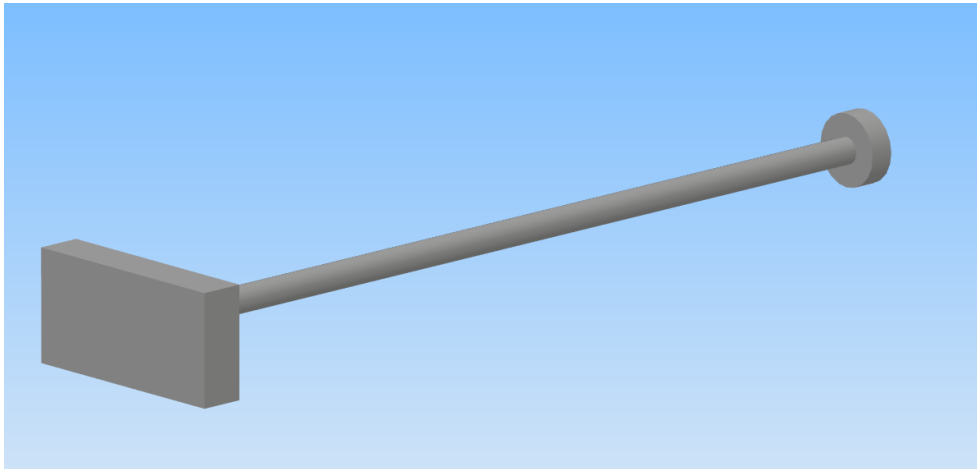


Figura 3.25. Ensamble vástago cabezal del cilindro B de la indentadora.

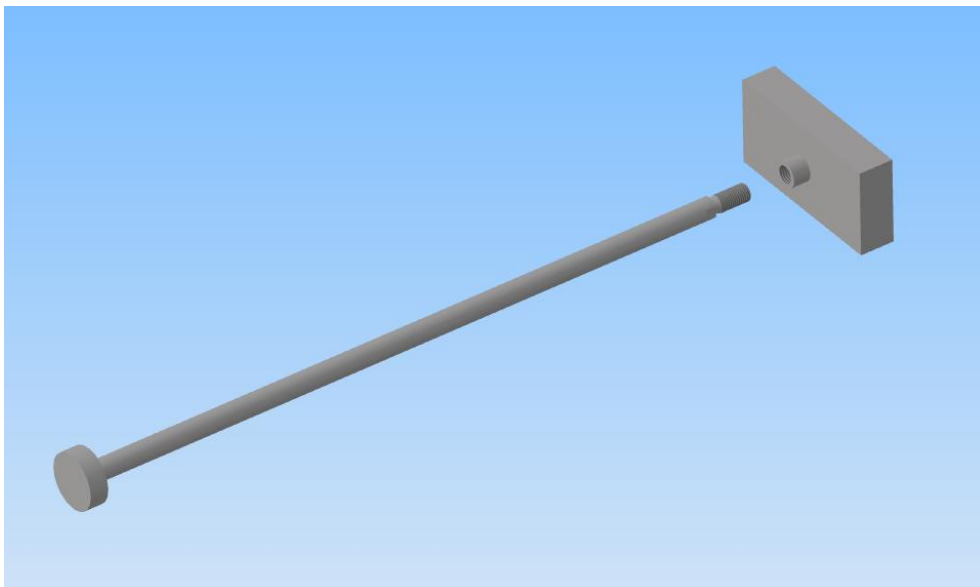


Figura 3.26. Despiece del ensamble mostrado en la figura 3.25.

La figura 3.27 muestra el ensamble de los cilindros junto con sus soportes fijados a la base. La figura 3.28 muestra una vista explosionada del ensamble de la figura 3.27 donde se pueden observar por separado los elementos de unión (pernos) y los soportes que mantienen fijos los cilindros neumáticos.

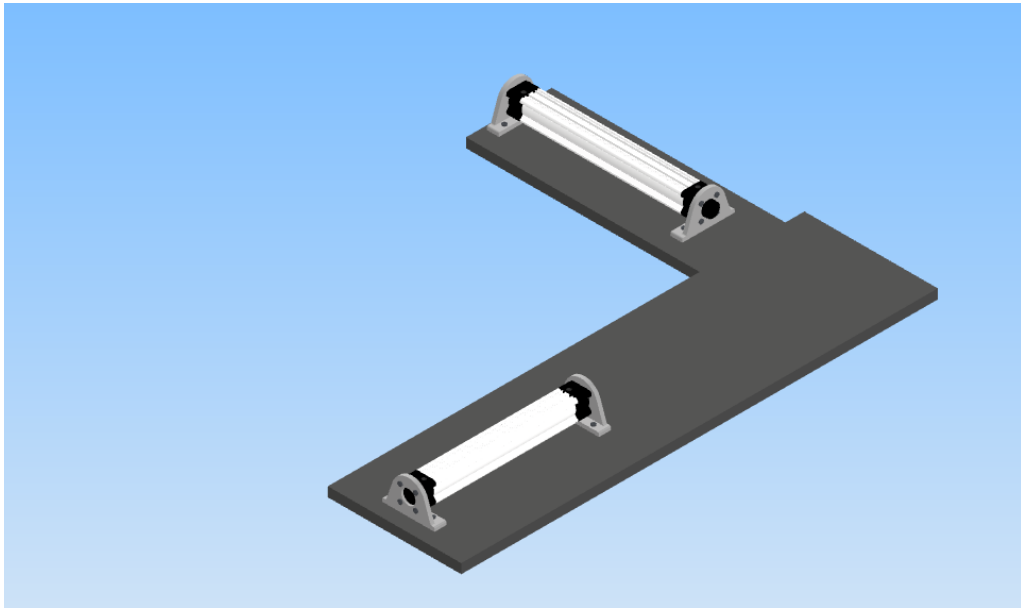


Figura 3.27. Ensamble de los cilindros con sus soportes en la base de la empacadora.

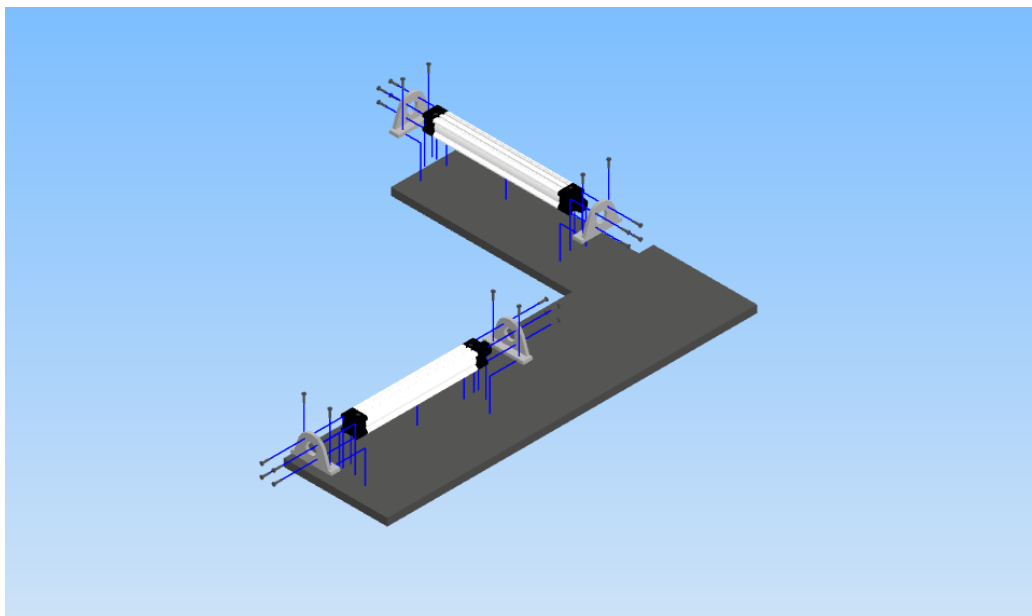


Figura 3.28. Vista explosionada del ensamble mostrado en la figura 3.27.

La figura 3.29 muestra el ensamblaje completo de la empacadora donde se pueden observar cada uno de los elementos descritos anteriormente además de la mesa donde descansa el modelo. Aquí no se presenta el modelo CAD de la mesa, pero puede consultarse su plano en el anexo A.

Este ensamblaje solo muestra la forma final que se espera del gemelo digital y resultó útil cuando se construyó, a partir de las piezas individuales, el modelo final en Unity®. Sin embargo, no es conveniente exportar el ensamblaje completo a Unity; ya que, como se ha mencionado anteriormente, al agregar los colisionadores (*Colliders*) cubrirían por completo el ensamblaje y no a los elementos individuales, por lo que se requeriría trabajo adicional para adaptar a cada componente su respectivo colisionador.

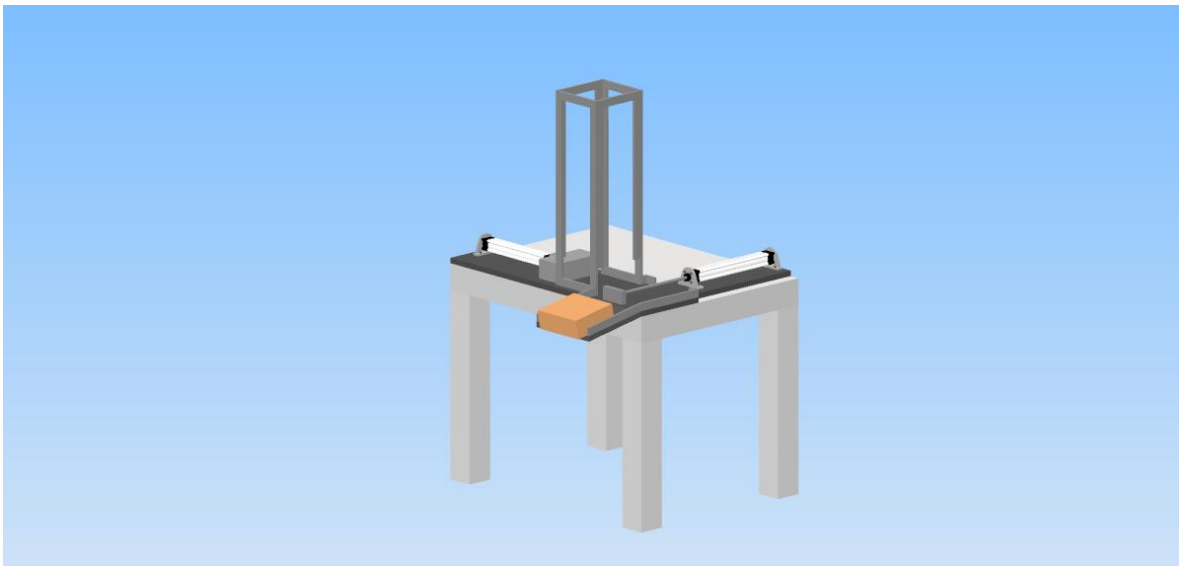


Figura 3.29. Vista del gemelo digital de la empacadora.

Ensamble de la Indentadora

El modelo CAD de la indentadora fue diseñado con base en el plano de situación de la figura 3.5 y una pieza de trabajo cúbica de dimensiones mostradas en el anexo B y cuyo modelo CAD se muestra en la figura 3.30.

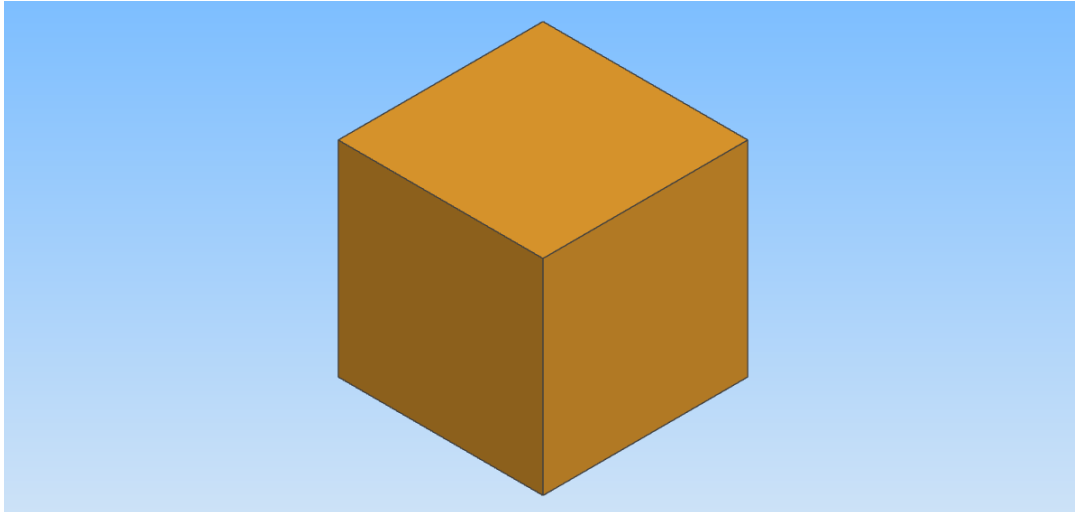


Figura 3.30. Pieza de trabajo cúbica para el modelo de la indentadora.

Se construyó una base horizontal para fijar el cilindro A y otra vertical para fijar el cilindro B. La base horizontal también sirve para el deslizamiento de los cubos durante la indentación. Las figuras 3.31 y 3.32 muestran las bases de fijación del cilindro A y del cilindro B respectivamente. En adelante nos referiremos a estas bases como base horizontal y base vertical de la indentadora.

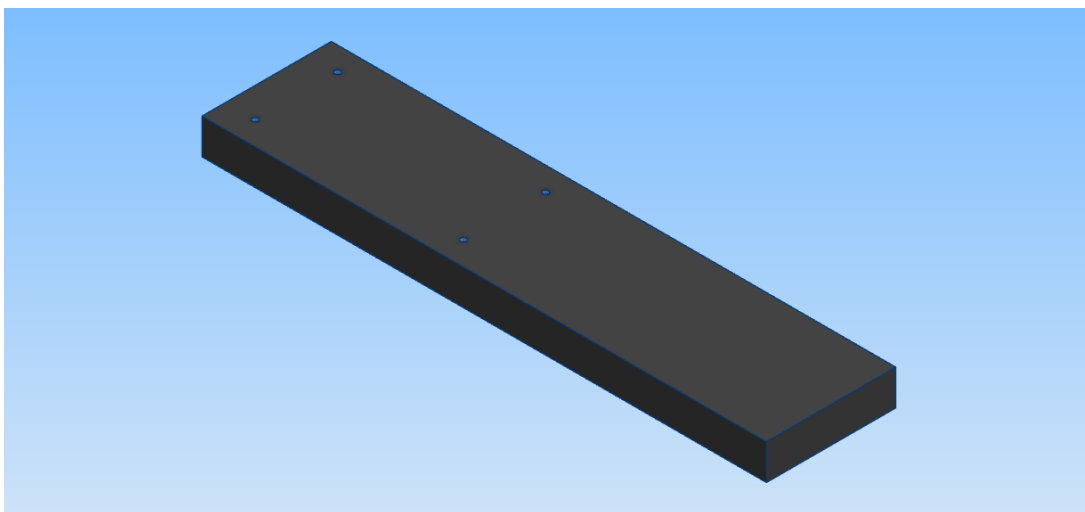


Figura 3.31. Base de fijación del cilindro A y de deslizamiento de la pieza de trabajo (base horizontal).

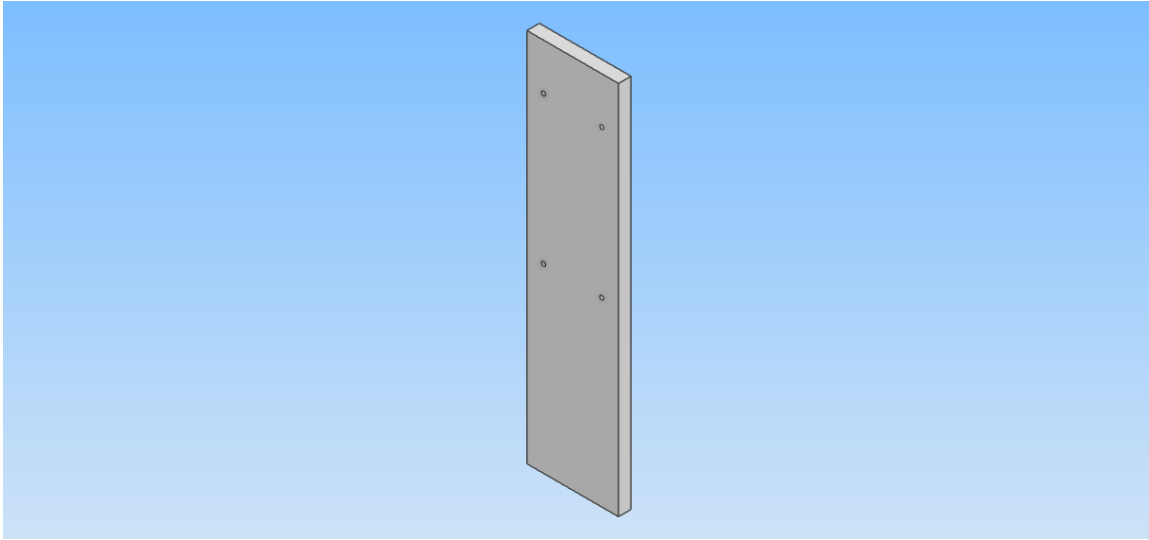


Figura 3.32. Base de fijación del cilindro B (base vertical).

Para este modelo también son requeridas guías para controlar el movimiento de la pieza cúbica. En la figura 3.33 se muestran estas guías montadas sobre la base donde se desliza la pieza de trabajo. En la figura 3.34 se muestra un vista explosionada del ensamble mostrado en la figura 3.33. De igual manera que en el caso de la empacadora, las guías fueron diseñadas individualmente como prismas rectangulares para poder asignarles colisionadores *Box Colliders* de manera sencilla.

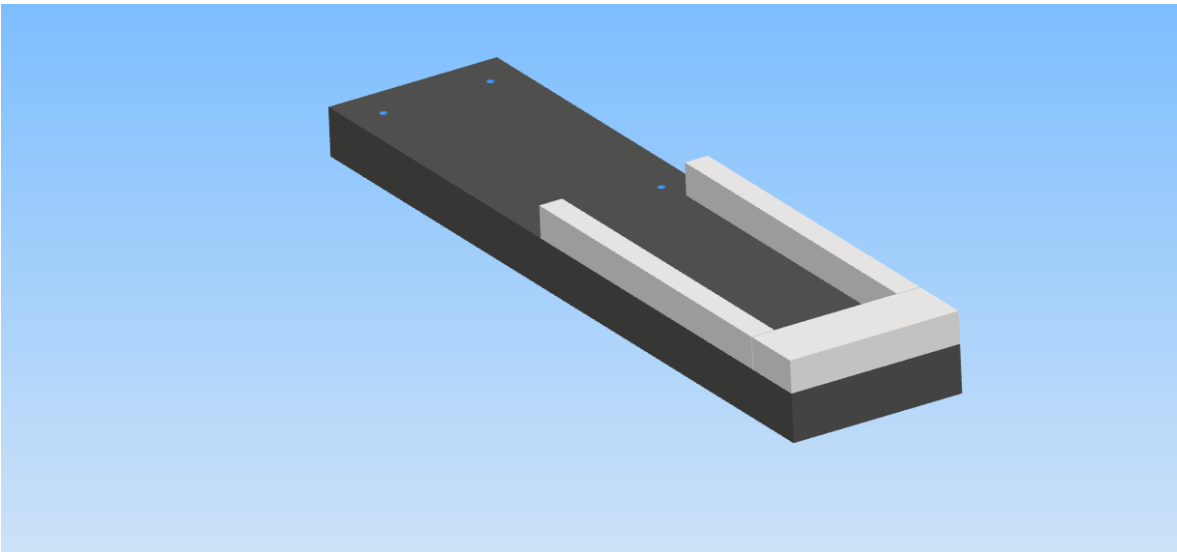


Figura. 3.33. Guías montadas en la base horizontal para el correcto deslizamiento y posicionamiento de la pieza de trabajo.

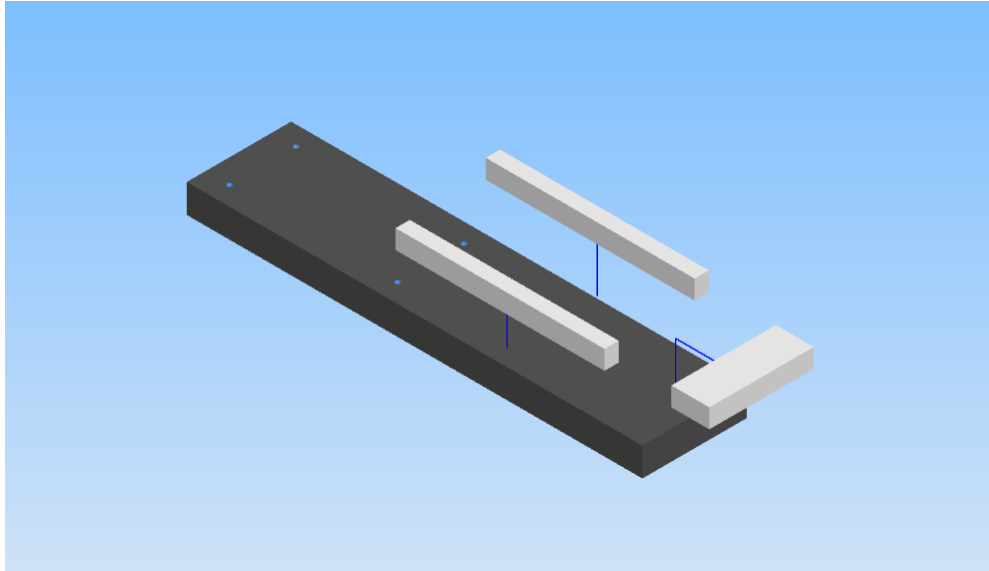


Figura 3.34. Vista explosiva del ensamble mostrado en la figura 3.33.

En todos los gemelos digitales se utilizaron los mismos soportes para los cilindros (ver figura 3.14). En este caso se requirió agregar elementos adicionales a los soportes de los cilindros para elevarlos un poco y que tanto el cabezal que realiza la indentación como el cabezal que posiciona la pieza estuvieran perfectamente centrados. Estos elementos adicionales tienen las mismas dimensiones para ambos cilindros. Nos referiremos a estos elementos como elevadores. La Figura 3.35 muestra los elevadores ensamblados con la base horizontal y la figura 3.36 muestra los elevadores ensamblados con la base vertical.

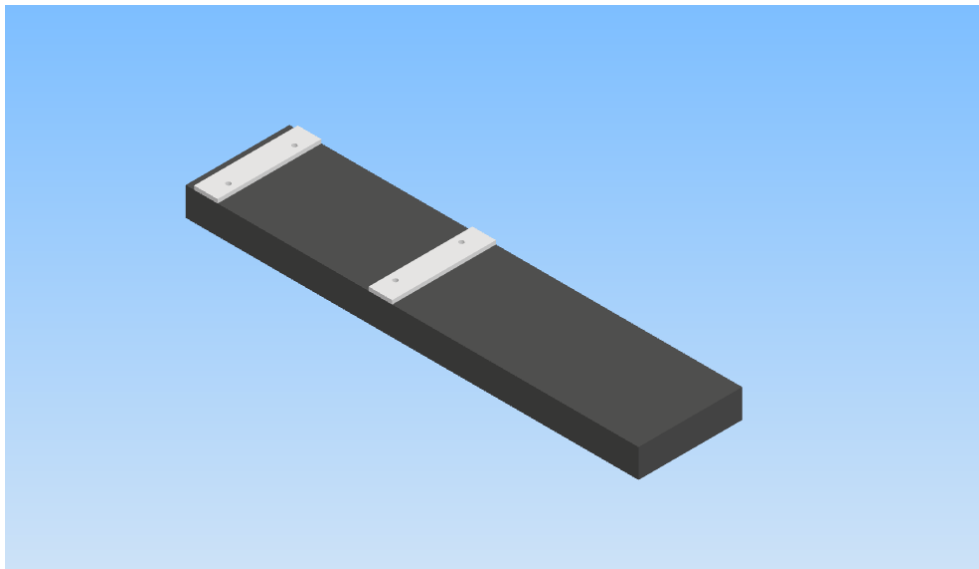


Figura 3.35. Ensamble de la base horizontal con los elementos que ajustan la posición de los soportes del cilindro A.

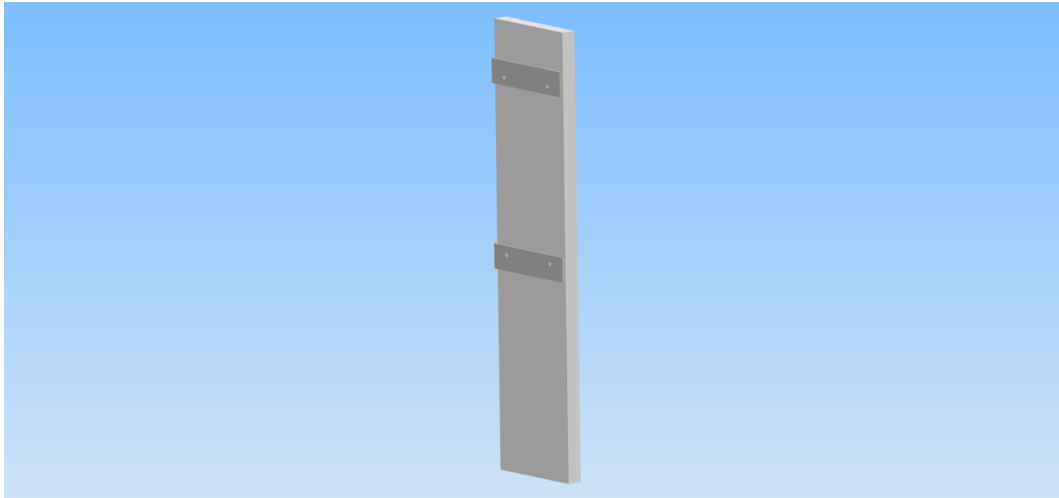


Figura 3.36. Ensamble de la base vertical con los elementos que ajustan la posición del cilindro B.

El ensamble del cabezal del cilindro A junto con su vástago se muestra en la figura 3.37 y una vista explosionada del mismo ensamble se muestra en la figura 3.38. Este cabezal solo se utiliza para posicionar la pieza cúbica de manera correcta. Debido a que no existe mucha holgura entre las guías y la pieza de trabajo; el cabezal podría tener cualquier forma ya que las guías impedirán que la pieza se coloque de manera inadecuada, sin embargo, se prefirió hacer un cabezal rectangular para evitar complicaciones.

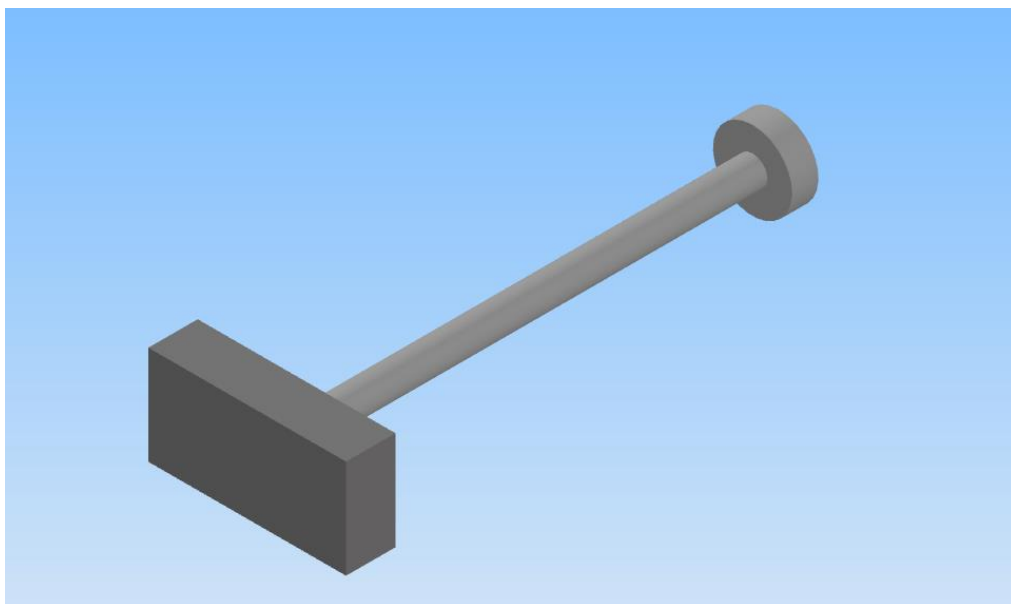


Figura 3.37. Ensamble del cabezal del cilindro A con su vástago.

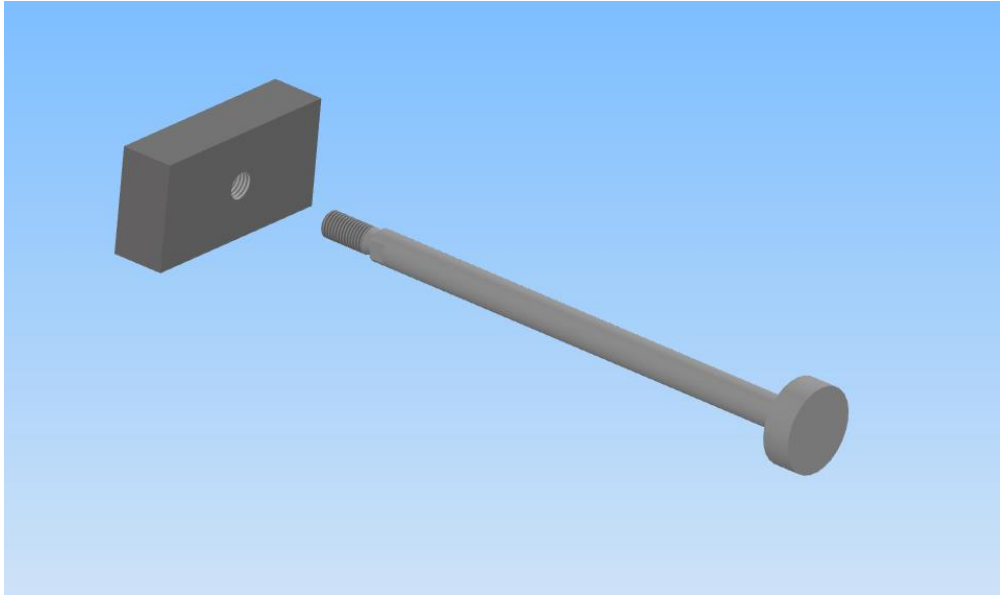


Figura 3.38. Vista explosionada del ensamble vástago – cabezal de la figura 3.37.

El ensamble del cabezal del cilindro B (Indentador) junto con su vástago se muestra en la figura 3.39 y una vista explosionada del ensamble se muestra en la figura 3.40. El indentador se realizó a semejanza del mostrado en el plano de situación (fig. 3.5), por lo que es casi del mismo tamaño que la pieza de trabajo.

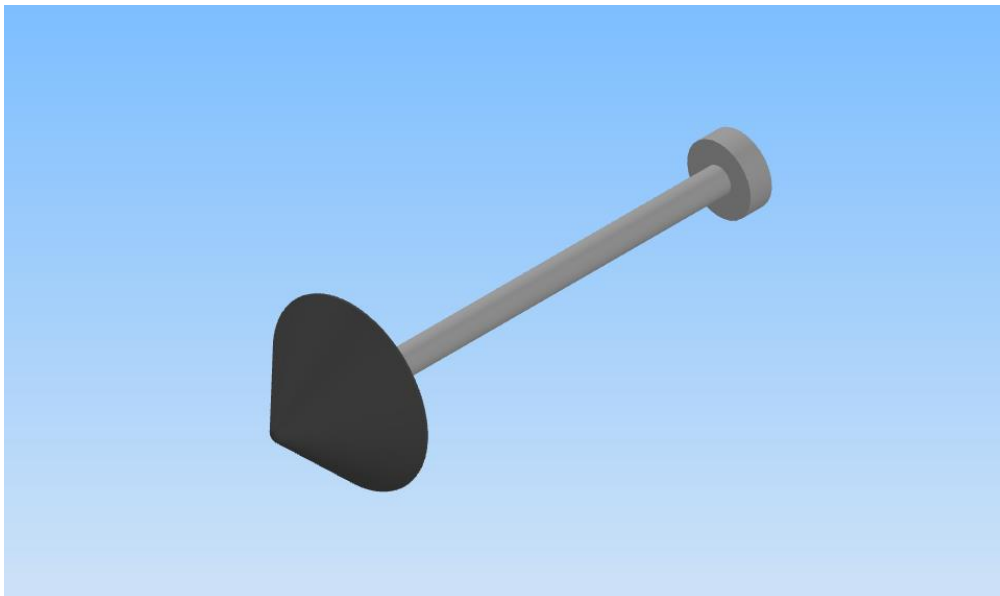


Figura 3.39. Ensamble vástago – Indentador.

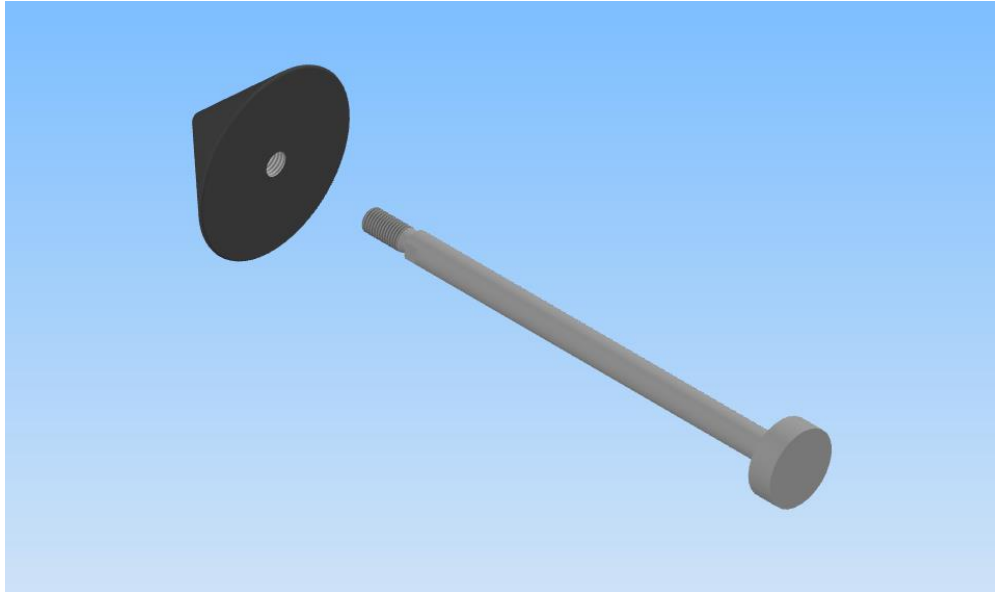


Figura 3.40. Vista explosionada del ensamble de la figura 3.30.

La figura 3.41 muestra ensambladas las bases horizontal y vertical de la indentadora junto con los cilindros y sus soportes; además de los elevadores mostrados en las figuras 3.35 y 3.36. En la figura 3.42 se muestra una vista explosionada de estos ensambles. Se puede notar en la figura 3.41 que las bases no están unidas. Esto se debe a que la base vertical en realidad se une a una de las guías y no directamente con la base horizontal. Los cilindros utilizados tienen una carrera de 100 *mm* en ambos casos.

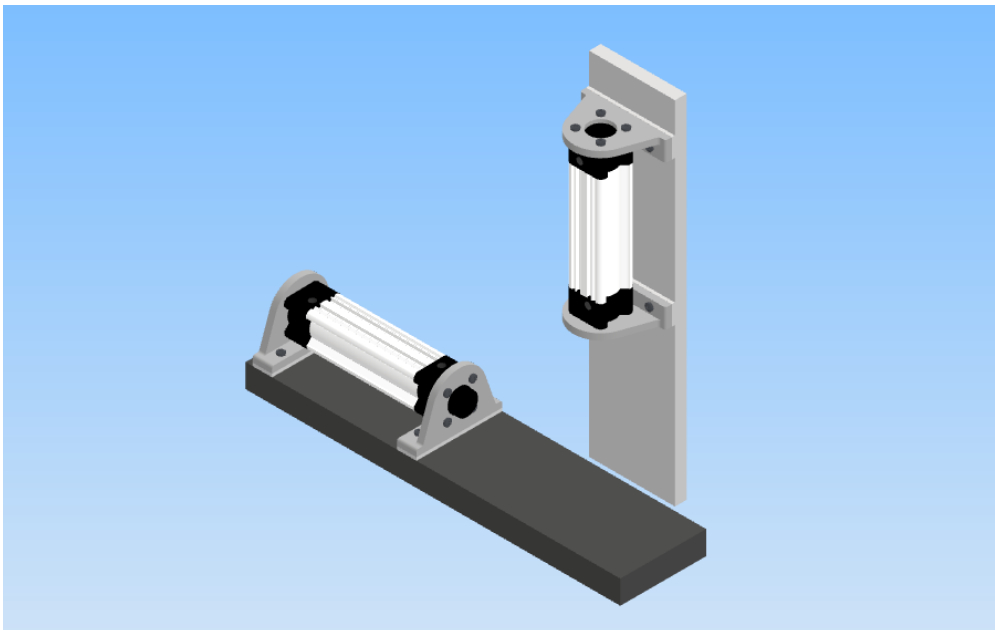


Figura 3.41. Ensamble de los cilindros con sus soportes y con las bases.

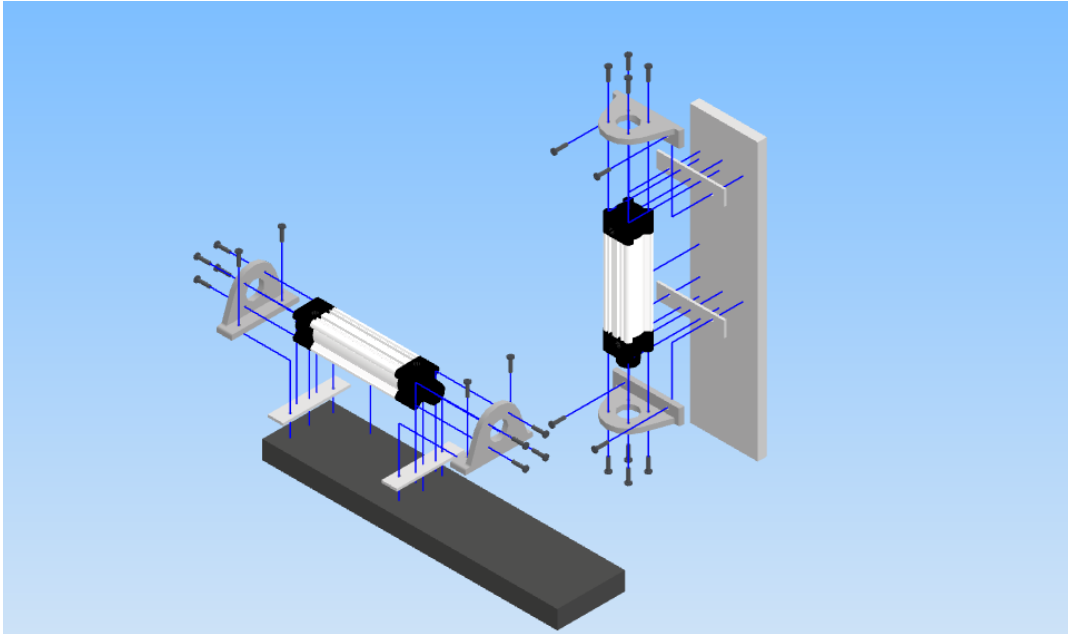


Figura 3.42. Vista explosionada del ensamble mostrado en la figura 3.41.

El ensamble final de la indentadora se muestra en la figura 3.43, donde se ha agregado también una mesa de soporte para el modelo. El plano de la mesa de soporte puede consultarse en el anexo B. Al igual que en el caso de la empacadora el ensamble de la indentadora sirve de referencia para la construcción del ensamble dentro de Unity.

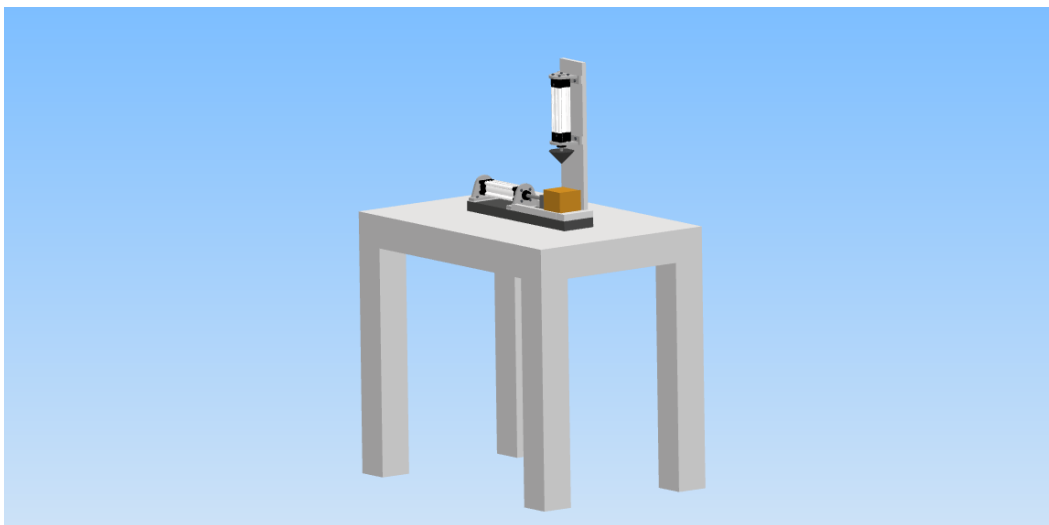


Figura 3.43. Ensamble completo de la indentadora.

Elementos de la estampadora

El diseño de la estampadora se basó en el plano de situación mostrado en la figura 3.9 y en una pieza de trabajo de dimensiones mostradas en el anexo C y en el modelo CAD mostrado en la figura 3.44. El modelo mostrado en la figura 3.44 es un ensamble. Las piezas de dicho ensamble se muestran en la figura 3.45 en una vista explosionada del modelo. Esta pieza se hizo por partes por razones que se harán evidentes en la animación del estampado de la pieza.

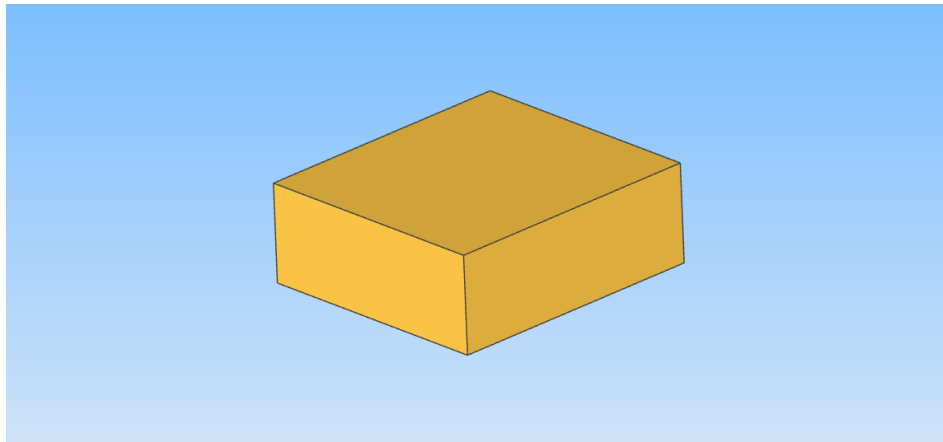


Figura 3.44. Modelo CAD de la pieza de trabajo de la estampadora. Esta pieza es un ensamble de las piezas mostradas en la figura 3.45.

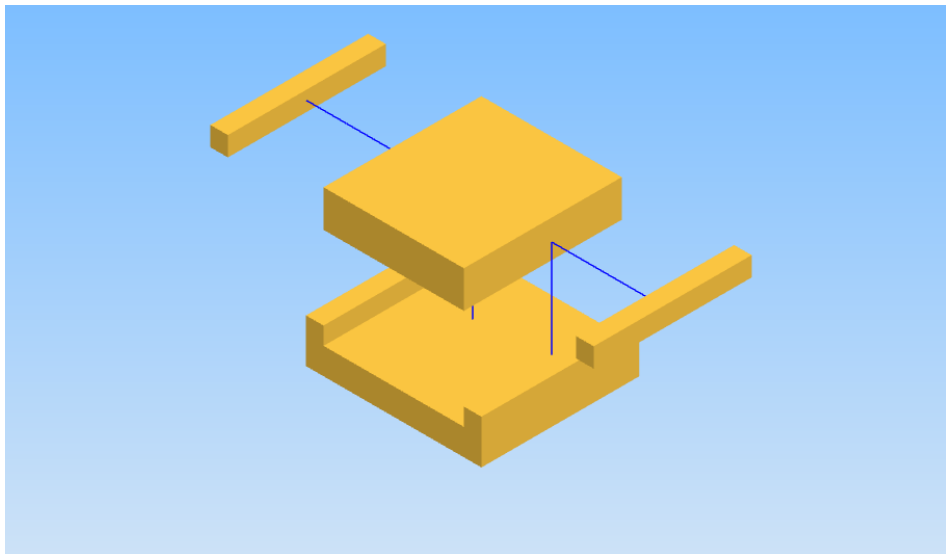


Figura 3.45. Vista explosionada de la pieza mostrada en la figura 3.44. Esta pieza se construyó en partes para hacer una animación donde solo se modifiquen las escalas de cada componente.

En la figura 3.46 se muestra el diseño de la base de la impresora donde se fijan los soportes de los cilindros A y C (figura 3.14), además del soporte del cilindro B que es distinto al de los cilindros A y C (se verá más adelante).

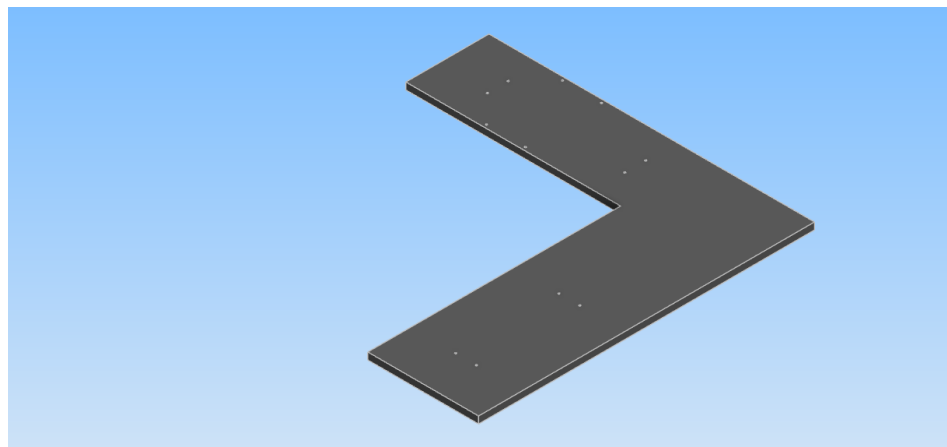


Figura 3.46. Base donde se fijan los soportes de los cilindros A y C, además del elemento de soporte del cilindro B.

La figura 3.47 muestra el ensamblaje del soporte para las piezas que serán impresas. La figura 3.48 muestra una vista explosionada del ensamblaje de la figura 3.47. En adelante nos referiremos a este elemento como el almacén vertical. Como en el caso de la impresora este modelo se realizó en piezas para poder asignarles sus colisionadores (*Box Colliders*) de manera sencilla. No se especificaron elementos mecánicos de fijación.

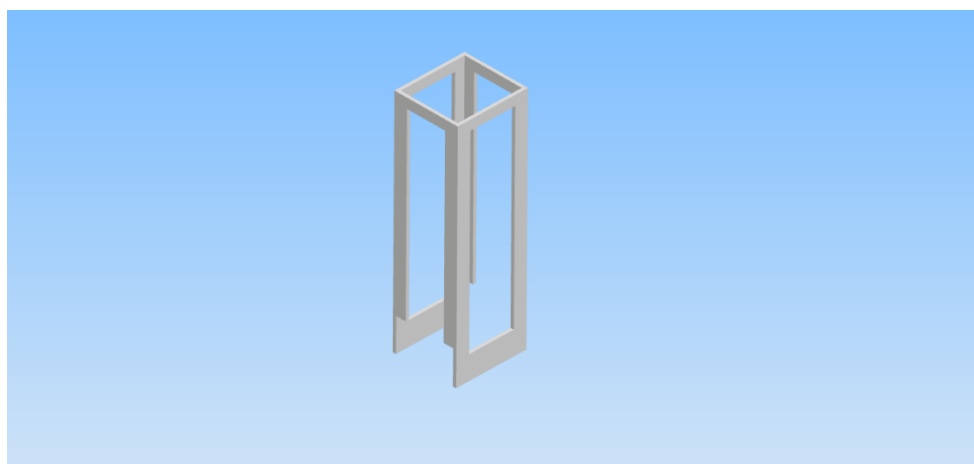


Figura 3.47. Soporte almacenador de piezas de la impresora.

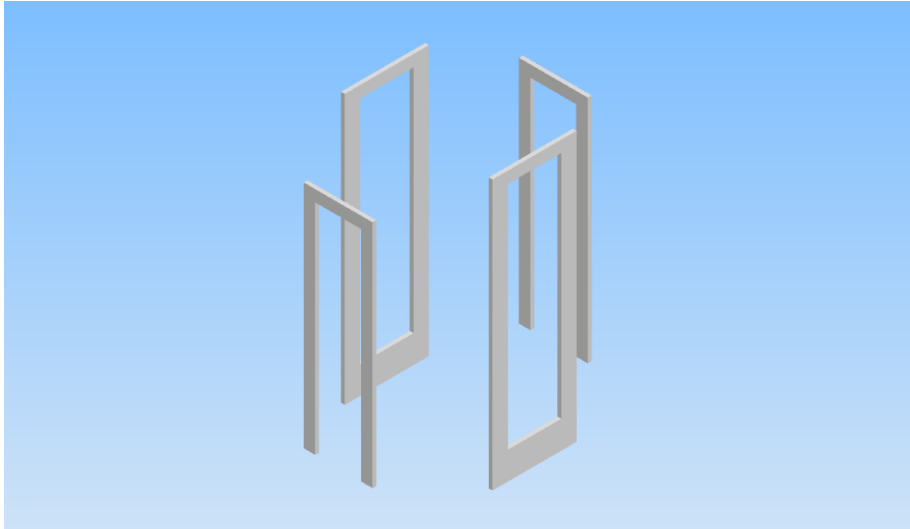


Figura 3.48. Vista explosionada del soporte almacenador de piezas mostrado en la figura 3.47.

Como puede observarse del plano de situación de la estampadora (fig. 3.9), el cilindro que realiza el estampado (cilindro B) tiene un soporte especial. Este soporte, mostrado en la figura 3.49 también fue diseñado siguiendo como base el mostrado en el plano de situación de la estampadora. Este soporte no fue necesario realizarlo en piezas más pequeñas ya que no interactúa directamente con las piezas a estampar, por lo que no es necesario agregarle colisionadores o se pueden agregar algunos sin la necesidad de que estén colocados de forma exacta.

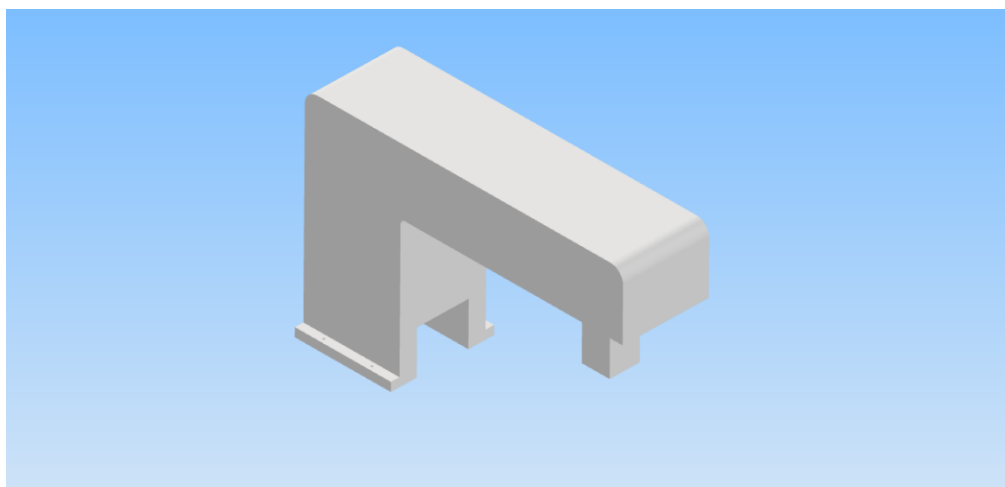


Figura 3.49. Soporte del cilindro que realiza el estampado de las piezas.

En el plano de situación de la estampadora no se especifica, pero fue agregada al modelo CAD de la estampadora, una rampa para el desalojo de las piezas. El ensamble de la rampa y las guías que mantienen la pieza en buena posición se muestra en la figura 3.50. En la figura 3.51 se muestra una vista explosionada del ensamble de la figura 3.50. Como estos elementos interactúan directamente con la pieza estampada requieren colisionadores que se adapten bien a la geometría de la pieza, por eso se realizaron los modelos en piezas individuales.

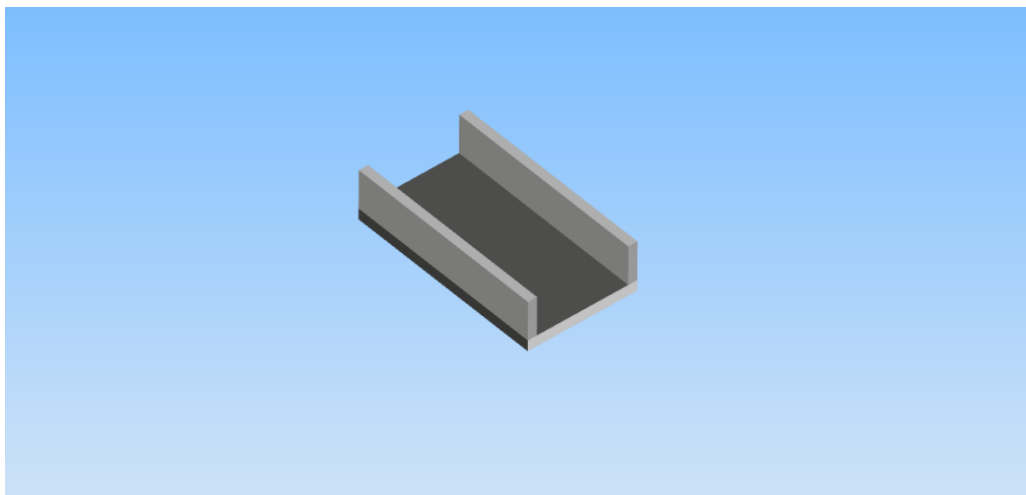


Figura 3.50. Ensamble rampa – Guías de la estampadora.

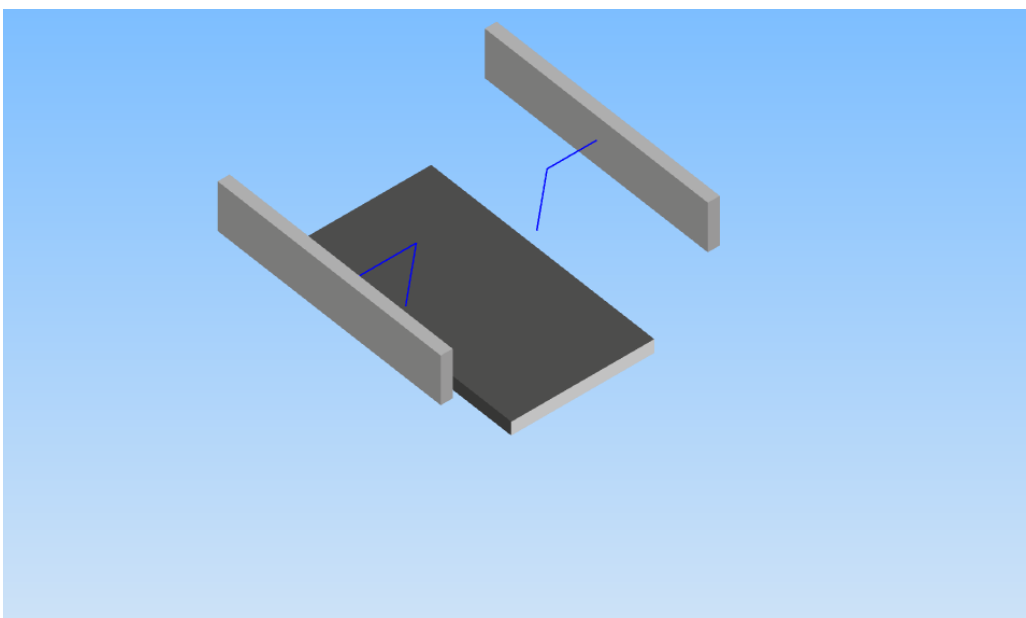


Figura 3.51. Vista explosionada del ensamble de rampa – guías mostrado en la figura 3.50.

Las demás guías que mantienen la posición adecuada de la pieza de trabajo durante el estampado se muestran en la figura 3.52 en ensamble con la base de la estampadora para mostrar la posición relativa de las guías respecto de la base. En la figura 3.53 se muestra una vista explosionada del mismo ensamble. Estas piezas igualmente fueron diseñadas individualmente para asignarles colisionadores que encajaran de manera sencilla.

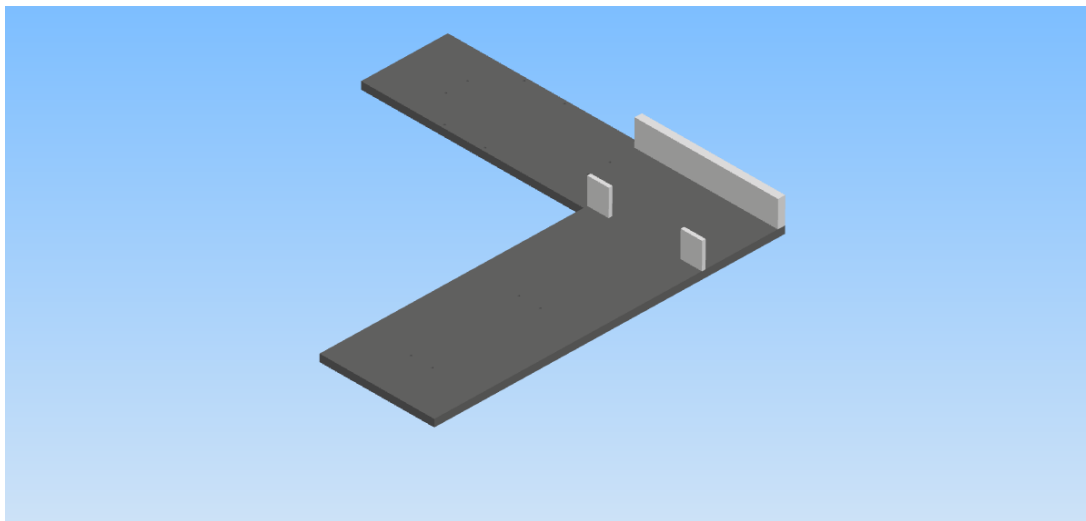


Figura 3.52. Ensamble de la base de la estampadora y las guías que posicionan la pieza a estampar.

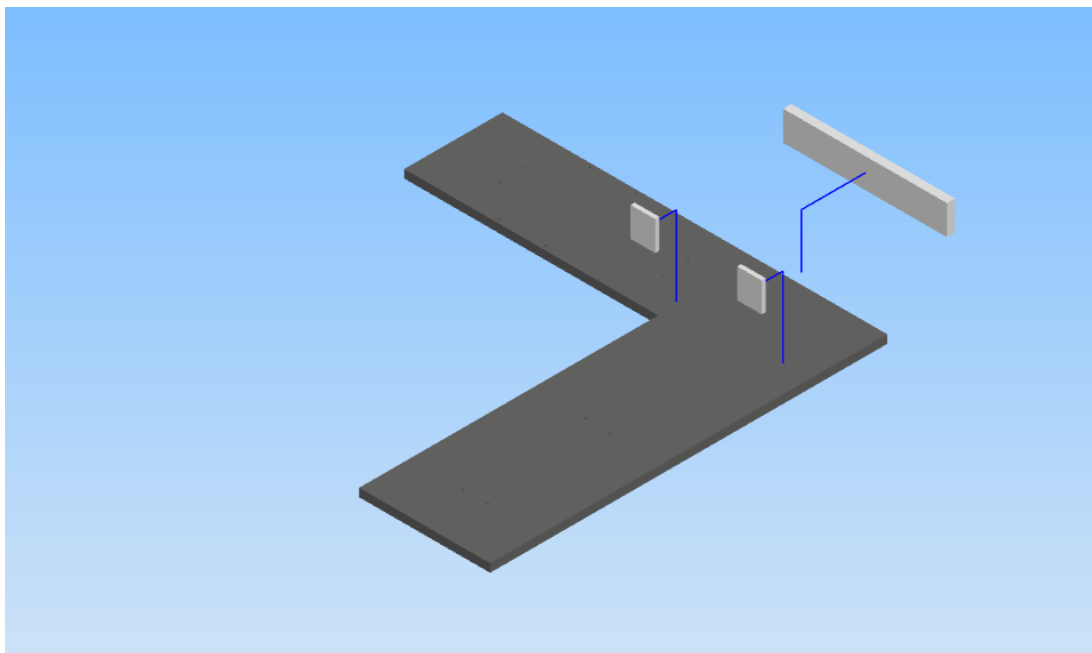


Figura 3.53. Vista explosionada del ensamble mostrado en la figura 3.52.

La figura 3.54 muestra el ensamble de la base de la estampadora con los cilindros A y C y sus respectivos soportes. La figura 3.55 muestra el mismo ensamble en una vista explosionada para poder visualizar todos sus componentes. El cilindro A tiene una carrera de 220 mm y el cilindro C tiene una carrera de 320 mm .

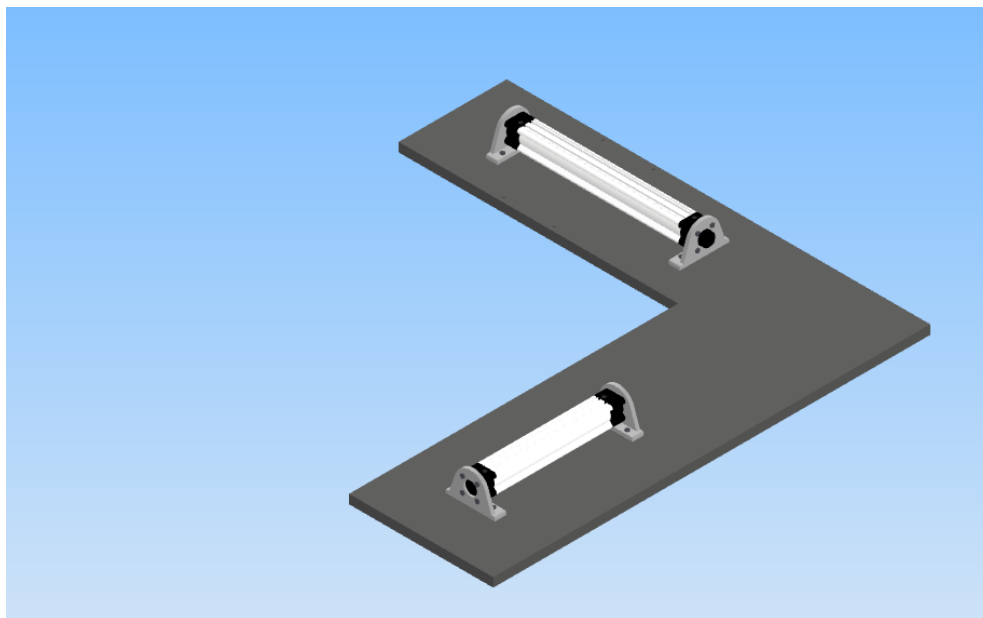


Figura 3.54. Ensamble de la base con los cilindros A y C y sus respectivos soportes.

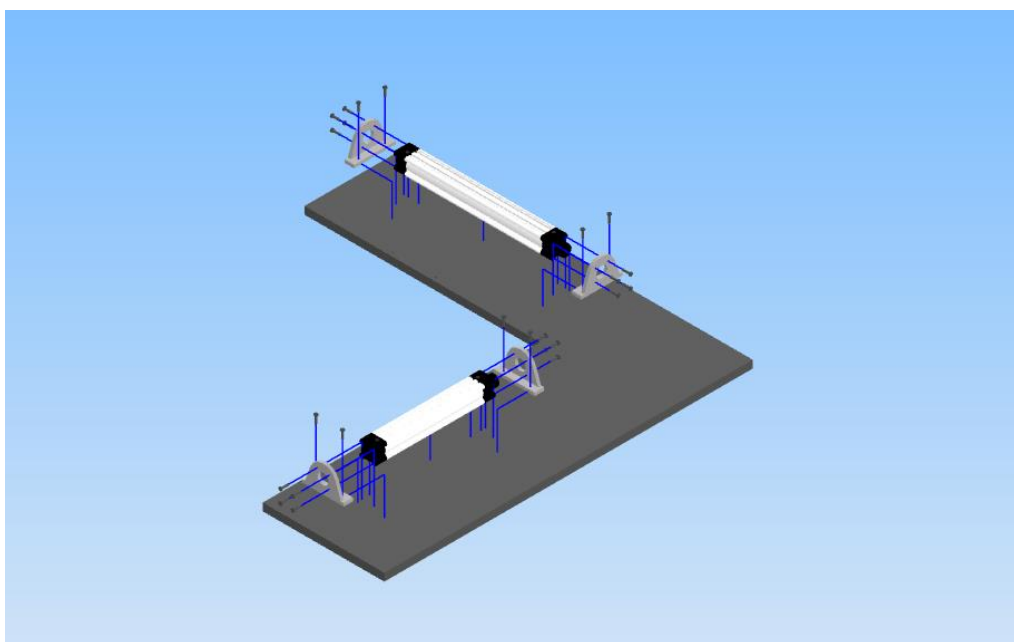


Figura 3.55. Vista explosionada del ensamble mostrado en la figura 3.54.

La figura 3.56 muestra el ensamble del soporte del cilindro B junto con su cilindro. En la figura 3.57 se muestra una vista explosionada del ensamble. Como ya se mencionó este soporte se basó en el plano de situación de la estampadora. Para la parte que sostiene el cilindro no se consideraron elementos adicionales de fijación mecánica (como pernos). El cilindro B tiene una carrera de 100 *mm*.

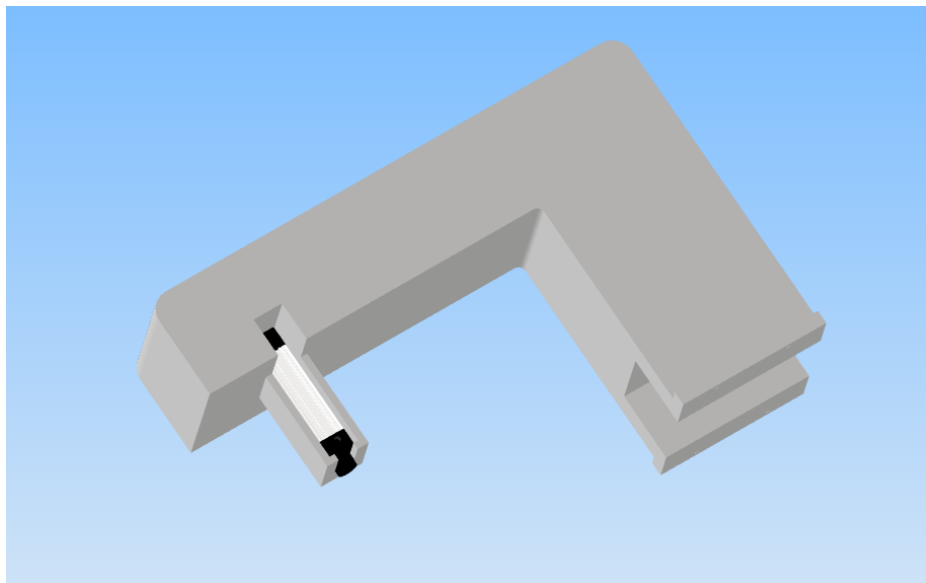


Figura 3.56. Ensamble del cilindro B con su soporte. La posición mostrada del soporte se eligió para dar más claridad sobre cómo encaja el cilindro dentro del soporte.

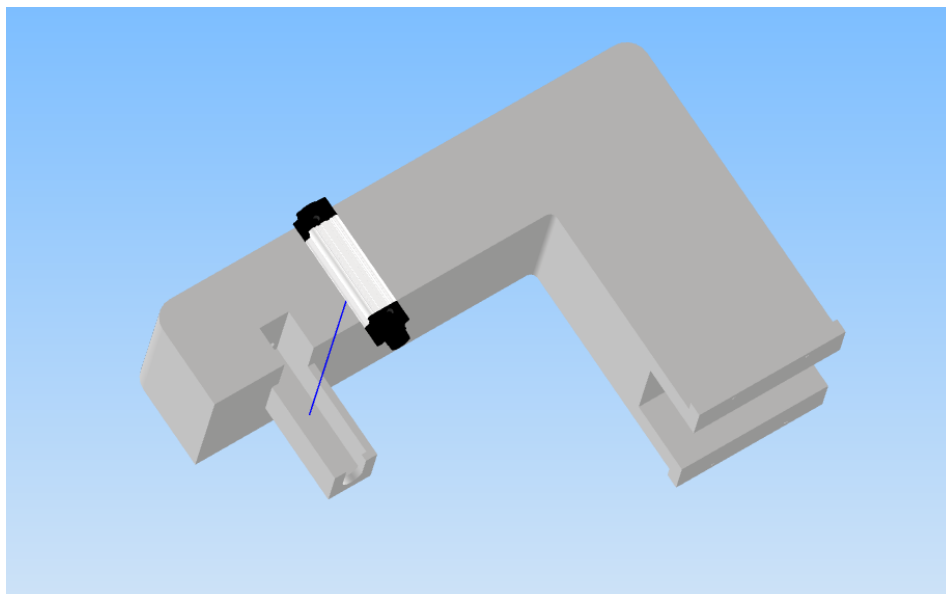


Figura 3.57. Vista explosionada del ensamble mostrado en la figura 3.56.

La figura 3.58 muestra una vista del ensamblaje de la base de la estampadora con el soporte del cilindro B para mostrar la posición relativa del soporte a la base. En la figura 3.59 se muestra la vista explosionada del mismo ensamblaje para poder visualizar los elementos. En este caso se colocaron cuatro pernos para sujetar el soporte a la base. Los pernos considerados son también M5.

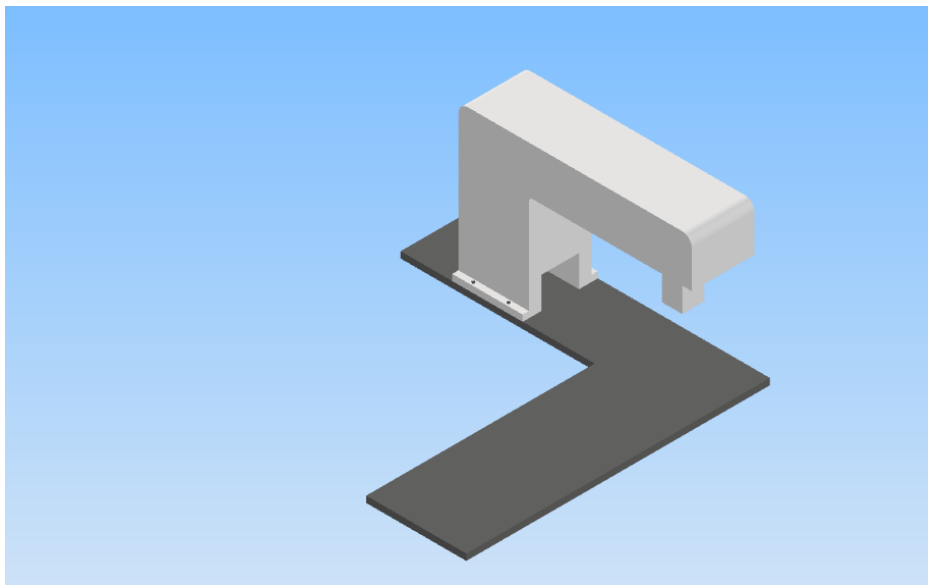


Figura 3.58. Ensamblaje de la base de la estampadora con el soporte del cilindro B.

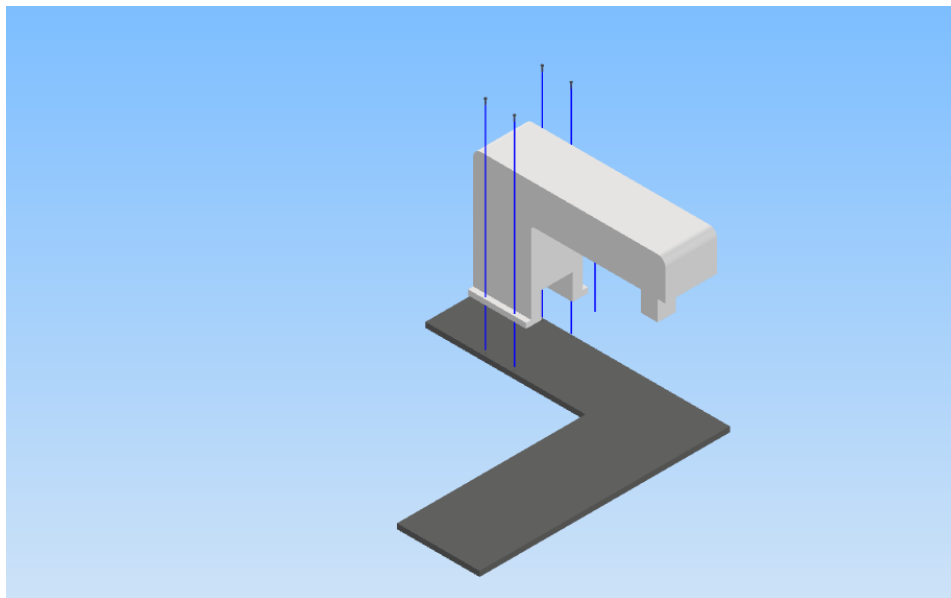


Figura 3.59. Vista explosionada del ensamblaje mostrado en la figura 3.58.

Las figuras 3.60 y 3.61 muestran el ensamble y su vista explosionada del cabezal y el vástago del cilindro A, respectivamente. Al igual que en el caso de la empacadora; el cabezal, además de empujar la pieza de trabajo, tiene la tarea de impedir que las piezas de trabajo restantes caigan a la base mientras el cilindro A está su posición extendida, de ahí el ancho del cabezal.

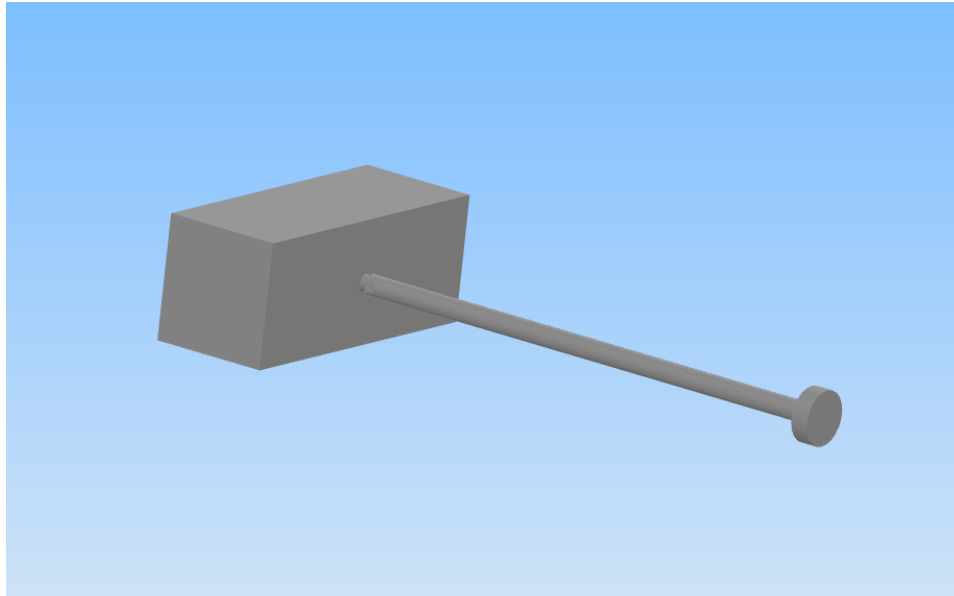


Figura 3.60. Ensamble vástago – cabezal para el cilindro A de la estampadora.

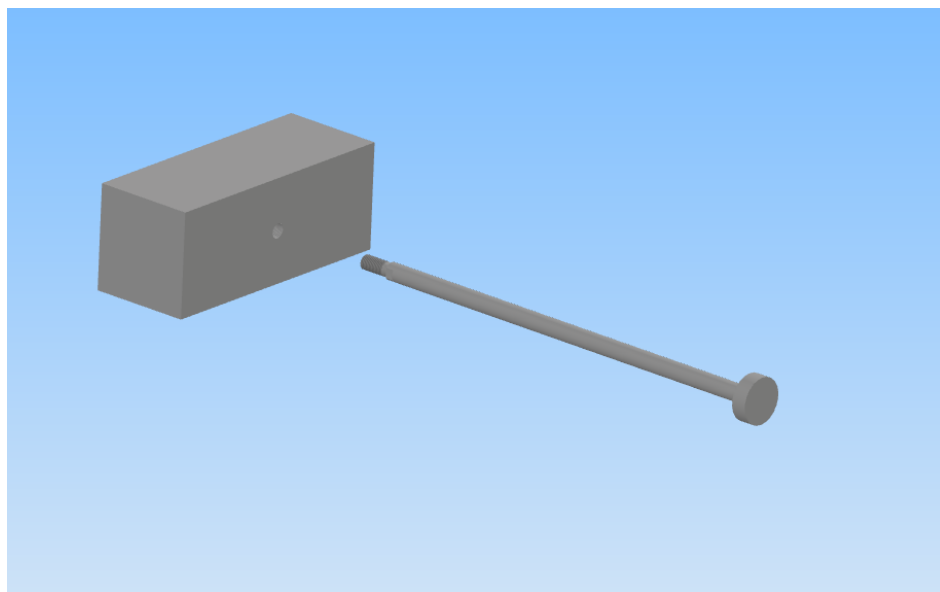


Figura 3.61. Vista explosionada del ensamble mostrado en la figura 3.60.

Las figuras 3.62 y 3.63 muestran el ensamble y su vista explosionada del cabezal y el vástago del cilindro C, respectivamente. Este cilindro tiene la tarea de expulsar la pieza estampada de la estampadora. La forma del cabezal ayuda a movilizar la pieza en una dirección previsible, aunque las guías impiden que la pieza tome otra dirección.

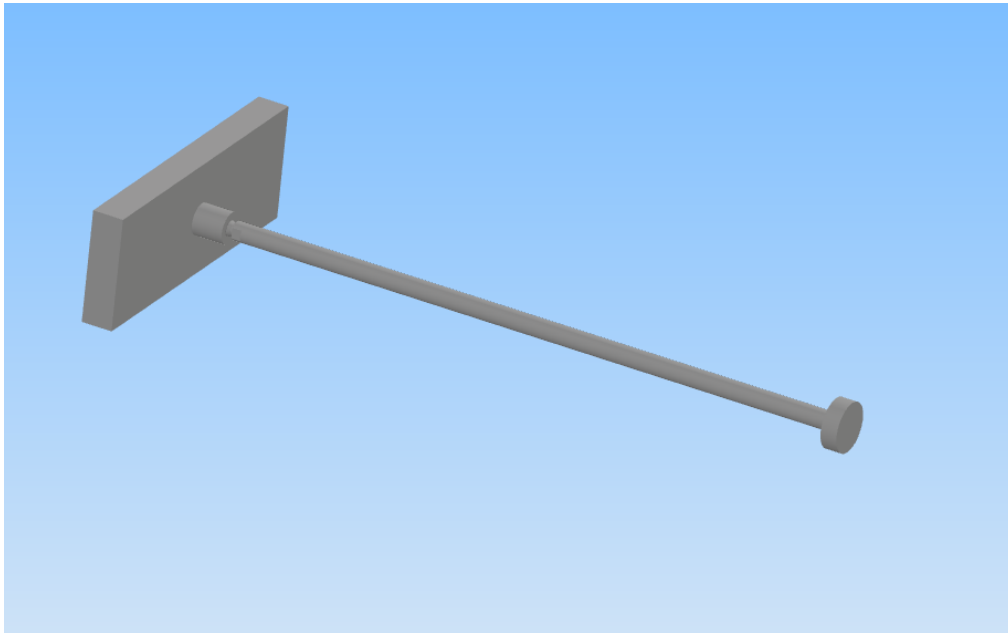


Figura 3.62. Ensamble vástago – cabezal del cilindro C.

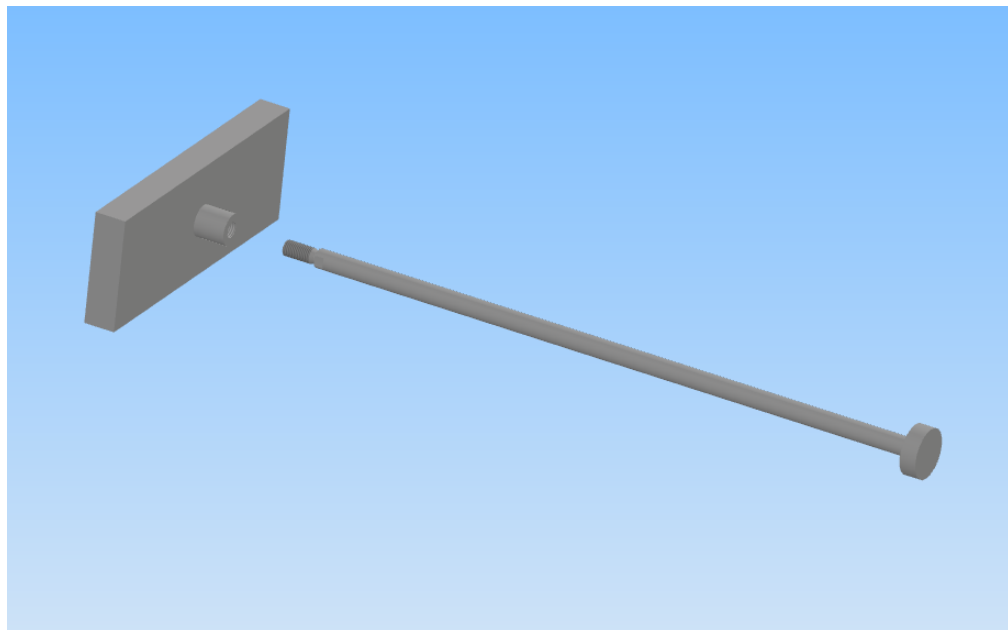


Figura 3.63. Vista explosionada del ensamble mostrado en la figura 3.62.

Por último, las figuras 3.64 y 3.65 muestran el ensamble y su vista explosionada del cabezal y el vástago del cilindro B. Como puede observarse de estas últimas dos figuras el cabezal tiene una forma especial que permite el estampado de la pieza de trabajo. Podemos observar en la figura 3.64 que el cabezal no impediría la deformación de la pieza de trabajo hacia los costados, son los elementos guías los que impiden que la pieza se deforme a lo ancho.

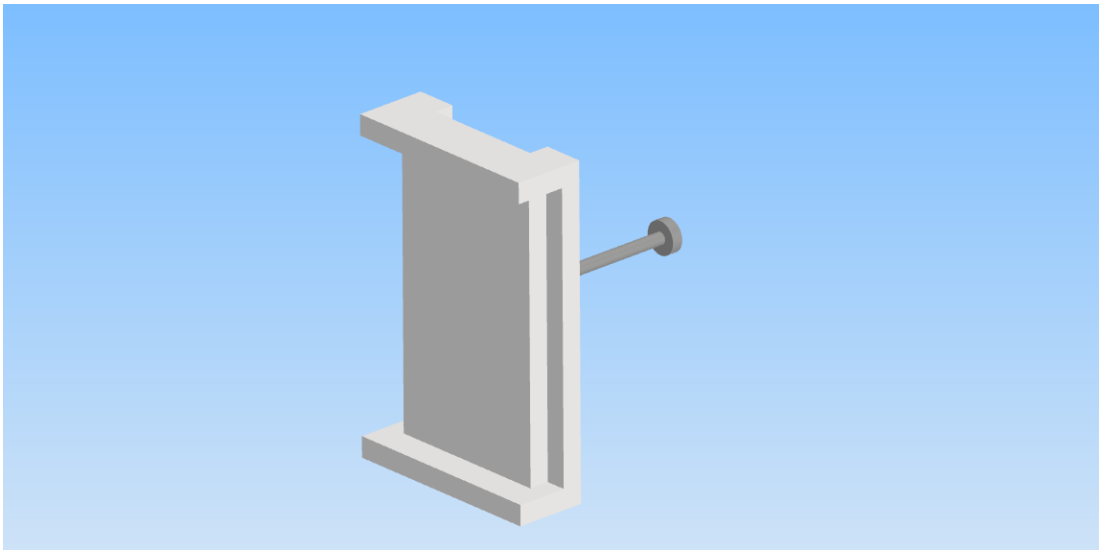


Figura 3.64. Ensamble vástago – cabezal del cilindro B.

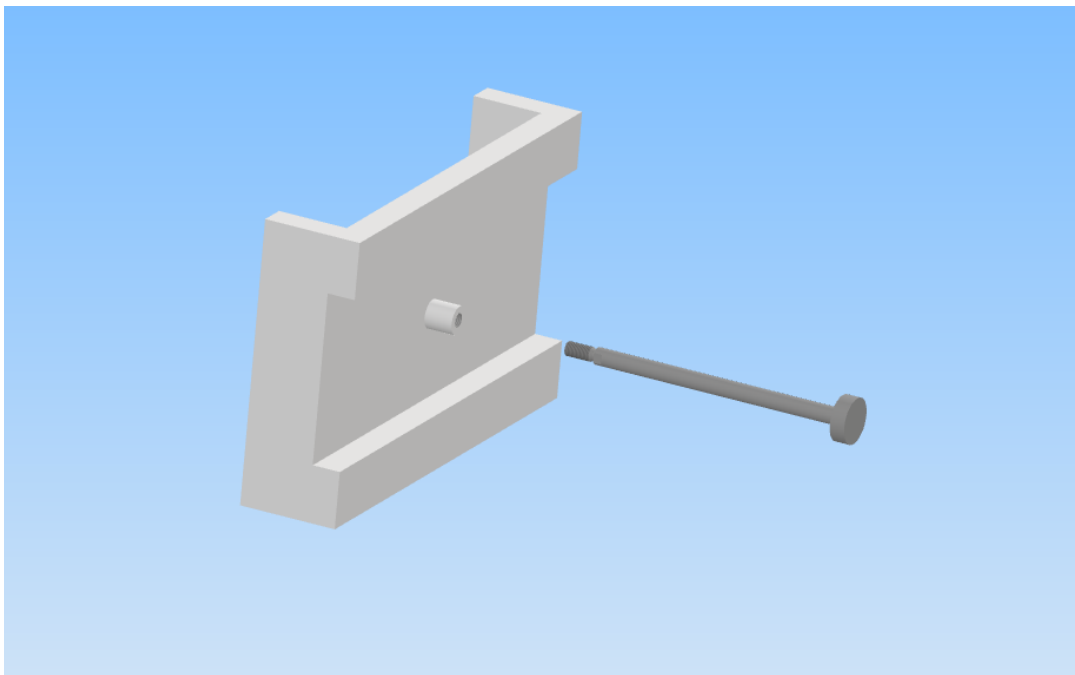
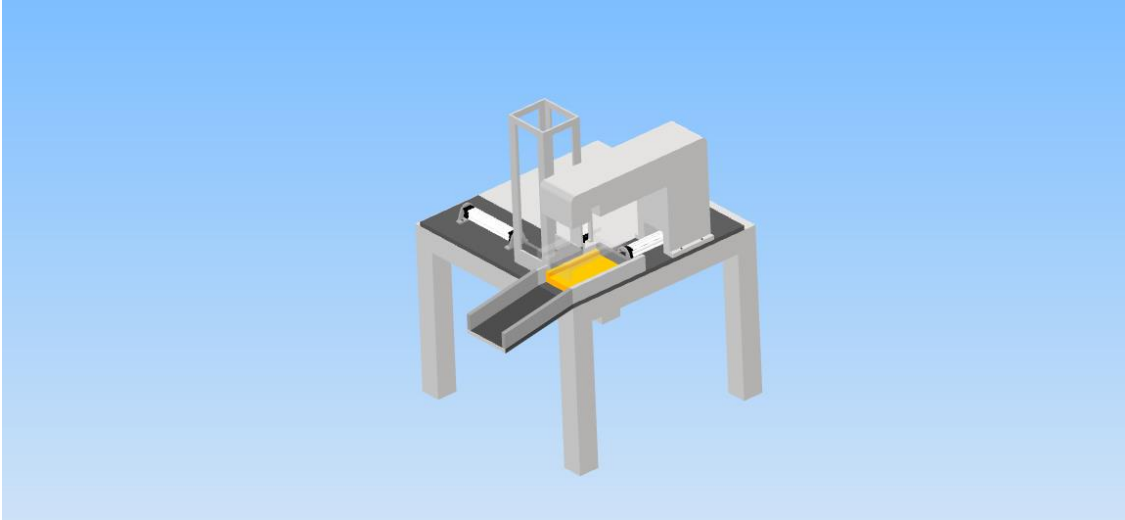


Figura 3.65. Vista explosionada del ensamble mostrado en la figura 3.64.

El ensamble final para la estampadora se muestra en la figura 3.66, donde el cabezal B se ha puesto transparente para poder visualizar de mejor manera los componentes. Este último ensamble sirve de referencia para la construcción del ensamble dentro de Unity.



3.66. Ensamble de la estampadora. El Cabezal B del cilindro B es transparente para poder visualizar de mejor manera el conjunto.

Por último, se diseñó una caja contenedora tanto para las piezas de trabajo de la empacadora como para las de la estampadora en vista de que estas caerían al suelo durante el ciclo de movimiento si no hay algún elemento que lo impida. La figura 3.67 muestra un ensamble de la caja utilizada como el contenedor de ambas piezas.



Figura 3.67. Ensamble del contenedor de las piezas para la empacadora y la estampadora.

La figura 3.68 muestra una vista explosionada de la caja. Las partes fueron diseñadas como se muestra para poder hacer una animación del cerrado de la caja. Esto se verá en la programación de los gemelos digitales.

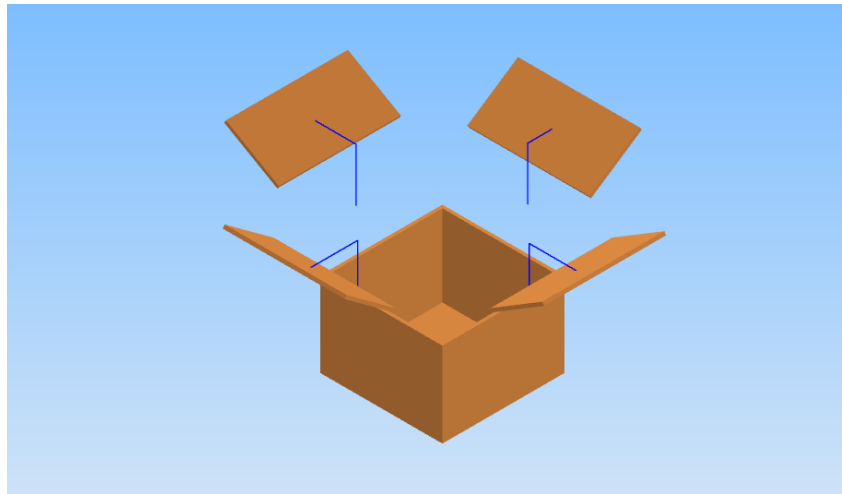


Figura 3.68. Vista explosionada de la caja contenedora de piezas mostrada en la figura 3.67.

3.3. Optimización de los modelos para un menor consumo de recursos computacionales

Varios de los elementos de los gemelos digitales se hicieron particularmente simples para reducir su número de vértices; estos incluyen, por ejemplo, el hecho de que no se especificaron elementos de fijación mecánica para varios elementos de los gemelos digitales, como los soportes de almacenamiento de las piezas de trabajo (en la estampadora y la empacadora) y las guías. Los objetos 3D en Unity® y Blender® están formados por primitivas. Una primitiva es una geometría de tres dimensiones formada por polígonos. Un polígono se refiere a una figura plana cerrada construida por segmentos donde cada segmento está compuesto por vértices [38]. En general la cantidad de vértices que tiene un simulador o videojuego influye directamente en su rendimiento. Es una buena práctica mantener el número de vértices lo más bajo posible y esto ayuda mucho en computadoras que no tengan especificaciones altas [39]. En el desarrollo de los gemelos digitales se intentó mantener el número de vértices los más bajo posible, por lo que fueron requeridos algunos ajustes, sobre todo a los elementos 3D que fueron descargados desde la web.

3.3.1. Simplificación de los modelos CAD

Simplificación de los modelos CAD de los cilindros neumáticos

Se mencionó que los modelos CAD de los cilindros neumáticos fueron descargados de la referencia [36], por lo que estos modelos poseen varios detalles que añaden una gran cantidad de polígonos adicionales que la computadora tendría que procesar aumentando en cierta medida los recursos computacionales de la implementación. De modo que se procedió a simplificar algunos de los detalles que no eran necesarios o incluso que no eran perceptibles para los usuarios. También se mencionó que se utilizó el mismo tipo de cilindro para todos los gemelos digitales variando únicamente la carrera de los mismos, por lo que las simplificaciones realizadas a continuación para un cilindro se replicaron para los demás. La figura 3.69 muestra un corte transversal de uno de los cilindros utilizados. Como puede observarse este cilindro posee una cavidad interna que es por donde ingresa el aire que empuja a los vástagos. Esta cavidad no es requerida ya que los usuarios no la verán, por lo que puede eliminarse. Los orificios superiores por donde ingresa el aire al cilindro, aunque son visibles, tampoco son requeridas por lo que también se eliminan.

La figura 3.70 muestra una vista frontal del cilindro neumático. En esta vista podemos observar los orificios para la fijación mediante los pernos. Estos orificios también están presentes en la parte trasera del modelo. Los soportes de los cilindros (figura 3.14) cubren por completo estos orificios, de modo que tampoco son requeridos y se eliminan.

La figura 3.71 muestra un acercamiento a la parte superior del cilindro neumático, para poder observar que en esta parte se tiene una geometría elaborada. Estos detalles son difíciles de ver desde el simulador además de que no son necesarios de modo que también se modifica la geometría.

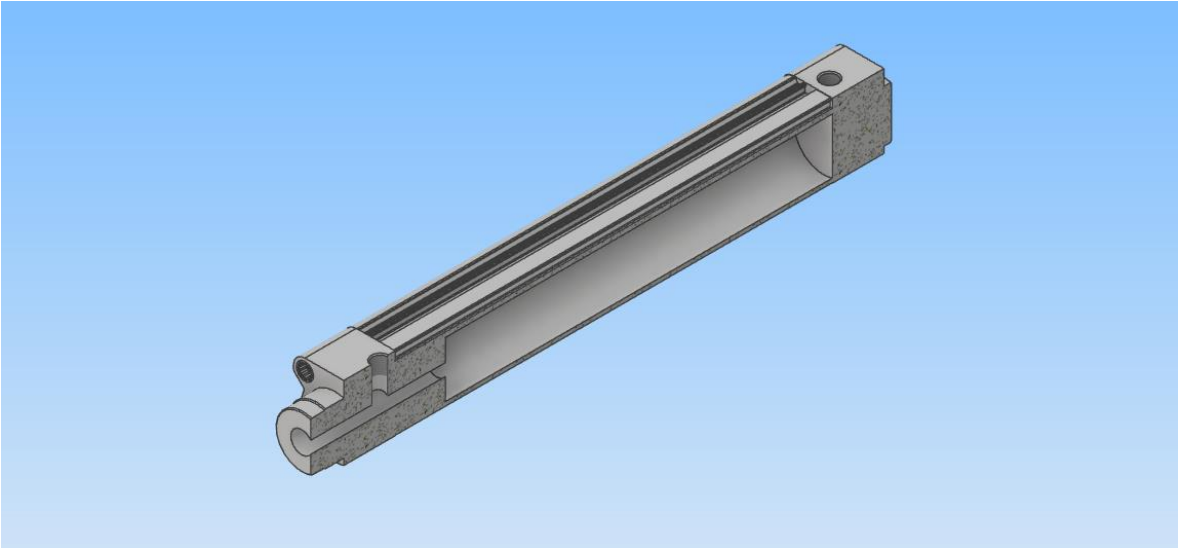


Figura 3.69. Corte transversal a un modelo CAD de un cilindro neumático.

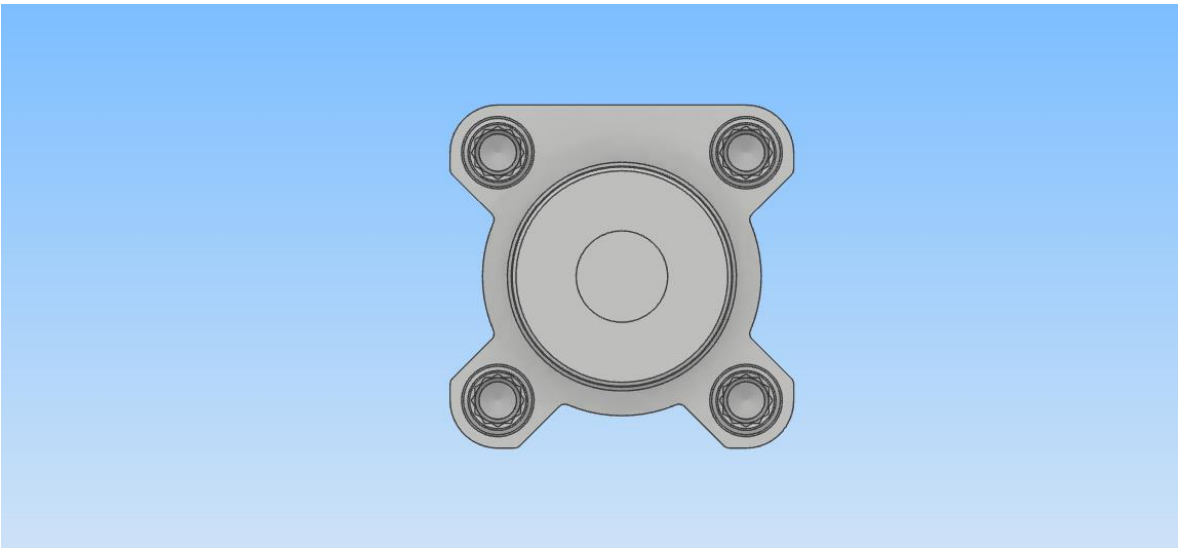


Figura 3.70. Vista Frontal de los cilindros neumáticos donde se observan los 4 barrenos para la fijación por pernos.

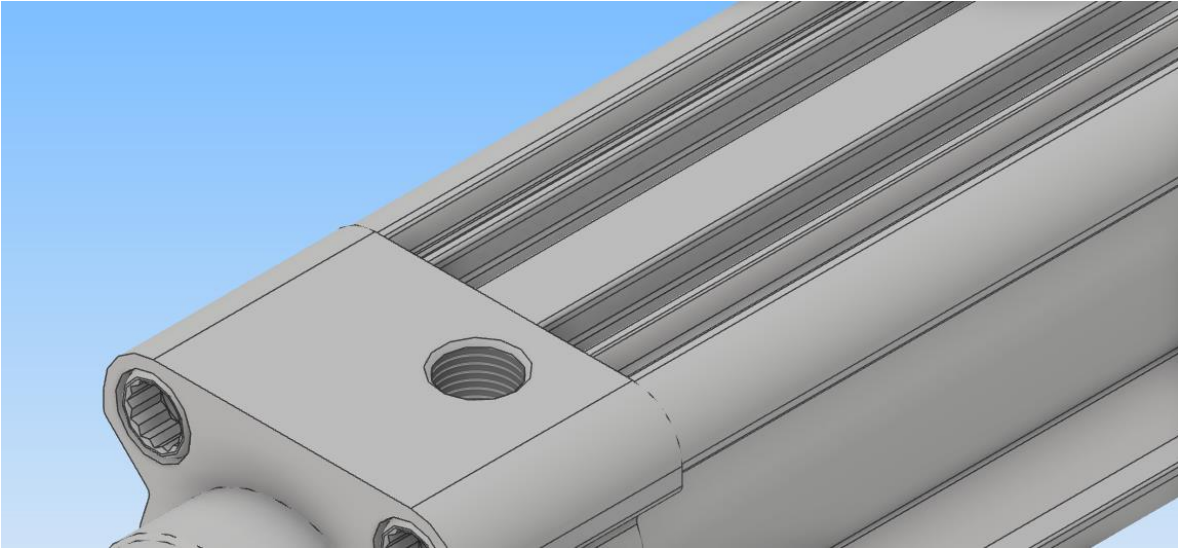


Figura 3.71. Acercamiento a la parte superior del cilindro neumático.

La figura 3.72 muestra el resultado de las simplificaciones hechas al cilindro. Mientras que la figura 3.73 muestra un corte transversal del modelo simplificado para observar cómo ha quedado un modelo sólido.

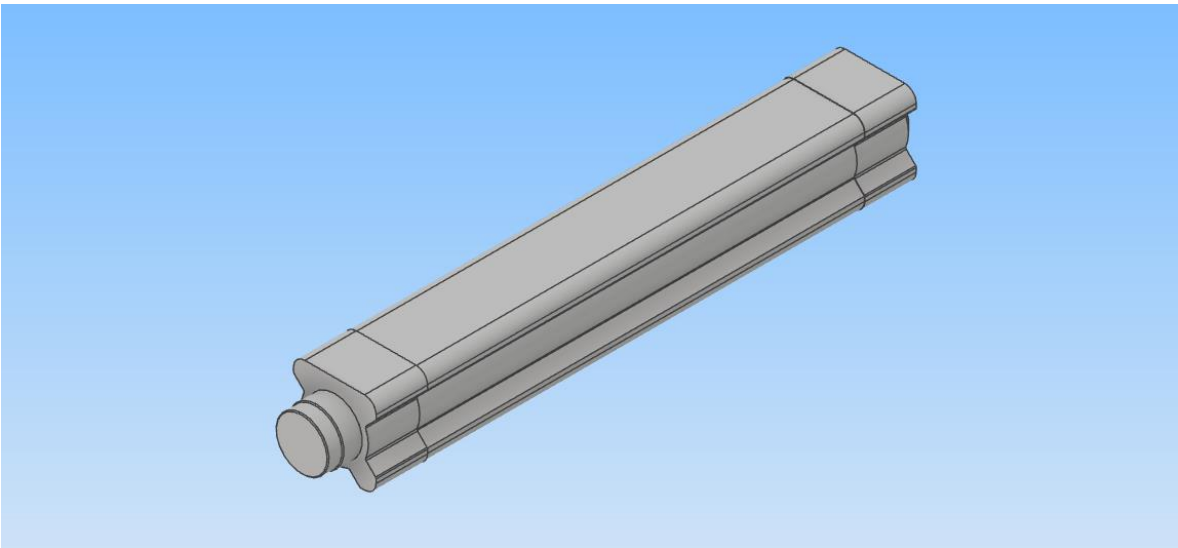


Figura 3.72. Cilindro obtenido después de realizadas las simplificaciones.

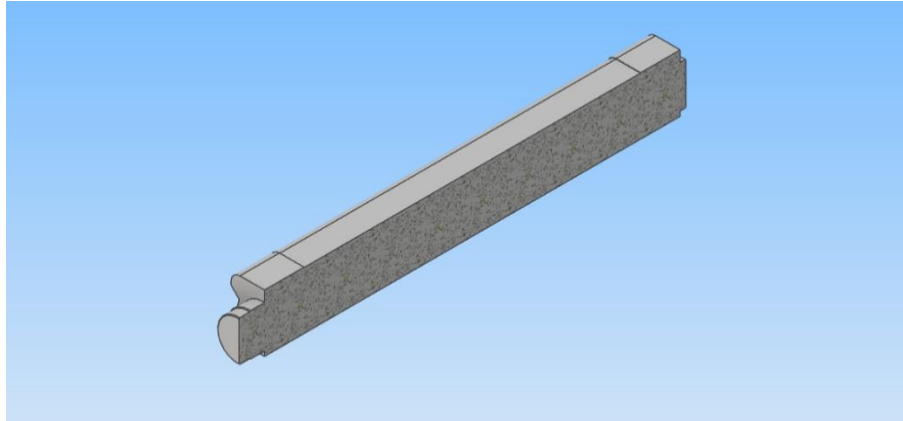


Figura 3.73. Corte Transversal del cilindro neumático donde se puede observar que ha quedado un cuerpo sólido.

Simplificación de los modelos CAD de los Vástagos

También se simplificaron los modelos CAD de los vástagos. En este caso se eliminaron las partes que no serían visibles a los usuarios, como las roscas el émbolo. También se eliminó parte del vástago que no es visible cuando el vástago se extiende. En la figura 3.74 se muestra de nueva cuenta el vástago de los cilindros. La rosca y el émbolo fueron eliminados en todos los casos, pero para eliminar la parte del vástago que no es visible una vez que el cilindro está en su posición extendida; se realizaron mediciones, desde el editor *3D model* de Inventor®, de la parte visible del vástago en su posición extendida. Ya que se utilizaron diversas carreras en el desarrollo de los gemelos digitales, las partes eliminadas no fueron iguales en todos los casos. La figura 3.75 muestra el resultado de la simplificación realizada para el vástago de un cilindro de carrera de 220 *mm*.

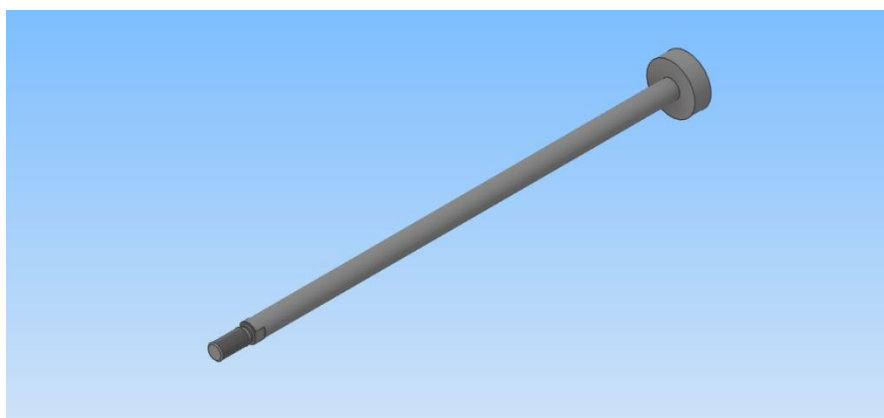


Figura 3.74. Modelo CAD del vástago utilizado en los cilindros neumáticos.

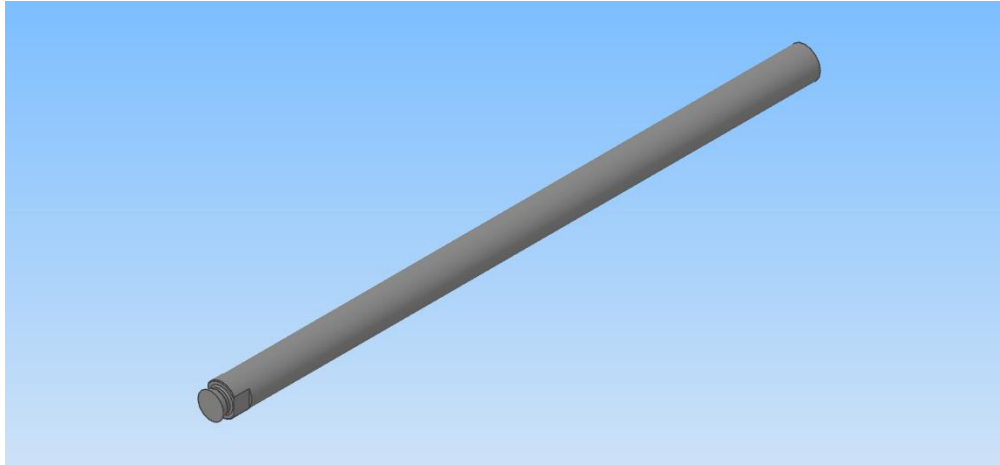


Figura 3.75. Vástago simplificado. Se quitó la rosca el émbolo y la parte no visible del vástago cuando el cilindro está en su posición extendida.

Simplificación de los Pernos

Aunque se consideró hacer varios componentes sin elementos de sujeción mecánica, se prefirió dejar los pernos de fijación que ya se habían puesto en los ensambles. Sin embargo, no fue posible modificar los pernos descargados, por lo que, se diseñó un nuevo perno simplificado el cual se muestra en la figura 3.76. En esta figura se muestra una vista lateral para poder visualizar de mejor manera su geometría. Como puede observarse el perno no tiene vástago y a pesar de la vista también es posible distinguir que no tiene ranuras, es un simple modelo sólido.

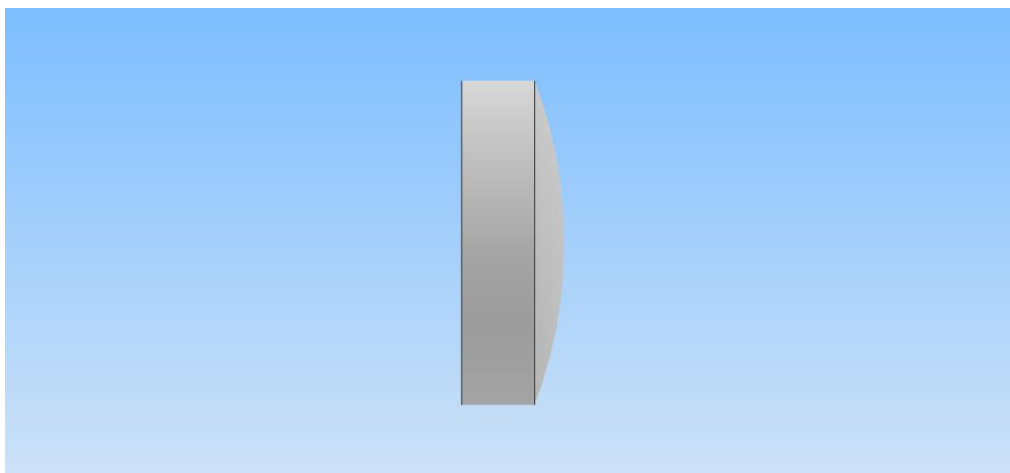


Figura 3.76. Modelo de perno simplificado para su utilización en los gemelos digitales.

Otros elementos simplificados

Durante el diseño de los modelos CAD se diseñaron los orificios donde se fijan los pernos. Estos orificios están presentes en las bases y en los soportes de los cilindros. Estos elementos también fueron eliminados ya que no están a la vista de los usuarios y solo agregan polígonos adicionales a la implementación. Los orificios a los cuales se hace mención pueden ser apreciados en las figuras 3.17 (base empacadora), 3.31 (base horizontal indentadora), 3.32 (base vertical indentadora), 3.35 y 3.36 (elevadores de la indentadora), 3.46 (base de la estampadora). También se eliminaron los orificios de los soportes de los cilindros neumáticos (ver figura 3.14 y 3.59). En la figura 3.77 se puede observar uno de los soportes de los cilindros neumáticos simplificado (sin los orificios para los pernos).

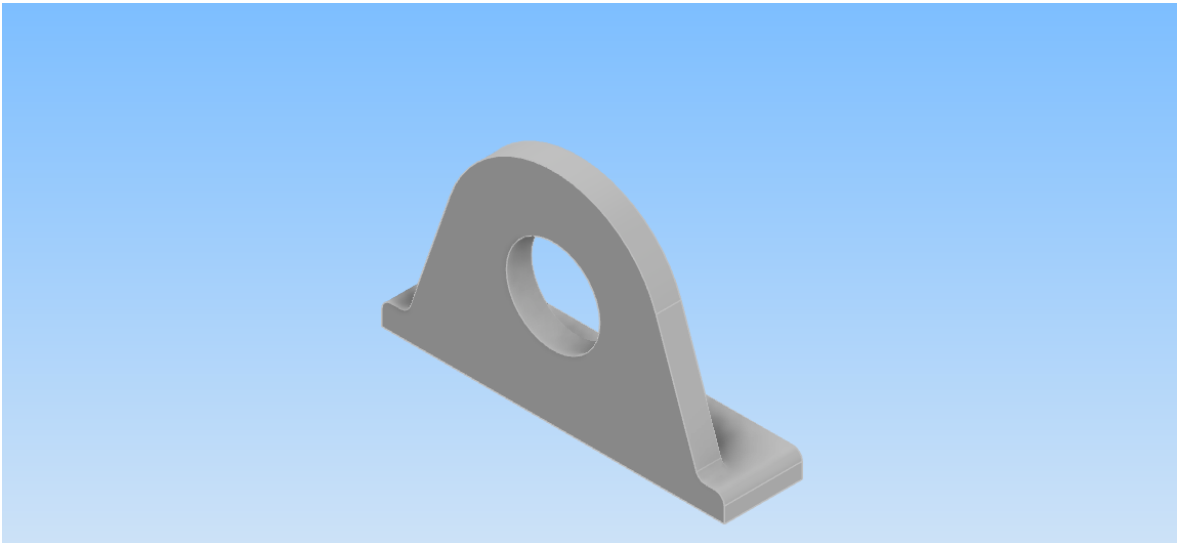


Figura 3.77. Soporte de los cilindros neumáticos sin orificios para los pernos.

Las simplificaciones realizadas no cambiaron el acomodo de los ensambles, por lo que los resultados son similares a los ensambles mostrados anteriormente. Para los soportes simplificados, se prefirió exportar los modelos en ensamble junto con sus pernos (como un modelo unificado), como se muestra en la figura 3.78. La figura 3.79. muestra una vista explosionada del ensamble de la figura 3.78, para mostrar que la disposición no cambió respecto a los ensambles mostrados anteriormente.

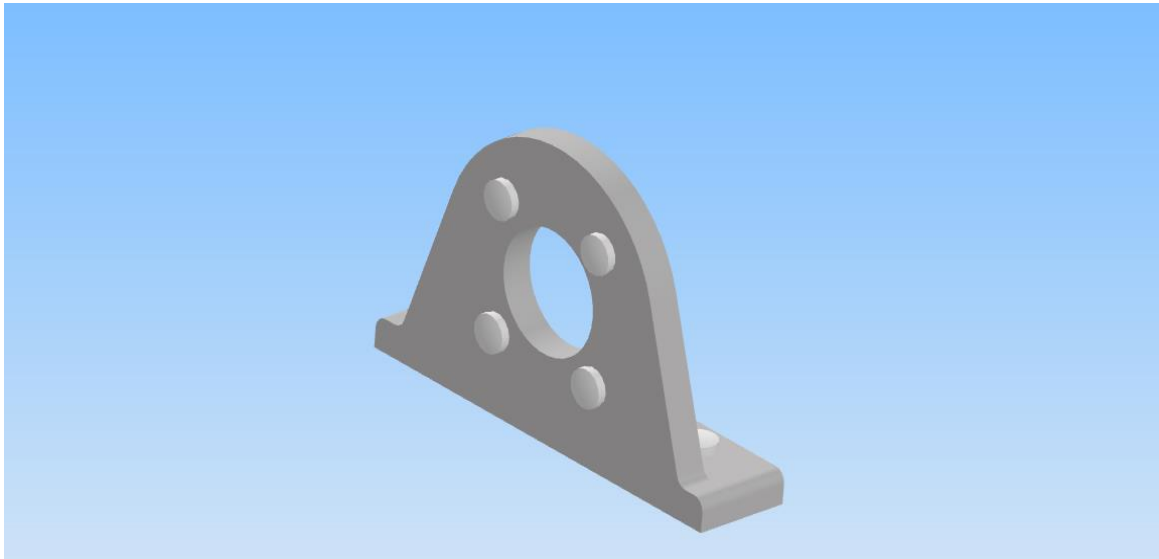


Figura 3.78. Ensamble del soporte – pernos ambos simplificados.

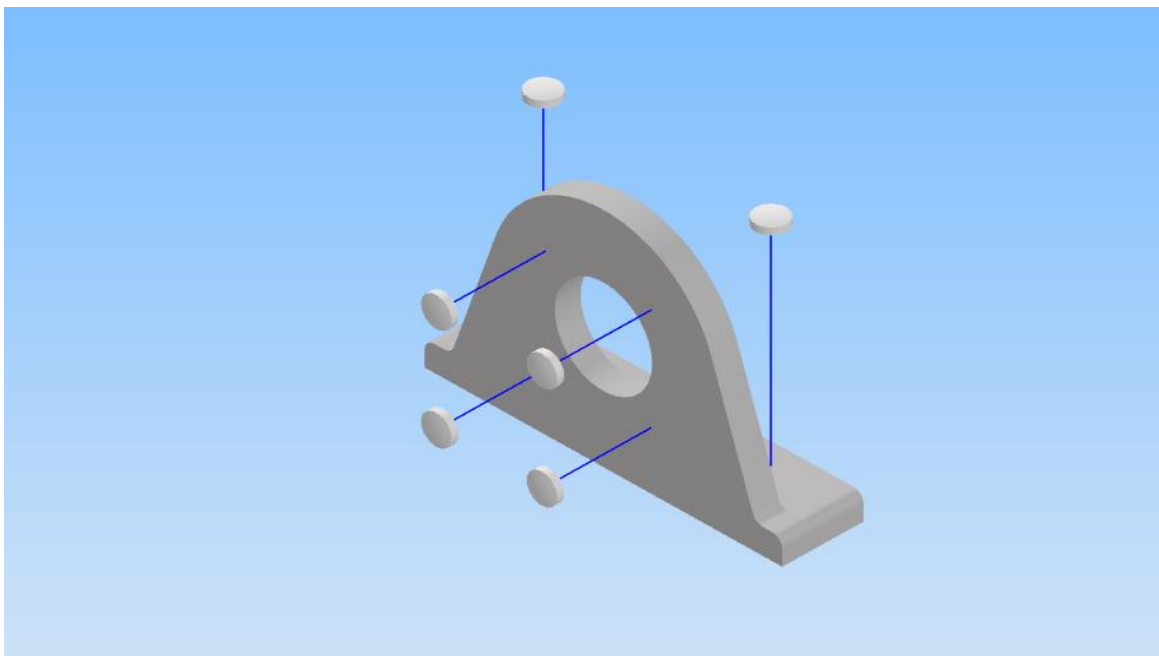


Figura 3.79. Vista explosionada del ensamble de la figura 3.78.

El soporte del cilindro B de la estampadora (que solo se utiliza una vez) también se exportó junto con sus pernos (en un modelo unificado) de manera similar, después de haber quitado los orificios de los pernos y haberlos sustituido por sus versiones simplificadas. La figura 3.80 muestra este ensamble y la figura 3.81 hace un acercamiento a la parte donde están los pernos, ya que debido al tamaño de la pieza es difícil distinguirlos en la figura 3.80.

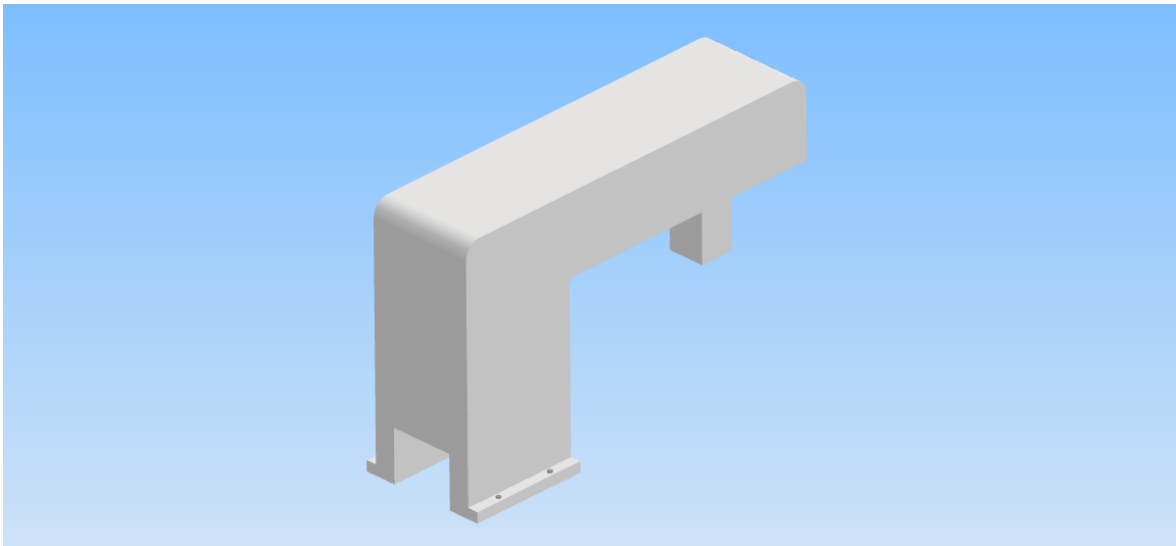


Figura 3.80. Ensamble soporte – pernos para la fijación del cilindro B de la estampadora.

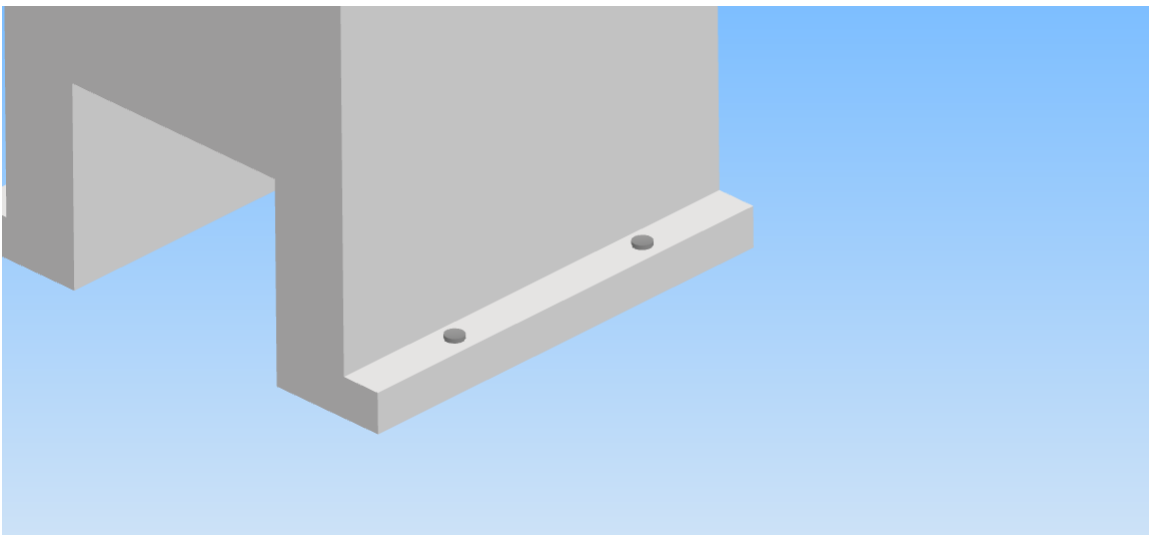


Figura 3.81. Acercamiento al ensamble de la figura 3.80 para poder apreciar de mejor manera los pernos simplificados.

La figura 3.82 muestra una vista explosionada del ensamble de la figura 3.80 para mostrar la posición relativa de los pernos respecto del soporte.

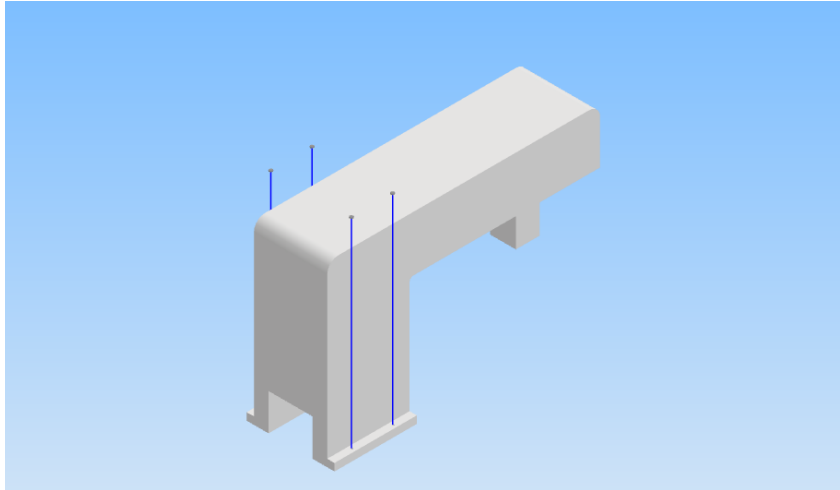


Figura 3.82. Vista explosionada del ensamble mostrado en la figura 3.80.

También se exportaron en ensamble los vástagos con sus respectivos cabezales. En la figura 3.83 se muestra el ensamble vástago – cabezal del cilindro B de la empaedora. En esta figura se pueden observar las simplificaciones que se realizaron anteriormente en el vástago. En la figura 3.84 se observa una vista explosionada del ensamble mostrado en la figura 3.83 donde se puede observar; además, que se eliminó el barreno del cabezal donde se inserta el vástago. Todos los barrenos de los cabezales fueron eliminados como en este ejemplo.

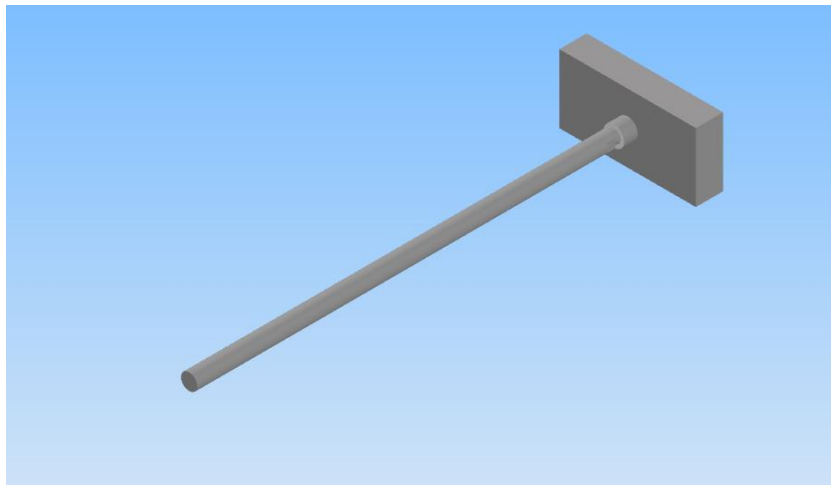


Figura 3.83. Ensamble vástago - cabezal para el cilindro B de la empaedora.

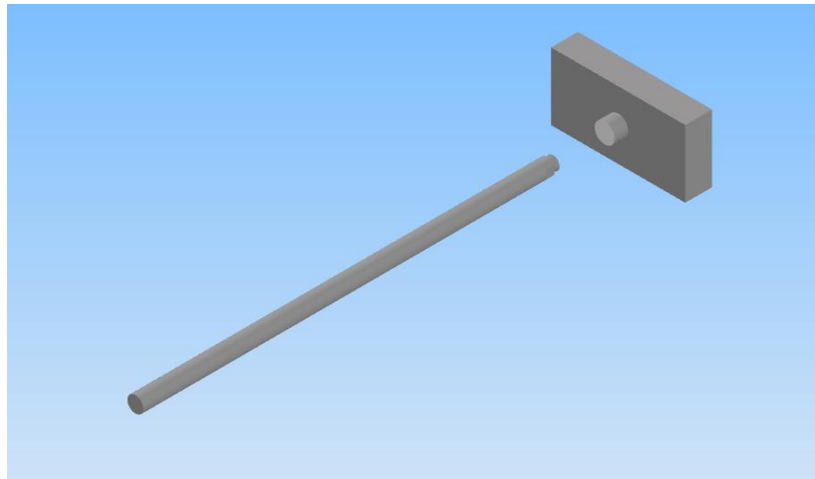


Figura 3.84. Vista explosionada del ensamble de la figura 3.83. Se observa que se eliminó el barreno del cabezal.

3.4. Adición de texturas a los gemelos digitales

Una vez obtenidos los modelos de las piezas y ensambles definitivos fue momento de agregarles colores y texturas. Para algunos modelos se recurrió a Blender® y para otros se trabajó directamente en Unity®. Sin embargo, todas las piezas tuvieron que ser importadas primero a Blender, aunque se añadieran texturas directamente en Unity. La razón es que se requerían importar los modelos a Unity con la extensión FBX, pero esto no podía hacerse desde inventor, por lo que fue necesario exportar los modelos primero a Blender con la extensión STL y una vez en Blender exportarlos con la extensión correcta.

Colores y Texturas de los Cilindros Neumáticos

La figura 3.85 muestra un cilindro neumático exportado a Blender para agregarle colores y texturas. Como puede observarse, a pesar de que se agregaron colores a los modelos en inventor®, estos no se conservan cuando son exportados a Blender.

Primero se crearon dos materiales distintos para agregar a los cilindros. Uno para la parte central y otro para sus partes externas. Al material que se utilizó en la parte interna de los cilindros se le asignó un color blanco, mientras que para las partes externas se utilizó un color negro. Para ambos materiales se asignó la propiedad metálico en 0.8 y la propiedad de rugosidad en 0.2, esto porque asignar 1 en la propiedad metálico provocaba que los cilindros fueran demasiado brillantes. Asignar 0.2 a la rugosidad permitió que los cilindros se tornaran un poco opacos. La figura 3.86 muestra al modelo del cilindro en el modo de edición lo que permite seleccionar manualmente las caras (polígonos) a los que se quieren

asignar los materiales. De este modo se seleccionaron las caras centrales para asignarles el color blanco y las caras externas para asignarles el material negro. La figura 3.87 muestra el resultado de agregar estos materiales al cilindro de ejemplo. Para todos los cilindros de los gemelos digitales fueron utilizados materiales con las mismas características.



Figura 3.85. Modelo 3D exportado a Blender desde Inventor®.

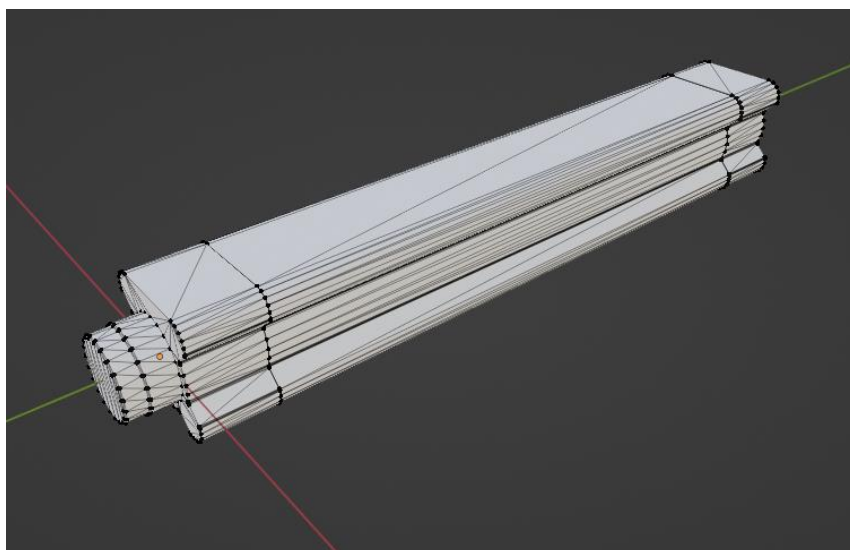


Figura 3.86. Modelo 3D en el modo edición de Blender®.



Figura 3.87. Se asignaron dos materiales distintos a los cilindros: negro y blanco, ambos con las mismas propiedades (0.8 metálico y 0.2 rugosidad).

Colores y Texturas de los Ensamblados de Vástago – Cabezal

Todos los ensamblados de vástago – cabezal fueron trabajados de manera similar a los cilindros. Se crearon dos materiales con sus propiedades metálico y rugosidad establecidas en 0.8 y 0.2, respectivamente. Para dar un poco de contraste a los ensamblados se establecieron dos colores, un gris oscuro para los cabezales y un gris claro para los vástagos. A todos los ensamblados vástago – cabezal restantes se les asignaron materiales de manera similar. La figura 3.88 muestra el resultado de agregar estos materiales a uno de los ensamblados de vástago – cabezal.

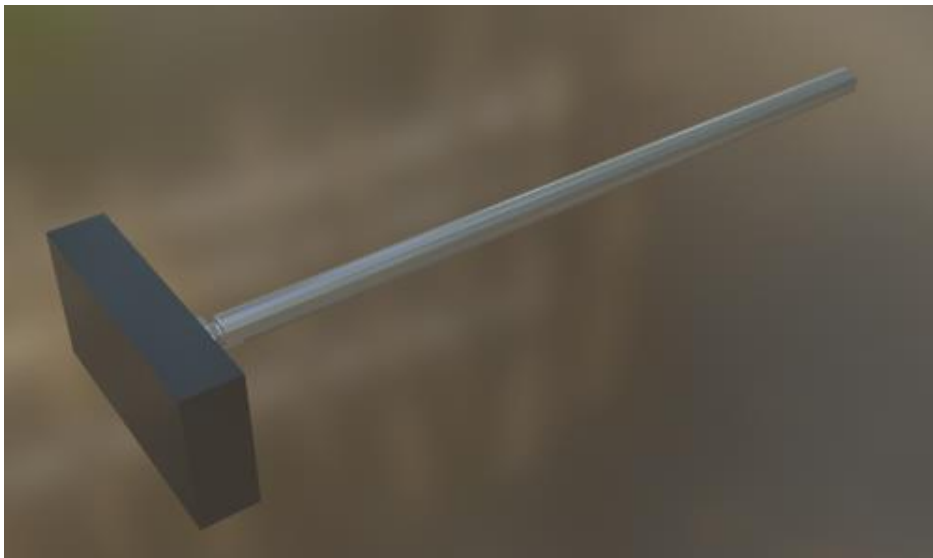


Figura 3.88. Se asignaron dos materiales distintos a los ensamblados de vástago – cabezal de los cilindros neumáticos, uno para el cabezal y otro para el vástago.

Texturas de los ensambles de soporte – pernos

Hay dos modelos de soporte el primero es el mostrado en la figura 3.78 y el segundo es el mostrado en la figura 3.80. El ensamble mostrado en la figura 3.78 fue utilizado en todos los gemelos digitales. El modelo mostrado en la figura 3.80 solo fue requerido una vez en el modelo de la estampadora.

Para el modelo del soporte de la figura 3.78 se utilizó una textura a base de imagen (como se explicó anteriormente se utiliza una imagen en vez de utilizar un color) y una textura PBR para los pernos. La imagen utilizada representa un metal rayado y se muestra en la figura 3.89 y fue obtenida de la referencia [40]. Para el perno se utilizaron texturas PBR las cuales son mostradas en la figura 3.90. Las imágenes PBR del perno fueron obtenidas de la referencia [41]. De la figura 3.90 puede observarse que las texturas corresponden a una variedad de ranuras (cruz, allen, línea), por lo que se tuvo que delimitar la región que se usaría por medio de la manipulación de mapas UV. El resultado de la implementación de las texturas mencionadas da como resultado el modelo mostrado en la figura 3.91, para el primer soporte.



Figura 3.89. Imagen de metal rayado. Obtenido de [40].

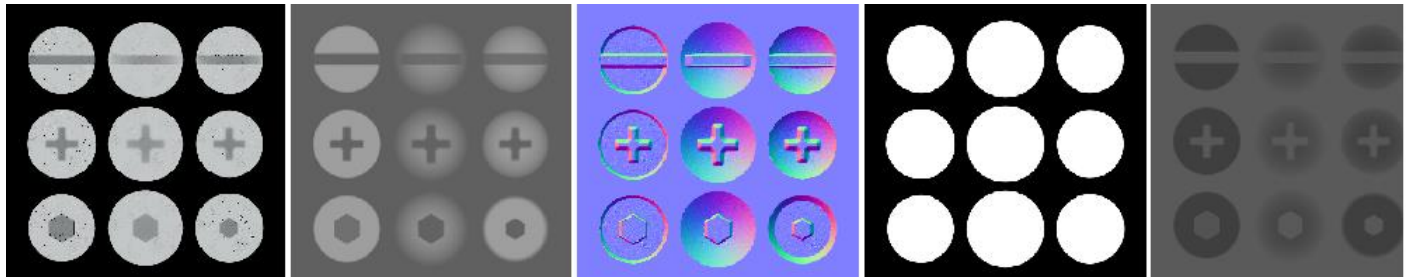


Figura 3.90. Imágenes PBR de pernos. Obtenidas de [41]. Texturas de izquierda a derecha: Color, altura (*height*), normales, opacidad (*opacity*) y rugosidad (*roughness*).



Figura 3.91. Modelo resultante después de la adición de una textura de imagen para el soporte y texturas PBR para los pernos.

Para el soporte del cilindro B de la estampadora se prefirió usar un material de color gris parecido al de los vástagos. Para sus pernos se utilizaron las mismas texturas PBR mencionadas anteriormente. La figura 3.92 muestra el resultado de agregar las texturas al modelo. La figura 3.93 muestra un acercamiento a la parte donde están colocados los pernos ya que debido al tamaño de la pieza es difícil verlos en la figura 3.92.



Figura 3.92. Soporte del cilindro B de la estampadora una vez que se agregaron texturas.



Figura 3.93. Acercamiento a la zona donde se colocan los pernos para visualizar como quedó colocada la textura.

Para las piezas restantes de la empacadora, la indentadora y la estampadora se utilizaron texturas PBR descargadas de la referencia [30]. Para las bases de todos los gemelos digitales se utilizaron las imágenes PBR mostradas en la figura 3.94.



Figura 3.94. Imágenes PBR para la representación de un material metálico. De izquierda a derecha: Textura (color), altura (*height*), Normales (*normal map*), Metálico (*metallic*), Rugosidad (*roughness*). La textura metálico no se alcanza a apreciar mucho ya que es casi blanca.

El resultado esperado de la aplicación de las imágenes PBR se muestra en la figura 3.95. La imagen mostrada en la figura 3.95 viene dentro de las imágenes PBR como una muestra de lo que sería la aplicación de las texturas. La imagen 3.96 muestra el resultado obtenido de la aplicación de las texturas a la base del modelo de la indentadora. Para las otras bases de los modelos se obtienen resultados similares. Cabe mencionar que las texturas aplicadas son sensibles a la luz por lo que los resultados mostrados, al no haber una luz aplicada, pueden lucir opacos.

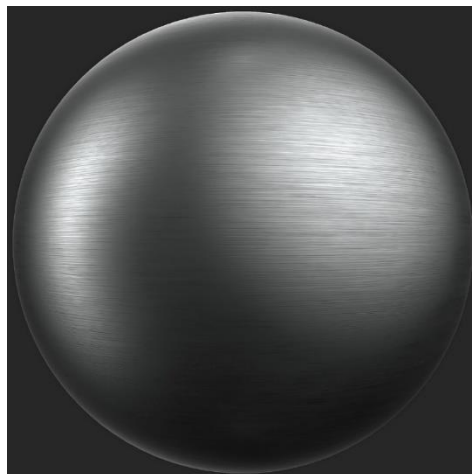


Figura 3.95. Resultado de la aplicación de las imágenes PBR mostradas en la figura 3.91. Obtenido de la carpeta de descarga de la imágenes PBR de la figura 3.90.

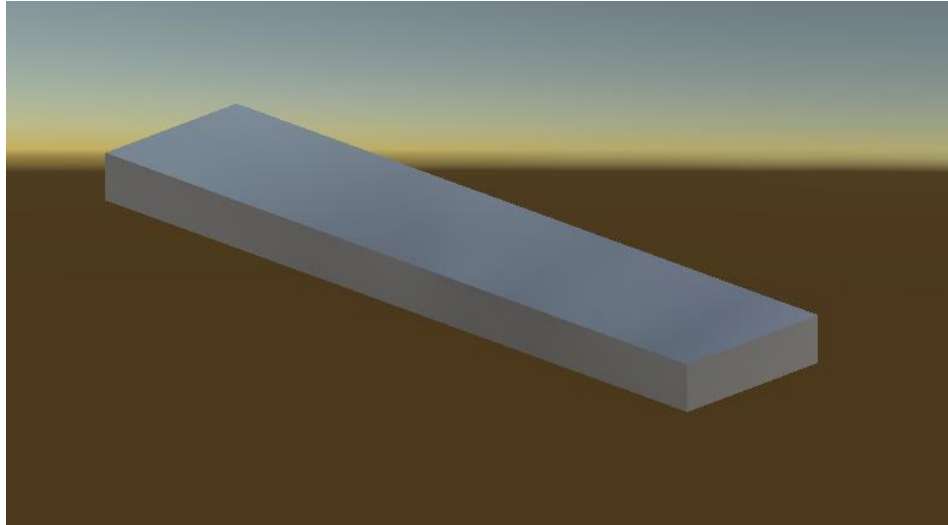


Figura 3.96. Aplicación de las texturas PBR mostradas en la figura 3.94 a la base horizontal de la indentadora. Imagen tomada desde Unity®.

Para las “rampas” de los modelos de la estampadora y la empacadora también se utilizaron las mismas texturas PBR mostradas en la figura 3.94. A los elementos guía de los modelos se les asignaron también texturas PBR de material metálico. En este caso se utilizó una textura que da por resultado el mostrado en la figura 3.97. Las imágenes PBR no se muestran en este caso ya que la idea general es la misma a la mostrada en la figura 3.94. La figura 3.98 muestra el resultado de agregar las texturas PBR a las guías de la indentadora. El uso de dos diferentes texturas para los elementos “bases” y las “guías” es para brindar contraste a los modelos y poder visualizar de mejor manera sus elementos. A las mesas de soporte de los modelos de los gemelos digitales, también se les asignó la misma textura que a las guías (figura 3.99).

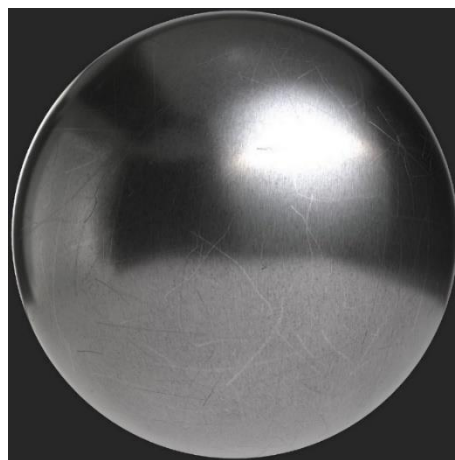


Figura 3.97. Material metálico utilizado para las guías de los gemelos digitales.

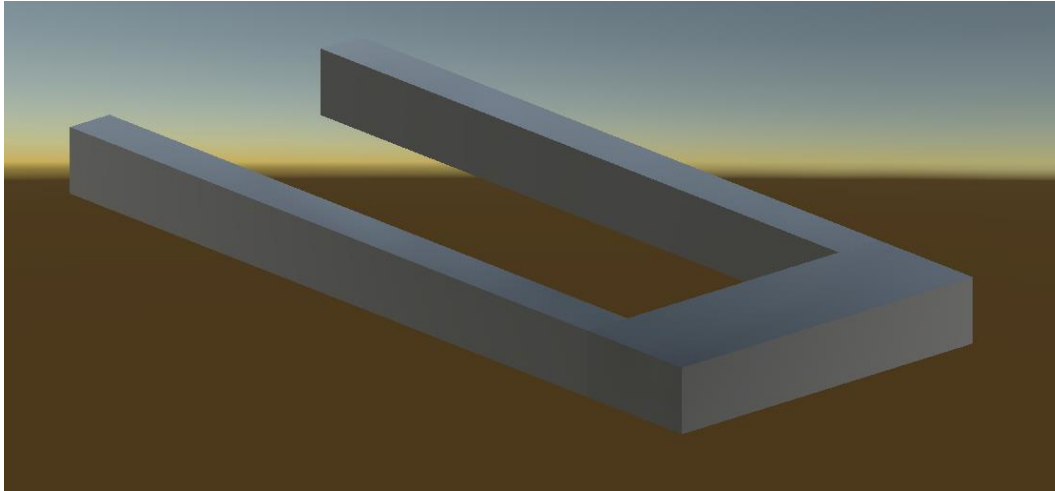


Figura 3.98. Elementos guía de la indentadora una vez colocada su textura PBR. Imagen tomada desde Unity®.

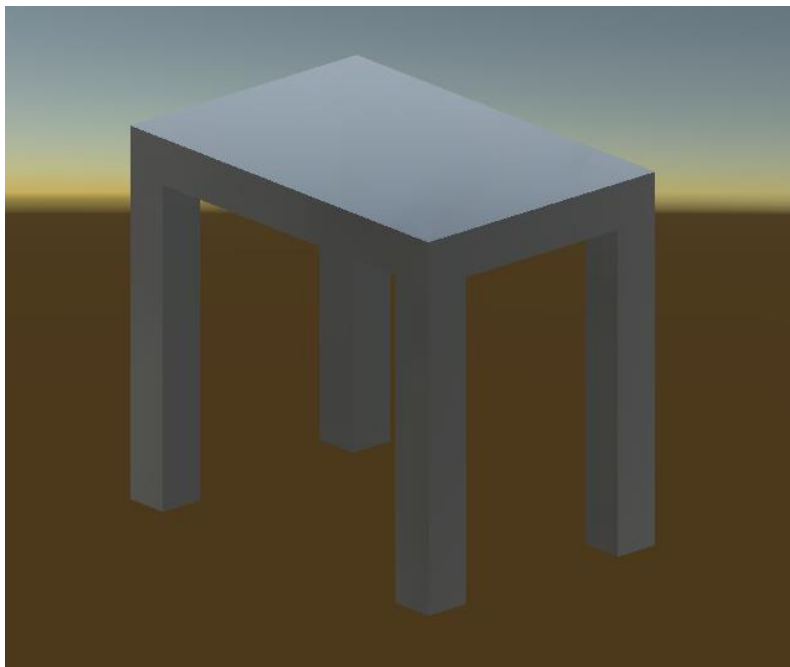


Figura 3.99. Modelo de la mesa de soporte de la indentadora después de agregadas las texturas PBR. Imagen tomada desde Unity®.

Para los almacenes de las piezas de trabajo de la empacadora y la estampadora, se utilizaron las mismas texturas utilizadas para las guías. A otros elementos de los gemelos digitales se les asignaron estas mismas texturas o materiales metálicos construidos desde Unity de colores grisáceos. En particular al cabezal del cilindro B de la estampadora se le asignó un material grisáceo transparente, para poder visualizar, durante la ejecución de la simulación, el estampado de la pieza de trabajo.

Los resultados obtenidos para los gemelos digitales después de agregar texturas se muestran en las figuras 3.100, 3.101 y 3.102 para la empacadora, la indentadora y la estampadora respectivamente.

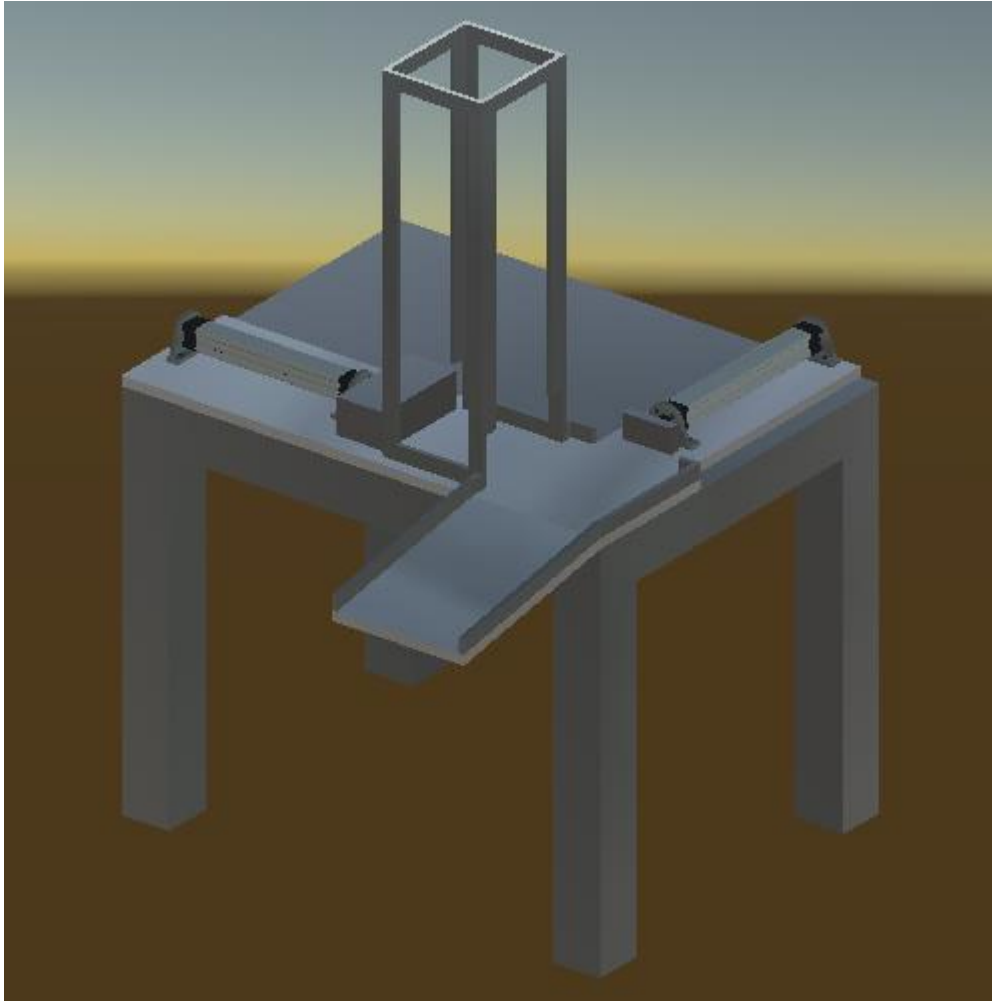


Figura 3.100. Modelo de la empacadora una vez exportado a Unity y después de agregarle texturas.

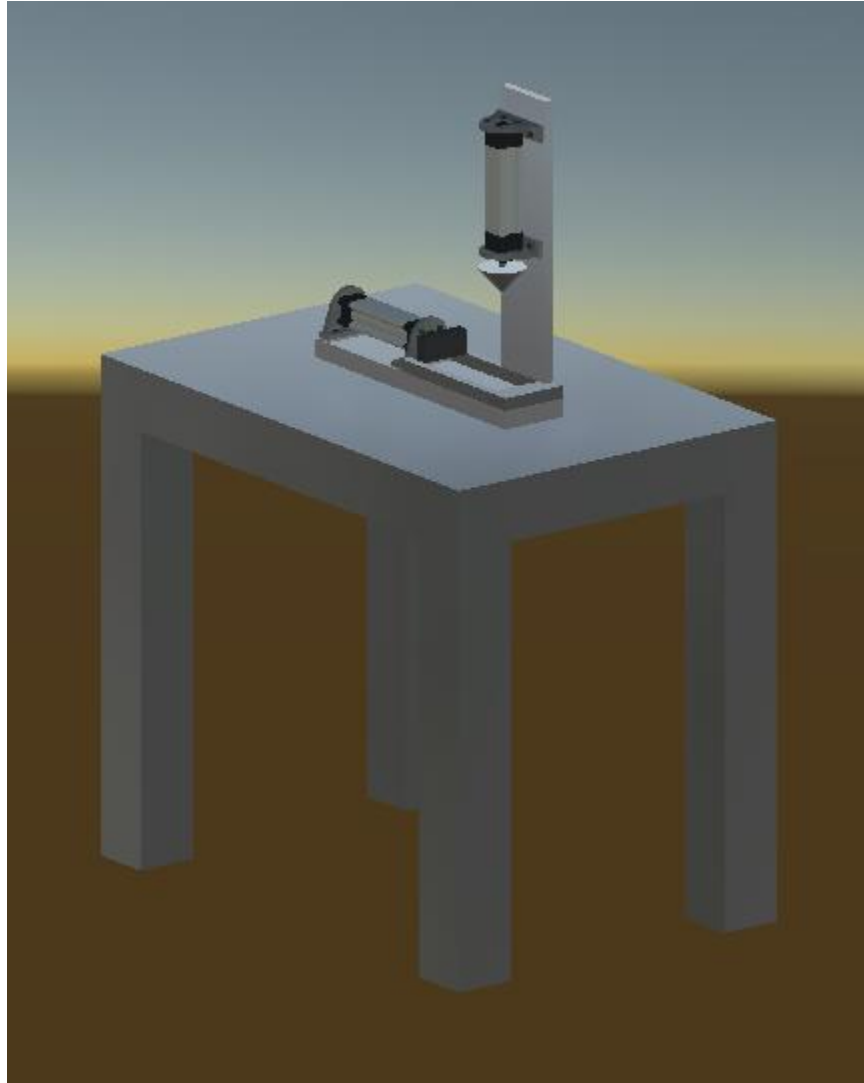


Figura 3.101. Modelo de la indentadora después de exportarlo a Unity® y agregarle texturas.

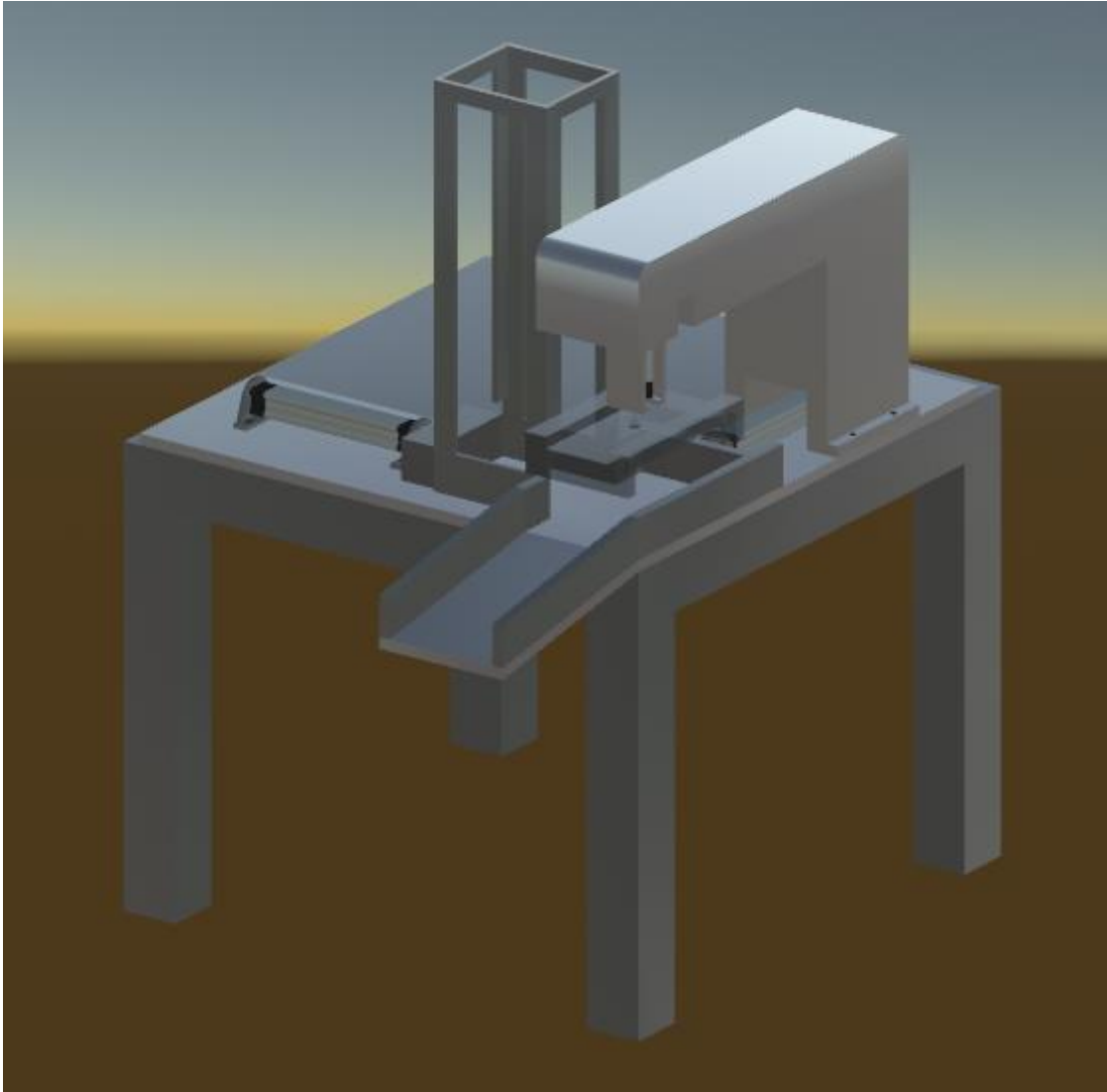


Figura 3.102. Modelo de la estampadora después de exportarlo a Unity® y de agregar de texturas.

Texturas de los elementos de trabajo

Para las piezas de trabajo de la empacadora (los paquetes) se eligió utilizar distintas texturas. En total se crearon 8 texturas distintas desde la página web de la referencia [43] y fueron agregadas al modelo de la pieza de trabajo de la empacadora como texturas de imagen. La figura 3.103 muestra una textura de las creadas y en el anexo D se muestran las restantes. El resultado de agregar esta textura a la caja utilizada en la empacadora se muestra en la figura 3.104. La figura 3.105 muestra los paquetes restantes con sus respectivas texturas.



Figura 3.103. Imagen creada desde la página web de la referencia [43] utilizada como textura de imagen para una de las piezas de trabajo de la empacadora.



Figura 3.104. Paquete utilizado en la empacadora después de agregarle textura a partir de una imagen.



Figura 3.105. Paquetes utilizados para ser empacados por la empacadora.

Para las piezas de trabajo de la indentadora y la estampadora se utilizó una misma textura PBR, cuyo resultado esperado se muestra en la figura 3.106 y corresponde a un material metálico dorado. Las figuras 3.107 y 3.108 muestran los resultados obtenidos para la pieza de trabajo del indentador (cubo) y para la pieza de trabajo de la estampadora.

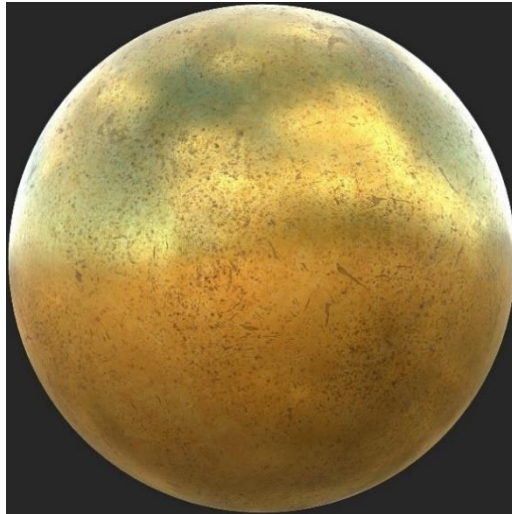


Figura 3.106. Resultado esperado de las texturas PBR aplicadas a las piezas de trabajo de la indentadora y la estampadora.

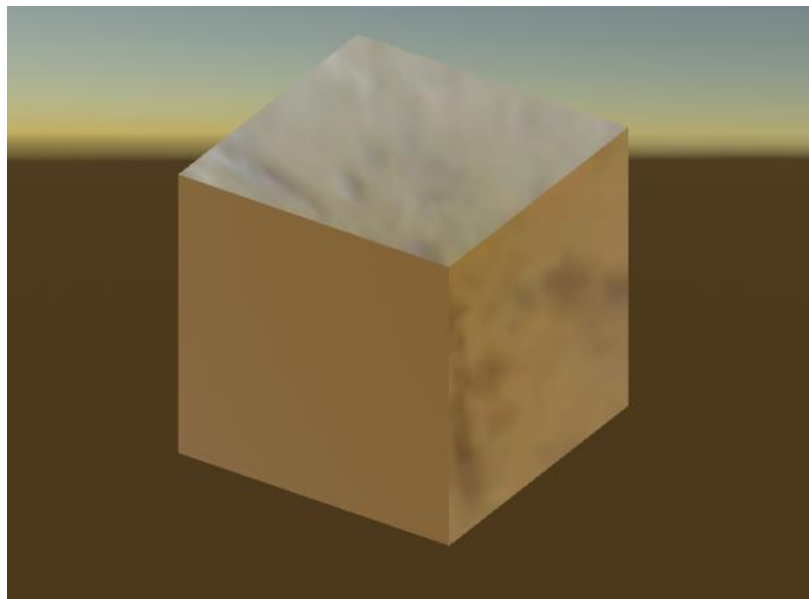


Figura 3.107. Pieza de trabajo de la indentadora una vez agregada su textura.

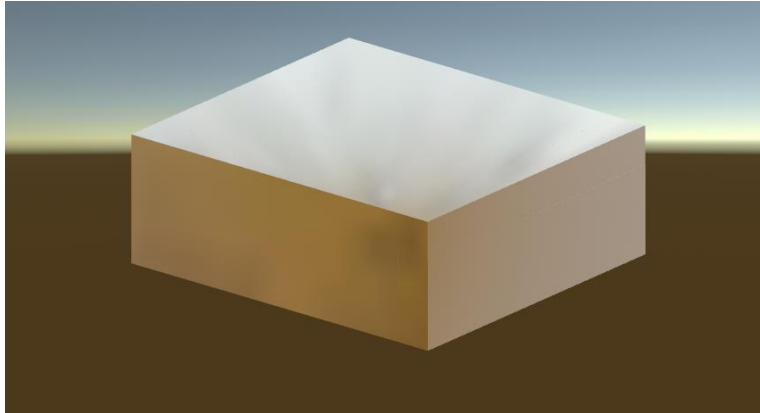


Figura 3.108. Pieza de trabajo de la estampadora con su textura agregada.

Por último, se agregó la textura de la caja contenedora de paquetes que es utilizada en los gemelos digitales de la empacadora y la estampadora. La imagen 3.109 muestra la imagen utilizada como textura para esta caja la cual fue creada en la página web de la referencia [43]. La imagen puede consultarse también en el anexo D. La figura 3.110 muestra el resultado después de haber agregado la textura a la caja contenedora de piezas. Se recordará que la caja contenedora y las tapas son piezas separadas. Para las tapas de la caja se utilizó la misma textura mostrada en la figura 3.109, pero solo se utilizaron las partes sin impresión. La figura 3.111 muestra una de las tapas de la caja una vez que se agregó su textura.

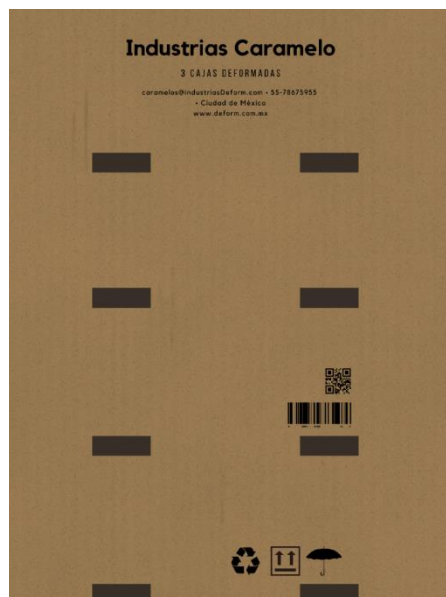


Figura 3.109. Imagen utilizada como textura para la caja contenedora de paquetes de la empacadora y la estampadora. Elaboración propia.

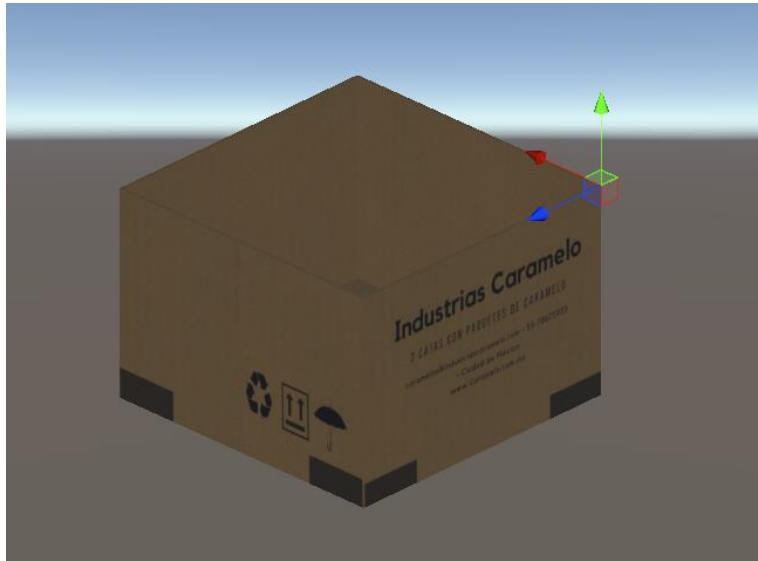


Figura 3.110. Textura agregada a la caja contenedora de paquetes de la empackadora y la estampadora.

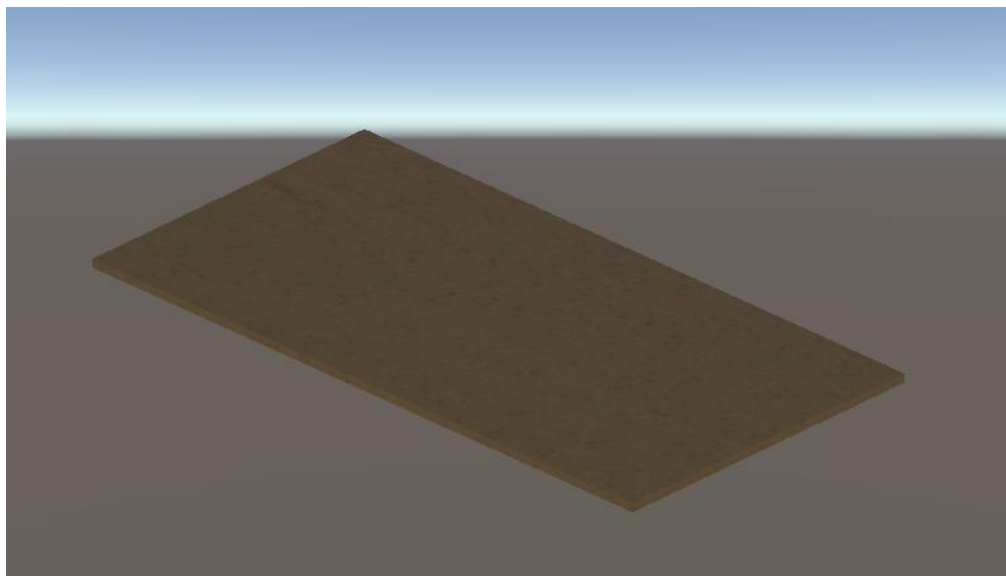


Figura 3.111. Una de las piezas que forman la tapa de la caja a la que se agregó la textura mostrada en la figura 3.109. La textura se acomodó para que las impresiones no aparecieran sobre la tapa.

4. Integración de los gemelos digitales al simulador virtual NERV

Una vez agregadas las texturas a los gemelos digitales fue momento de integrarlos al simulador NERV para su control. En la introducción se ha mencionado que el simulador NERV tiene tres etapas documentadas, sin embargo, se ha seguido con su desarrollo. Para la integración de los gemelos digitales en el simulador NERV, se recibió un archivo de Unity que contenía una mesa de trabajo neumática junto con la programación requerida para su funcionamiento. Las mesa de trabajo contenida en este archivo se muestra en la figura 4.1, donde se observan: un cilindro de simple efecto, un cilindro de doble efecto, dos temporizadores (uno positivo y otro negativo), 2 válvulas 3/2 de accionamiento mecánico (finales de carrera), 2 válvulas 3/2 accionadas por botón enclavado, 4 conectores en T, unidad de mantenimiento y un piloto neumático. El escenario donde se encuentra la mesa de trabajo también cuenta con un museo donde se pueden consultar los diversos elementos disponibles en la mesa de trabajo neumática (fig. 4.2).

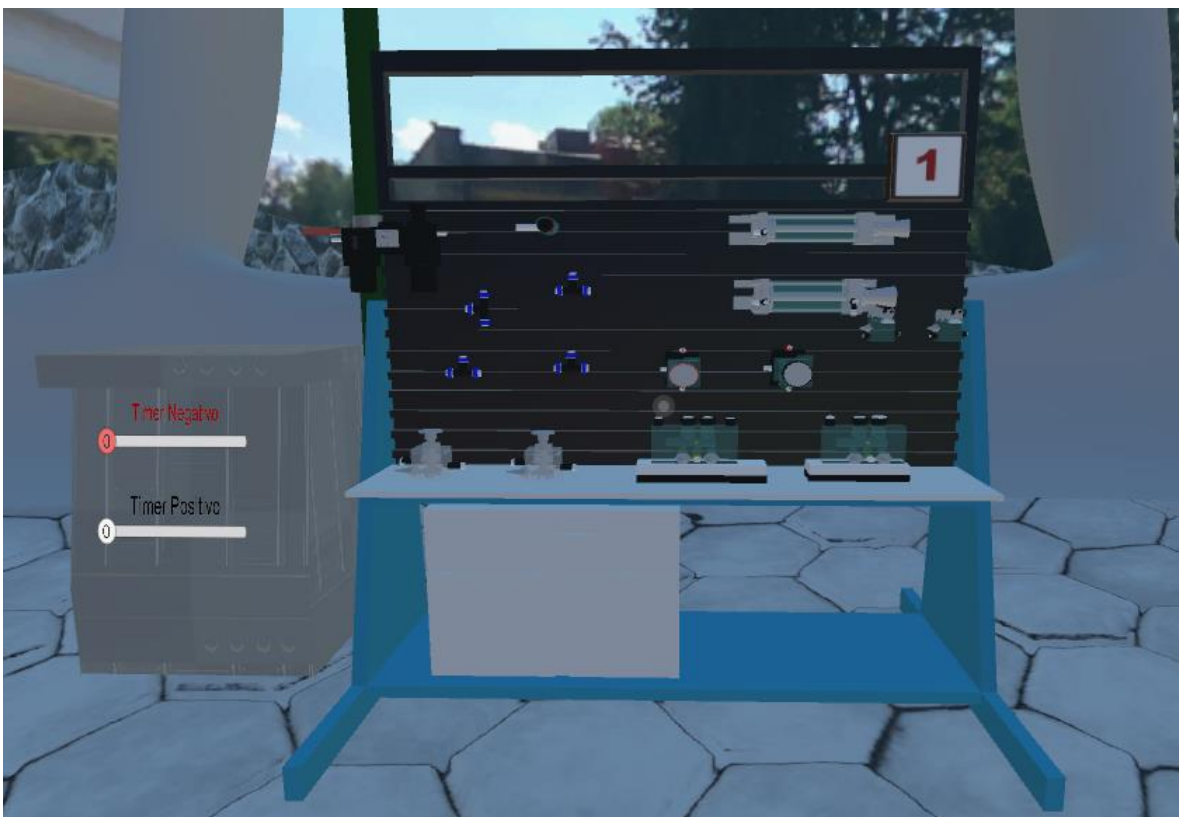


Figura 4.1. Mesa de trabajo neumática del NERV individual recibido para realizar la integración con los gemelos digitales.

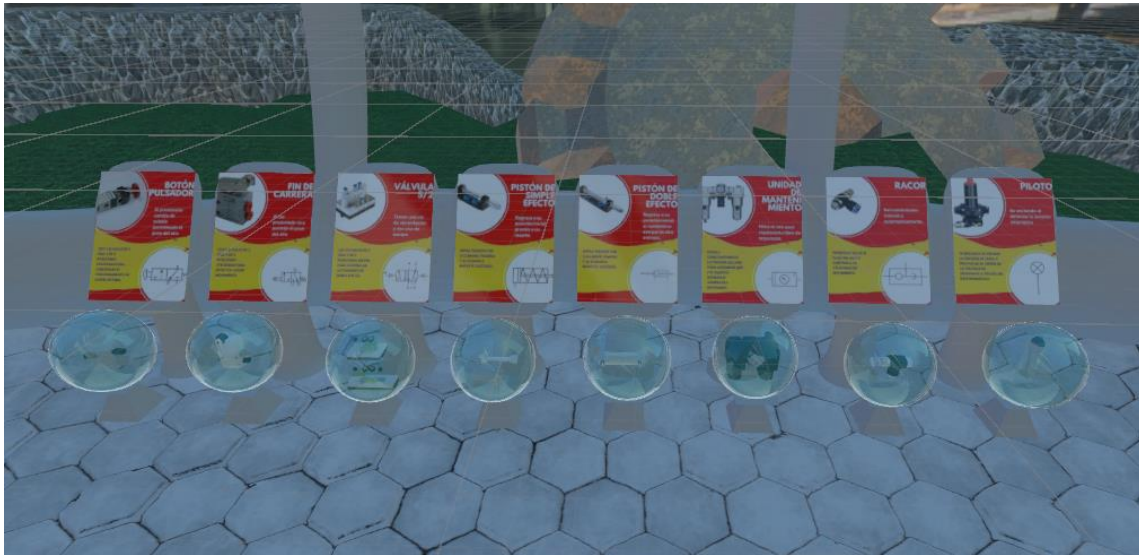


Figura 4.2. Museo donde se pueden consultar los diversos elementos que se encuentran en la mesa de trabajo neumática.

4.1. Adecuaciones al simulador virtual NERV

En la figura 4.1 podemos observar que faltan elementos para lograr el funcionamiento de los gemelos digitales. Por ejemplo, todos los gemelos digitales tienen por lo menos 2 cilindros neumáticos de doble efecto, por lo que falta 1 cilindro para la empaedora y otro para la indentadora y 2 más para la estampadora. Faltan, además, válvulas 5/2 biestables de accionamiento neumático tanto para activar los cilindros neumáticos como para su control. Faltan válvulas 3/2 de accionamiento mecánico (finales de carrera) para los cilindros adicionales y faltan conectores en T ya que los existentes no serían suficientes para lograr todas las conexiones requeridas. También podemos darnos cuenta que tenemos elementos que no utilizaríamos para el control de los gemelos digitales como: el cilindro de simple efecto y la válvula 5/2 monoestable de accionamiento neumático. Además, para controlar cada gemelo digital se precisó contar con una mesa de trabajo por modelo, por lo que se requieren dos mesas adicionales.

Lo primero que se realizó fue quitar los elementos que no se requerían para el control de los gemelos digitales: el cilindro de simple efecto y la válvula 5/2 monoestable de accionamiento neumático. Para la primera mesa de trabajo, la de la empaedora, se agregaron: un cilindro de doble efecto, dos finales de carrera, 3 válvulas 5/2 biestables de accionamiento neumático y 19 conectores T. Además, se quitó un botón enclavado ya que solo era requerido uno para el control de los gemelos digitales. También se cambió el color de la unidad de mantenimiento ya que, al ser de color negro, no podía verse bien debido al color de la mesa, que era del mismo color. Los elementos se acomodaron como puede observarse en la figura 4.3 donde, además se agregaron etiquetas de identificación a los cilindros neumáticos y a la mesa neumática.

Una vez completadas las adecuaciones a la primera mesa, se hizo una copia de ésta para asignársela al gemelo digital de la indentadora. Solo se cambió la etiqueta de la estampadora por la de la indentadora y la etiqueta del número de la mesa (fig. 4.4).

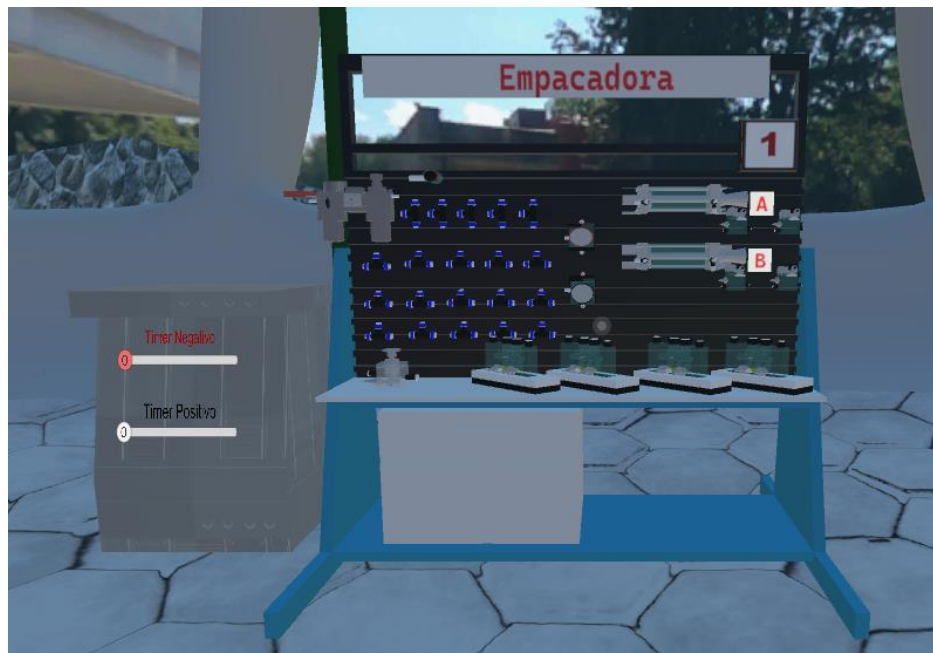


Figura 4.3. Mesa neumática para control de la empacadora.



Figura 4.4. Mesa neumática para el control de la indentadora.

La misma mesa de la empacadora se copió una segunda vez para asignarla al control de la estampadora. En este caso además se tuvieron que agregar: un cilindro neumático de doble efecto, válvulas 3/2 de accionamiento mecánico (finales de carrera) y una válvula 5/2 biestable de accionamiento neumático. Se agregó la etiqueta correspondiente a la estampadora y una etiqueta adicional para el último cilindro agregado. También se cambió la etiqueta correspondiente al número de la mesa (figura 4.5).



Figura 4.5. Mesa neumática para el control de la estampadora.

Junto a cada mesa de trabajo neumática se agregó el correspondiente gemelo digital. Las figuras 4.6, 4.7 y 4.8 muestran cada uno de los gemelos digitales al lado de su mesa de trabajo correspondiente. A los gemelos digitales también se les agregaron etiquetas de identificación para los cilindros. En las figuras 4.6 y 4.7 correspondientes a la empacadora y la estampadora respectivamente, se puede observar la adición de una mesita, la cual se utiliza para colocar la caja contenedora de piezas. Para esta mesita solo se hizo una copia de la mesa de la empacadora y se escaló al tamaño mostrado, por lo que no se requirió hacer un modelo nuevo.

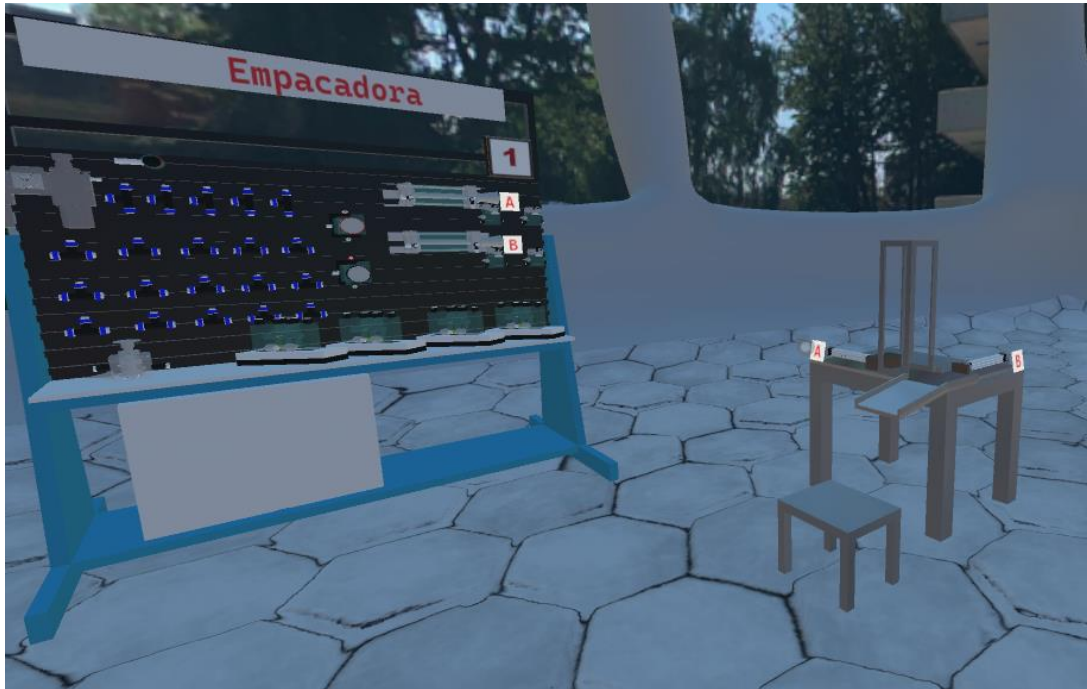


Figura 4.6. Gemelo digital de la empacadora junto a su mesa de trabajo. Se observa la adición de una mesita donde descansará la caja contenedora de paquetes.

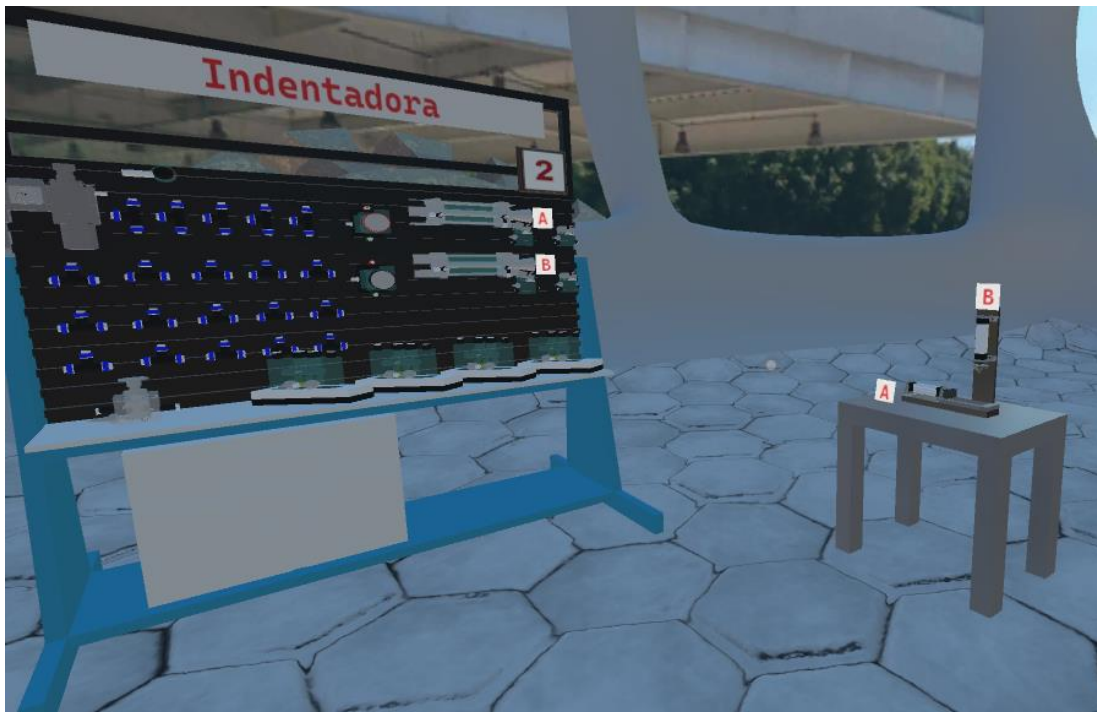


Figura 4.7. Gemelo digital de la indentadora junto a su mesa de trabajo.

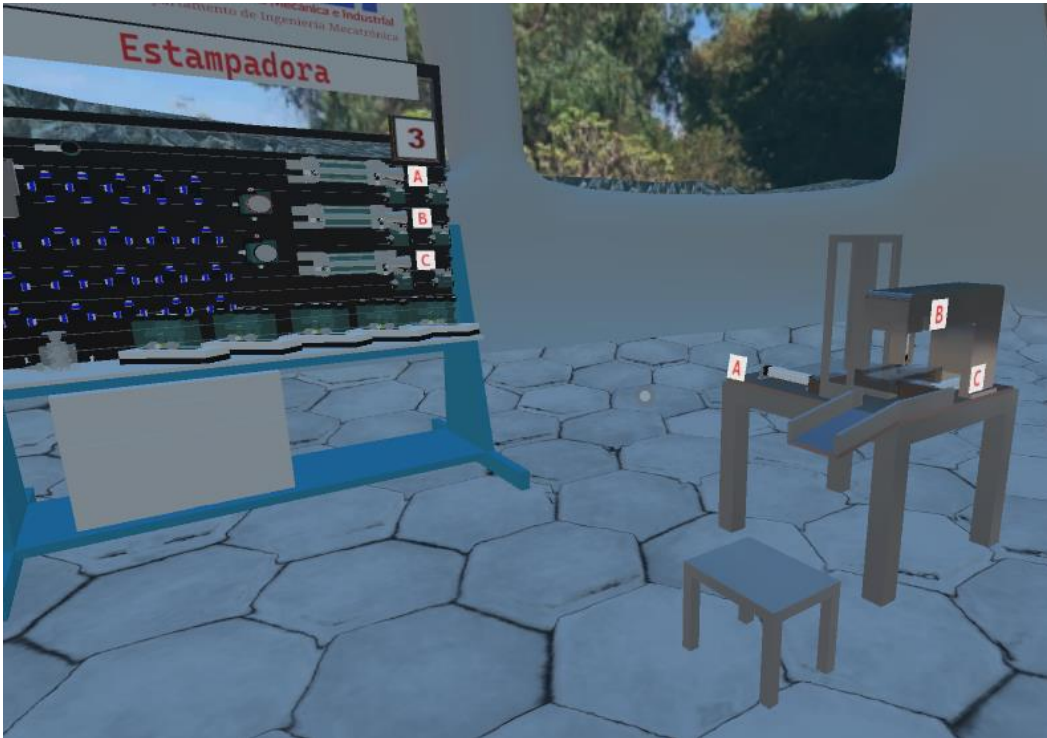


Figura 4.8. Gemelo digital de la Estampadora junto a su mesa de trabajo. Se observa la adición de una mesita donde descansará la caja contenedora de piezas estampadas.

4.2. Correcciones al simulador virtual NERV

Se tuvieron que hacer algunas correcciones al código de las mesas de trabajo para que funcionaran de manera correcta una vez hechas las adecuaciones requeridas.

Se tiene que mencionar que el código que acompaña a la mesa de trabajo neumática proporcionada para la presente implementación, no está orientado a objetos por lo que al hacer copias de los elementos (cilindros, válvulas, conectores), se tuvieron que agregar líneas de código adicionales al código principal. El código, que puede consultarse en el anexo E, está organizado por funciones por lo que en el código principal de la mesa neumática hay funciones para el control de cada uno de los elementos de la mesa neumática, por ejemplo, funciones para el control del funcionamiento de las válvulas, de los cilindros, de los conectores, etc. Las líneas de código adicionales que se tuvieron que añadir son en su mayoría llamadas a las funciones y solo se corrigieron algunos pequeños errores de lógica. En la figura 4.9 puede observarse parte del código donde se llaman a las funciones que controlan los elementos de la mesa de trabajo. Las líneas de código que se agregaron fueron para controlar los elementos adicionales que se copiaron, por ejemplo, los finales de carrera, las válvulas 5/2 biestables y los conectores. También algunas líneas de código se eliminaron ya que no se utilizaron los elementos que controlaban, por ejemplo, las llamadas a las funciones que controlan los cilindros de simple efecto o las válvulas 5/2 monoestables.

```

238 //Activación de la unidad de mantenimiento
239 NoHayFuga(sonidoFuga, botones);
240 ActivacionUM(10, Num, isActive, botones);
241
242 //Botones
243 Botonsito(22, 23, "Boton1", animBoton, Num, isActive, botones);
244 Botonsito(20, 21, "Boton1", animBoton2, Num, isActive, botones);
245
246 //FINALES DE CARRERA
247 FinalCarrera(24, 25, "finalBool", animFinal, Num, isActive, botones);
248 FinalCarrera(26, 27, "finalBool", animFinal2, Num, isActive, botones);
249
250 //VALVULAS
251 Valvula52Biestable(19, 16, 17, 18, 15, animValv52B, "val52Bool", Num, isActive, botones);
252 Valvula52Monoestable(14, 11, 12, 13, animValv52M, "val52Bool", Num, isActive, botones);
253
254 //CONECTAR T
255 ConectarT(00, 01, 02, Num, isActive, botones);
256 ConectarT(03, 04, 05, Num, isActive, botones);
257 ConectarT(06, 07, 30, Num, isActive, botones);
258 ConectarT(31, 32, 33, Num, isActive, botones);
259
260 //PISTONES
261 ActivarPistonSimple(28, animVastago, "VastagoAnimBool", animResorte, "ResorteAnimBool", sonidoPiston, isActive, botones);
262 ActivarPistonDoble(8, 9, animVastago2, "VastDobAnimBool", sonidoPiston, isActive, botones);
263
264 //PPILOTO
265 ActivacionPiloto(29, isActive, botones);
266
267 //Timers Prueba
268 TimerNegativo(34, 35, 36, Num, isActive, botones);
269 TimerPositivo(37, 38, 39, Num, isActive, botones);

```

Figura 4.9. Código de la mesa de trabajo neumática donde se observan las llamadas a las funciones que controlan los componentes de la mesa.

La figura 4.10 muestra las líneas de código de las llamadas a la función “*Botoncito*” la cual controla el funcionamiento del botón enclavado. Como se recordará, inicialmente había dos botones enclavados, pero se quitó uno, por lo que se ha quitado también una llamada a la función (ver figura 4.11). Se puede observar que la línea que se quería remover se convirtió en un comentario para que el compilador la ignorara. En este punto conviene señalar el cambio en los elementos enviados a la función “*Botoncito*”. Como puede observarse en la figura 4.10 se envían dos elementos enteros a la función “*Botoncito*”, además de otros elementos que no se explicarán aquí. Se puede observar que en la primera llamada a la función se envían los enteros 22 y 23 mientras que después de quitar la segunda llamada a la función se envían los enteros 61 y 62 (fig.4.11). Estos números enteros representan las conexiones de los componentes de las mesas neumáticas. Así, por ejemplo, a la salida de la unidad de mantenimiento se le asigna el número entero 101 (originalmente tenía el número 10), a una conexión de los conectores en T se le asigna el número 0, a otra el 1 y a otra el 2 y así sucesivamente con cada conexión de los componentes, de modo que ninguna conexión se repita. Entonces a la función “*Botoncito*” se le están enviando las conexiones del botón enclavado. Esto se menciona porque cuando se agregaron componentes adicionales fue necesario reasignar los valores numéricos de las conexiones debido a que se tuvieron problemas cuando se asignó la conexión número 100. El programa solo estuvo pensado para un número menor a 100 conexiones. Para solucionar este inconveniente se incrementó el número máximo de elementos a 1000 y se cambió la numeración de las conexiones para seguir un orden junto con los nuevos elementos agregados.

```

242 //Botones
243 Botonsito(22, 23, "Boton1", animBoton, Num, isActive, botones);
244 Botonsito(20, 21, "Boton1", animBoton2, Num, isActive, botones);

```

Figura 4.10. Llamadas a la función Botoncito para el control del botón enclavado. Los números enteros en la llamada a la función representan las conexiones de entrada y salida del botón enclavado.

```

255 //Botones
256 Botonsito(61, 62, "Boton1", animBoton, Num, isActive, botones);
257 //Botonsito(20, 21, "Boton1", animBoton2, Num, isActive, botones); Se quitó esta línea

```

Figura 4.11. Se ha quitado una llamada a la función "botoncito" en vista de que se ha eliminado una válvula 3/2 accionada por botón enclavado.

Continuando con los cambios realizados al programa; se agregaron líneas de código para los finales de carrera adicionales. La figura 4.12 muestra el código original donde se requerían dos llamadas a la función final de carrera ya que había únicamente dos de estos elementos. La figura 4.13 muestra las llamadas a función agregadas para el caso de la mesa de trabajo de la empacadora y de la indentadora. Para la estampadora se requieren dos líneas adicionales debido a que se tienen 3 cilindros neumáticos (fig. 4.14).

```

246 //FINALES DE CARRERA
247 FinalCarrera(24, 25, "finalBool", animFinal, Num, isActive, botones);
248 FinalCarrera(26, 27, "finalBool", animFinal2, Num, isActive, botones);

```

Figura 4.12. Llamadas a la función "FinalCarrera" en el código original, una por elemento.

```

259 //FINALES DE CARRERA
260 FinalCarrera(91, 92, "finalBool", animFinal, Num, isActive, botones);
261 FinalCarrera(93, 94, "finalBool", animFinal2, Num, isActive, botones);
262 FinalCarrera(97, 98, "finalBool", animFinal3, Num, isActive, botones); // Se Agregó esta línea
263 FinalCarrera(99, 100, "finalBool", animFinal4, Num, isActive, botones); // Se Agregó esta línea

```

Figura 4.13. Se agregaron 2 llamadas a la función "FinalCarrera" debido a la adición de un cilindro neumático (se requieren dos finales de carrera por cilindro).

```

296 //FINALES DE CARRERA
297 FinalCarrera(91, 92, "finalBool", animFinal, Num, isActive, botones);
298 FinalCarrera(93, 94, "finalBool", animFinal2, Num, isActive, botones);
299 FinalCarrera(97, 98, "finalBool", animFinal3, Num, isActive, botones); // Se agregó esta línea
300 FinalCarrera(99, 100, "finalBool", animFinal4, Num, isActive, botones); // Se agregó esta línea
301 FinalCarrera(104, 105, "finalBool", animFinal5, Num, isActive, botones); // Se agregó esta línea
302 FinalCarrera(106, 107, "finalBool", animFinal6, Num, isActive, botones); // Se agregó esta línea

```

Figura 4.14. En total se tienen 6 llamadas a la función "FinalCarrera" para el caso de la estampadora, una por final de carrera.

De manera similar se agregaron líneas adicionales para las funciones de control de las válvulas 5/2 biestables, para los conectores en T y para los cilindros neumáticos de doble efecto agregados.

Una de las correcciones más significativas realizadas al código fue la modificación de la función que controla el funcionamiento de las válvulas 5/2 biestables de accionamiento

neumático. La figura 4.15 muestra el código de esta función, pero con las instrucciones *if* contraídas ya que el código es muy amplio (el código original completo puede consultarse en el anexo E y el código modificado para las mesas de trabajo con dos cilindros neumáticos se puede consultar en el anexo F). En la línea 497 se crea la función de nombre: *Valvula52Biestable*. Dentro de esta función la línea 501 verifica que no exista una fuga de aire. Si no existe fuga de aire se ejecutan las instrucciones dentro de la instrucción *if* de la línea 503. En caso contrario se ejecuta la instrucción *else* de la línea 627. Dentro de la instrucción *if* de la línea 503 se tienen dos instrucciones: un *if* y un *else*. La instrucción *if* de la línea 506 indica lo que se debe realizar si la válvula está alimentada y el *else* de la línea 597 indica lo que se debe realizar si la válvula no está alimentada.

```

497 public void Valvula52Biestable(int alimentacion, int botIzq, int camIzq, int camDer, int botDer,
498     Animator anim52Bool, string val52Bool, int[] Num, bool[] isActive, GameObject[] botones)
499     {
500         //Comprueba si ya no hay fuga
501         NoHayFuga(sonidoFuga, botones);
502         //Si NO hay fuga
503         if (IsActiveGeneral == false)
504         {
505             //Si se esta alimentando la valvula
506             if (botones[alimentacion].GetComponent<MeshRenderer>().material.color == Color.blue)
507             {
508                 //Si NO se esta alimentando la valvula
509                 else
510             }
511         }
512         //Si hay fuga
513         else
514     }
515 }

```

Figura 4.15. Se muestra la función *Valvula52Biestable* que controla el funcionamiento de las válvulas 5/2 biestables de accionamiento neumático.

Para el caso cuando la válvula está alimentada, el código está bien implementado, pero cuando se tiene el caso de que no pasa aire a través de la alimentación es donde se encontraba el problema. Para este segundo caso, solo se consideró que, si no pasaba aire, entonces no debía salir aire por las vías, no se consideró que puede ocurrir movimiento de los pilotajes aun cuando no se tuviera presión de alimentación en la válvula. Esto fue un problema durante la implementación ya que, en algunos casos, las válvulas de memoria deben cambiar de posición aun cuando no estén alimentadas. Entonces la corrección consistió en agregar las líneas de código que controlan la posición de trabajo y de reposo de las válvulas 5/2 biestables cuando no están alimentadas. La figura 4.16 muestra las líneas de código agregadas las cuales corresponden a las instrucciones *if* de las líneas 405 y 411 en esta figura. La instrucción *if* de la línea 405 indica que, si la utilización 12 de la válvula 5/2 biestable tiene presión de aire y la utilización 14 no tiene presión de aire, entonces la válvula se mueve a su posición de reposo. La instrucción *if* de la línea 411 indica que, si la utilización 14 tiene presión de aire y la utilización 12 no tiene presión de aire, entonces la válvula se mueve a su posición de trabajo. No se especifica acción alguna si ambas utilizaciones tienen o no tienen presión de aire simultáneamente, por lo que si esto ocurre no hay cambio en la posición de la válvula. Se puede observar que estas líneas están por

fuera de las instrucciones que controlan el funcionamiento de la válvula cuando está y no está alimentada, por lo que independientemente de esto habrá movimiento de la posición de la válvula si algún pilotaje se activa (siempre y cuando no haya fuga de aire).

```
395 public void Valvula52Biestable(int alimentacion, int botIzq, int camIzq, int camDer, int botDer,
396     Animator anim52Bool, string val52Bool, int[] Num, bool[] isActive, GameObject[] botones)
397 {
398     // Comprueba si ya no hay fuga
399     NoHayFuga(sonidoFuga, botones);
400
401     //Si NO hay fuga
402     if (IsActiveGeneral == false)
403     {
404
405         if (botones[botDer].GetComponent<MeshRenderer>().material.color != Color.blue &&
406             botones[botIzq].GetComponent<MeshRenderer>().material.color == Color.blue)
407         {
408             anim52Bool.SetBool(val52Bool, true);
409         }
410
411         if (botones[botDer].GetComponent<MeshRenderer>().material.color == Color.blue &&
412             botones[botIzq].GetComponent<MeshRenderer>().material.color != Color.blue)
413         {
414             anim52Bool.SetBool(val52Bool, false);
415         }
416
417         //Se está alimentando la valvula
418         if (botones[alimentacion].GetComponent<MeshRenderer>().material.color == Color.blue) ...
419         //Si NO se esta alimentando la valvula
420         else ...
421     }
422
423     //Si hay fuga
424     else ...
425 }
```

Figura 4.16. Se modificó el código mostrado en la figura 4.15 para considerar el caso donde las válvulas 5/2 no están alimentadas, pero deben cambiar de posición.

Otra modificación realizada fue en la función que controla el temporizador negativo. En la figura 3.3 se mostró cómo usar un temporizador negativo como memoria neumática para el control del ciclo de movimiento de la empacadora. También se hizo una pequeña corrección a la función del temporizador negativo para que se pudiera utilizar en esta implementación como memoria neumática. En este caso el problema surgió porque para implementar el temporizador como memoria neumática se requiere una apertura del 100% del temporizador, en otras palabras: que obstruya el paso de aire “casi” de inmediato. En el simulador no se controla la apertura del temporizador sino el tiempo. Cuando el usuario ponía un tiempo de 0 segundos en el temporizador negativo, la válvula de potencia del cilindro que se estaba controlando no cambiaba de posición, porque inmediatamente el temporizador impedía el paso de aire. Para el temporizador negativo físico (real) en realidad tener una apertura de 100% no quiere decir que sean 0 segundos, ya que a pesar de que se espera que sea lo más cercano a 0 segundos; aun así, se debe llenar el tanque de almacenamiento de aire del temporizador, de modo que en realidad no son 0 segundos, pero es algo aproximado. Para la computadora 0 segundos es exacto, por lo que no ocurría el cambio de posición de la válvula de potencia desde el inicio del ciclo cuando debía moverse a su posición de trabajo. Para corregir esto en vez de 0 segundos se utilizó 0.2

segundos como el valor mínimo de espera para el temporizador negativo, con esto se evitó que el paso de aire a través del temporizador negativo se bloqueara antes de que la válvula de potencia cambiara de posición

5. Programación de los gemelos digitales

Para la programación del movimiento de los vástagos de los gemelos digitales se utilizaron métodos de las bibliotecas de Unity para trabajar con física. Se prefirió utilizar estos métodos para intentar que los gemelos digitales funcionaran con fuerzas similares con las que operarían en la realidad. Debido a que el simulador NERV recibido para realizar la integración con los gemelos digitales está implementado en la plataforma VRChat, se requirió el SDK proporcionado por VRChat para realizar la implementación. Antes de realizar la programación fue requerido agregar los componentes adecuados a los GameObjects en escena.

5.1. Programación de la empacadora

Primero se agregaron los colisionadores (*Colliders*) al gemelo digital de la empacadora. En la figura 5.1 se muestra un colisionador *BoxCollider* atado a la base de la empacadora. Como puede observarse el *BoxCollider* cubre por completo la geometría de la base, incluso las partes vacías. Esta es la razón por la que muchas piezas se hicieron por partes y no en modelos completos. Para la base esto no supone ningún problema, ya que los paquetes no tienen interacción con la zona vacía que cubre el colisionador y en caso de que, por alguna razón se tuviera interacción con esa zona, no ocurrirían efectos extraños ya que por debajo la mesa provocaría el mismo efecto. La figura 5.2 muestra el colisionador atado a la rampa de la empacadora, en este caso el colisionador es un poco más delgado que la pieza debido a que, cuando se tienen dos colisionadores uno al lado de otro, se encontró que los paquetes tienden a “tropezar” y hacer movimientos no deseados (como levantarse y girar). Este “tropiezo” se eliminó haciendo el colisionador un poco más delgado para que la pieza no se encontrara directamente con el colisionador, sino que callera sobre él.

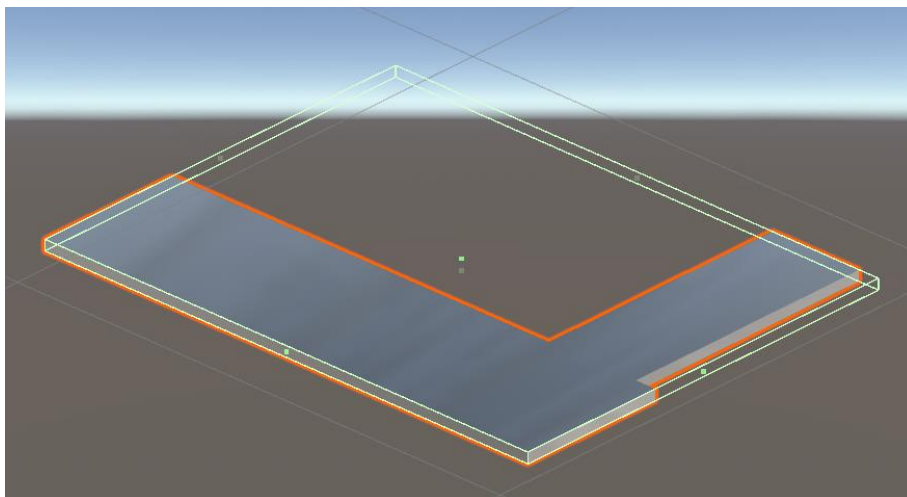


Figura 5.1. Base de la estampadora a la que se le ha atado un *BoxCollider* (prisma de color verde).

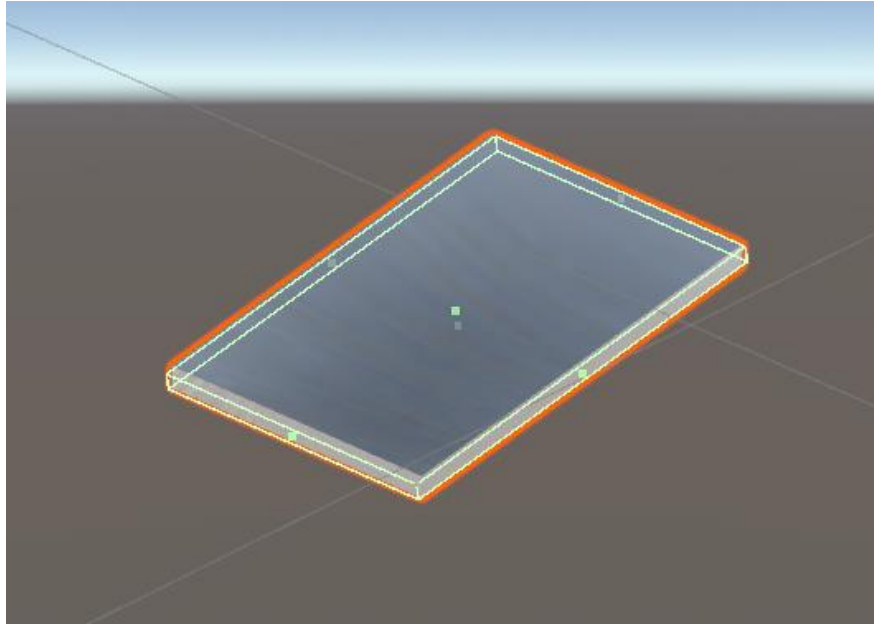


Figura 5.2. Colisionador *BoxCollider* atado a la rampa de la empacadora de cajas. El colisionador se hizo un poco más delgado que la pieza.

Al componente *BoxCollider* de la base de la empacadora y al de la rampa se les agregó la propiedad *PhysicsMaterial* para controlar la fricción entre la base y las piezas que deslizan por estos elementos. En la figura 5.3 se muestra la propiedad *PhysicsMaterial* nombrada *FricciónBase*. El valor adecuado de la fricción se estableció después de una serie de pruebas de deslizamiento de los paquetes por la base de la empacadora. Se puede observar que la fricción es un número muy pequeño, pero no cero. Utilizar un valor de cero ocasionaba que los paquetes deslizaran aun cuando no se les aplicaba directamente una fuerza, y utilizar valores un tanto elevados (pero siempre menores que 1) ocasionaba que para mover los paquetes se requiriera más fuerza, lo que provocaba que los paquetes salieran disparados.

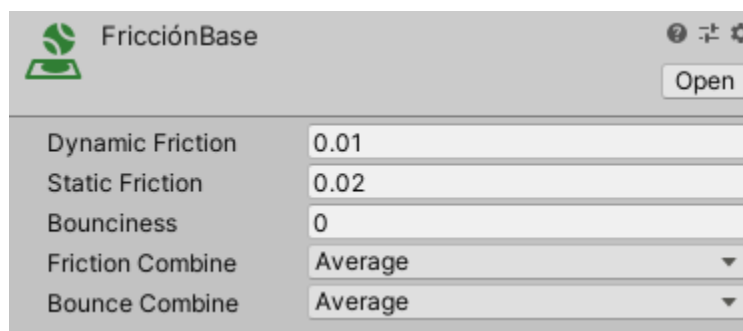


Figura 5.3. Se eligieron los valores de 0.01 para la fricción dinámica y 0.02 para la fricción estática de la base de la empacadora.

En la figura 5.4 se muestra el almacén de piezas de la empacadora, donde las líneas dibujadas en verde definen los colisionadores que se le colocaron. Aquí se muestra lo que sería el ensamblaje mostrado en la figura 3.18 de la sección 3, pero en realidad a cada una de las partes del ensamblaje (fig. 3.19) se le agregó un colisionador por separado como se muestra en la figura 5.5, donde se puede observar como el colisionador *BoxCollider* cubre la pieza por completo (aún las partes vacías), pero esto no afecta la interacción con las piezas ya que estas no deberían salir por entre los espacios vacíos. También a las guías de la empacadora se les agregó su colisionador de manera individual. En la figura 5.6 se muestran las guías con sus colisionadores *BoxCollider*. En este caso se optó por mostrar las guías en su posición relativa a la base y la rampa de la empacadora. Se puede observar que una de las guías tiene un colisionador más grande que el elemento. Esto se hizo así para impedir que el paquete saliera disparado cuando se empujaba con mucha fuerza. En lugar de modificar la pieza se prefirió simplemente modificar su colisionador. Tanto para el almacén de piezas como para las guías se utilizó una fricción de cero para evitar que la fricción pudiera ocasionar un movimiento que atascara la pieza de trabajo (fig. 5.7).

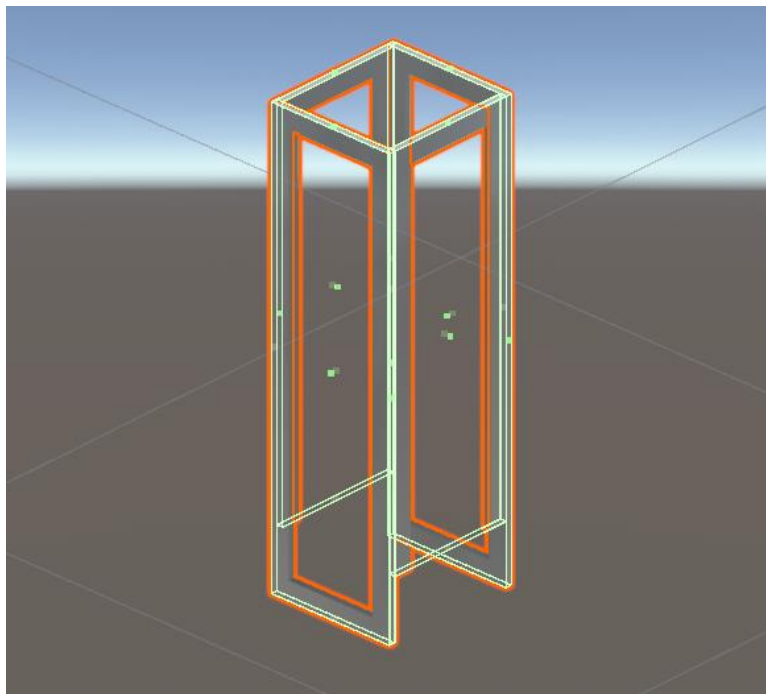


Figura 5.4. Almacén de piezas de la empacadora.

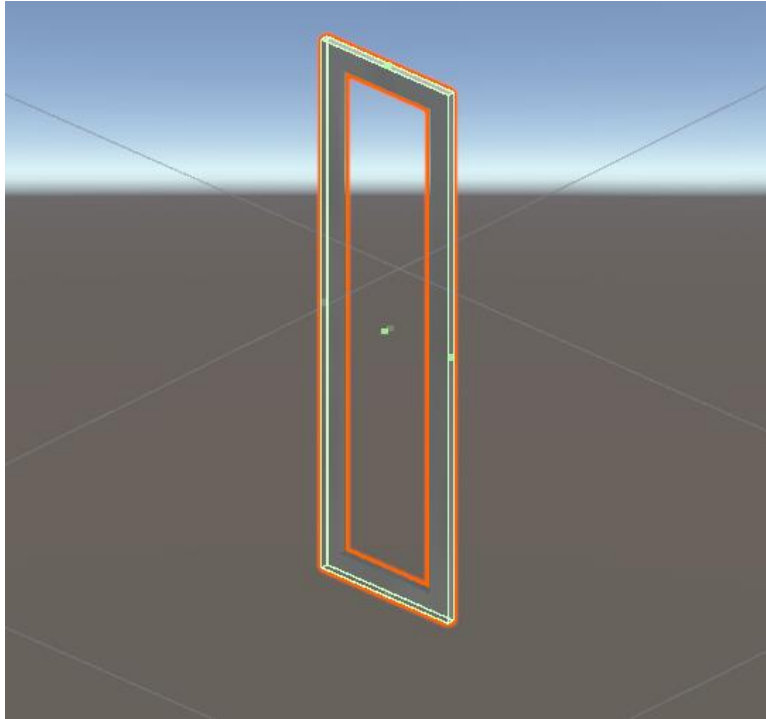


Figura 5.5. El componente *BoxCollider* (en verde) cubre por completo una de las piezas del modelo del soporte almacenador de piezas

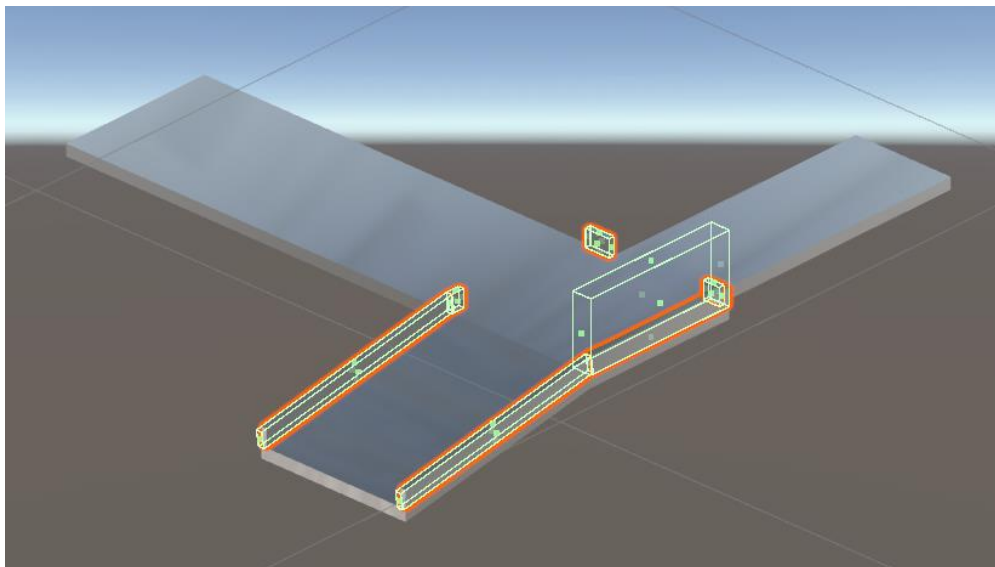


Figura 5.6. Componentes *BoxCollider* atados a las guías de la empacadora.

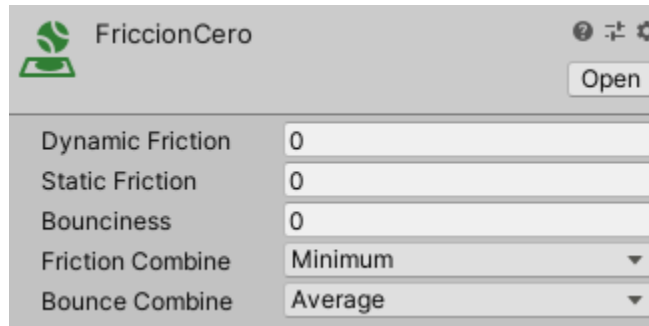


Figura 5.7. Para el almacén de piezas y para las guías se utilizó una fricción de 0.

Para los cilindros neumáticos se utilizaron colisionadores compuestos (varios colisionadores sobre la pieza) para poder controlar tanto el movimiento de los vástagos como las colisiones con otros colisionadores. En la figura 5.8 se muestra el ensamble del vástago y cabezal del cilindro A de la empacadora. Como puede observarse este ensamble tiene dos colisionadores atados y no se cubre toda la pieza (queda parte del vástago sin cubrir). Uno de los colisionadores cubre el cabezal y parte del vástago y el otro colisionador está por fuera del ensamble. El colisionador que cubre el cabezal del vástago es el responsable de empujar los paquetes y también de sostenerlos en el almacén cuando se encuentra extendido. Durante las pruebas de desarrollo se encontró que los paquetes se inclinaban demasiado si el colisionador solo cubría el cabezal por lo que este colisionador se extendió para evitar tal inclinación. La inclinación de los paquetes no afecta el funcionamiento del sistema, pero el movimiento no luce bien por lo que se prefirió evitar la inclinación de los paquetes.

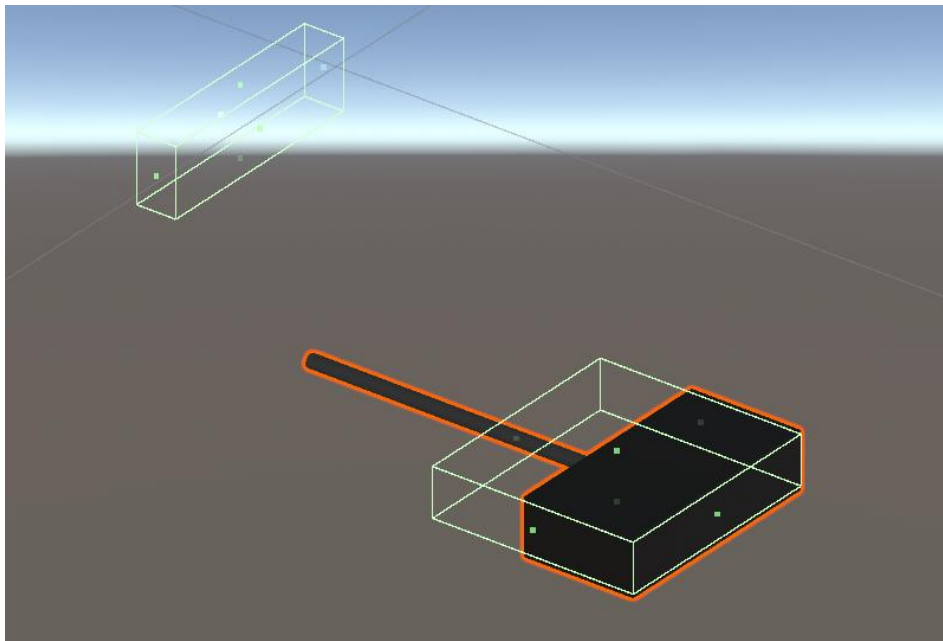


Figura 5.8. Colisionadores colocados al ensamble de vástago-cabezal del cilindro A de la empacadora.

El colisionador restante, opuesto al colisionador del cabezal interactúa con los colisionadores del cilindro mostrados en la figura 5.9. Estos últimos colisionadores delimitan la carrera del vástago por lo que los colisionadores se sitúan en los extremos del cilindro. En la figura 5.10 se pueden observar juntos los colisionadores tanto del ensamble de vástago – cabezal como los del cilindro mostrados en las figuras 5.8 y 5.9 respectivamente. Podemos observar que el colisionador mostrado a la izquierda en la figura 5.8, del ensamble vástago – cabezal, está situado en medio de los colisionadores extremos del cilindro, mostrado en la figura 5.9. De esta manera cuando el ensamble vástago – cabezal se mueve junto con sus colisionadores, el colisionador del ensamble situado entre los colisionadores extremos del cilindro, choca al llegar al final de su carrera por lo que no puede seguir avanzando. Los colisionadores se situaron por encima del cilindro para evitar que el colisionador situado en el cabezal choque con los colisionadores del cilindro. La figura 5.11 muestra una vista lateral de los colisionadores del cilindro y del ensamble vástago – cabezal para mostrar otra perspectiva de la localización de los colisionares. Todos los movimientos de los cilindros de los gemelos digitales se controlaron mediante la aplicación de una fuerza y a todos se les agregó colisionadores como los mostrados en la figura 5.11. En algunos casos fue posible colocar los colisionadores dentro del cilindro y en otros casos fue requerido colocarlos por fuera, pero en ambos casos los colisionadores tienen las mismas funciones.

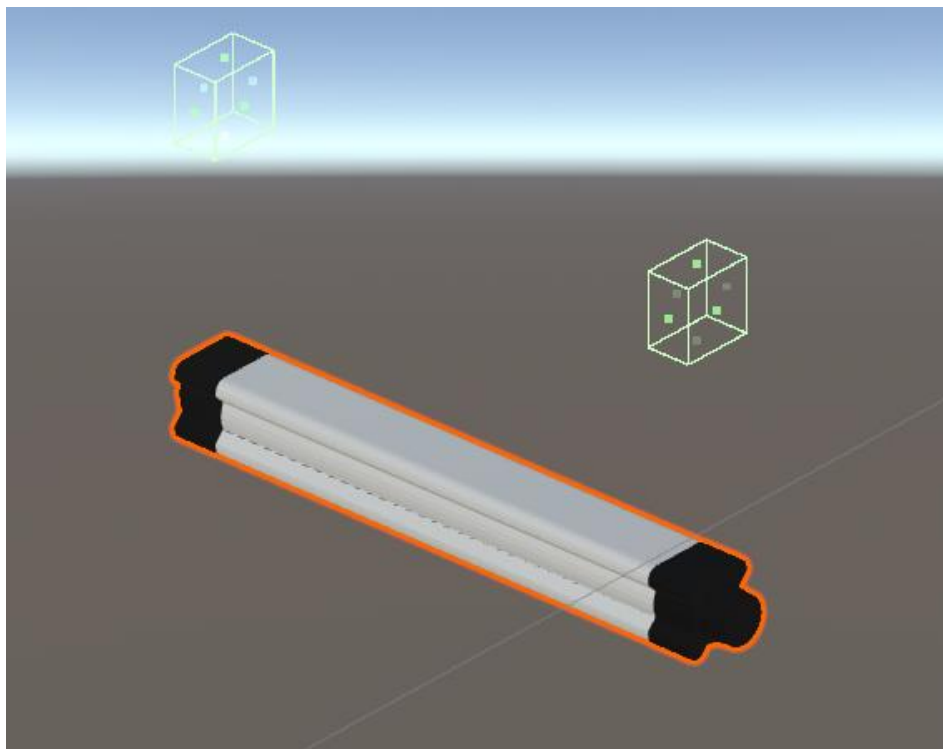


Figura 5.9. Colisionadores atados al cilindro A de la empacadora los cuales sirven para delimitar el movimiento del vástago.

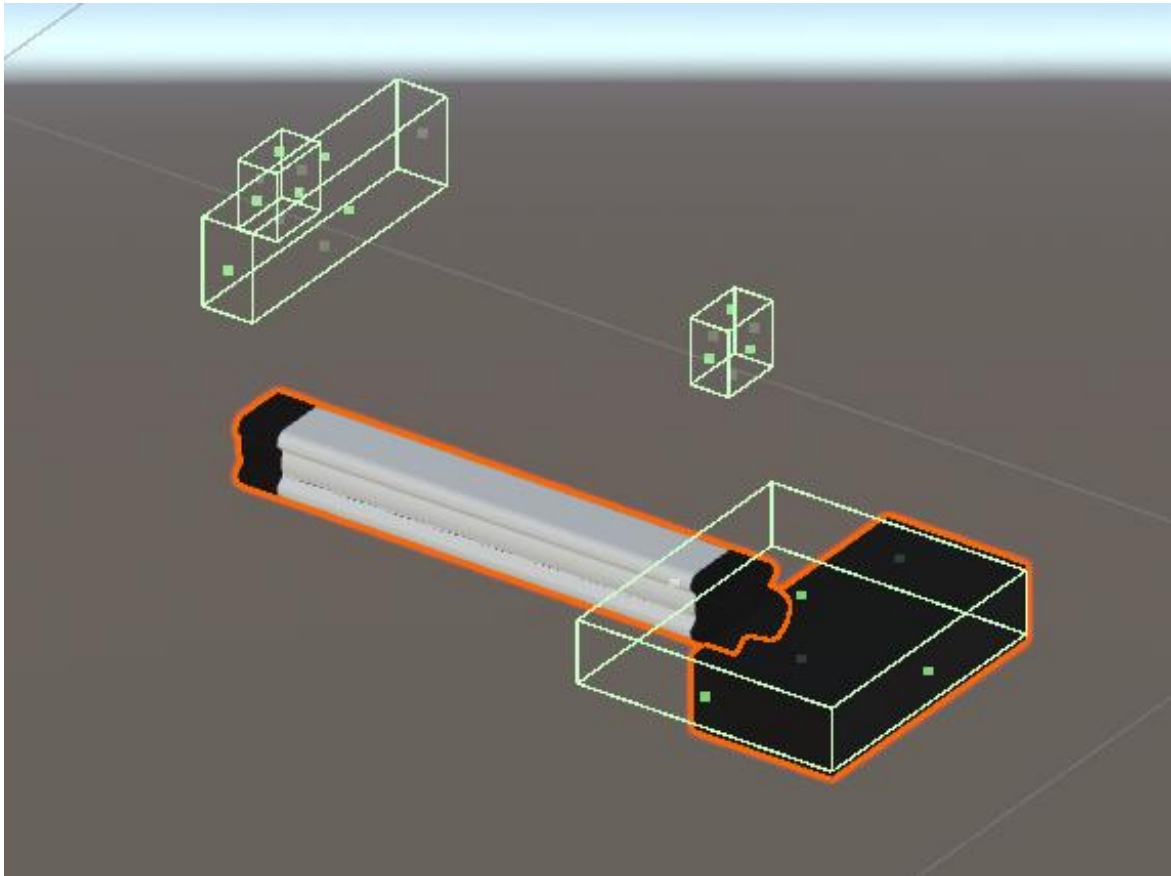


Figura 5.10. Colisionadores del ensamble vástago – cabezal y del cilindro A de la empacadora.

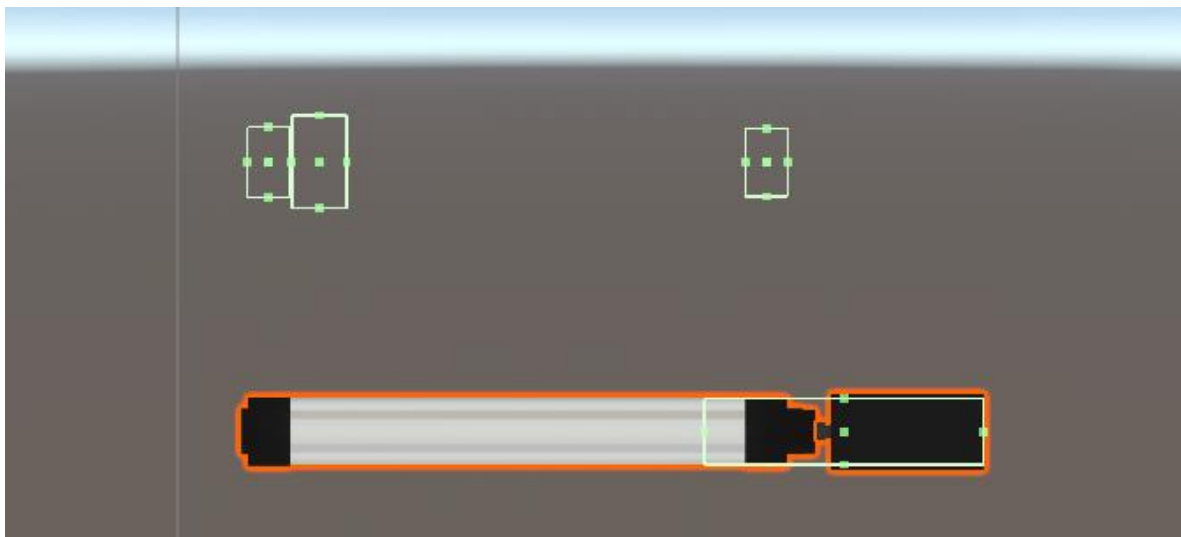


Figura 5.11. Se muestran los colisionadores del ensamble vástago – cabezal y del cilindro. Se delimita la carrera del cilindro con los colisionadores extremos del cilindro (fig 5.9) y el colisionador interno que se mueve entre estos (fig.5.8).

Para lograr el control mediante física se requiere agregar al vástago el componente *Rigidbody*. En la figura 5.12 se muestra el componente *Rigidbody* atado al vástago A de la empacadora y donde se observan las propiedades del componente. Para este ensamble de vástago – cabezal, se eligió una masa de 5 *Kg*, la cual también fue elegida mediante prueba y error. No se especificaron valores de resistencia lineal y angular al aire y tampoco fue necesario hacer uso de la gravedad para controlar el movimiento del vástago, ya que solo se precisa moverlo en una dirección (en la dirección *X*). En vista de esto último, se restringieron los movimientos lineales en las direcciones *Y* y *Z*; así como los movimientos angulares en todas las direcciones, como puede observarse en la misma figura en el parámetro de restricciones (*Constraints*).

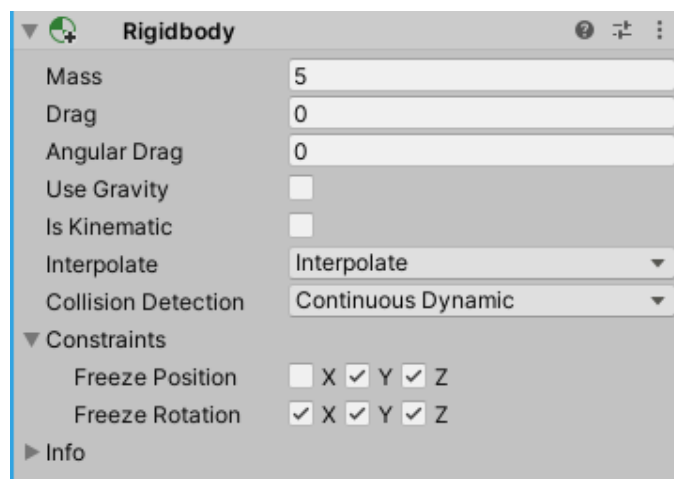


Figura 5.12. Componente *Rigidbody* atado al vástago A de la empacadora.

Para el vástago B de la empacadora se siguió un procedimiento similar al empleado para el vástago A. La figura 5.13 muestra los colisionadores tanto para el vástago como para el cilindro B de la empacadora. Se puede observar que los colisionadores que delimitan el movimiento del vástago se han colocado por encima del cilindro B y que hay un colisionador rodeando el vástago (no solo el cabezal). Esto se hizo de esta manera para recrear la situación en la que los usuarios no realizan la secuencia correcta de movimiento para la empacadora. Si el usuario primero activara el cilindro B y luego el A el paquete se atoraría al chocar con el vástago del cilindro B. Esta es la razón de haber colocado el colisionador sobre el vástago (en las pruebas se verá este caso). Colocar los colisionadores que delimitan el movimiento del vástago por encima del cilindro evita interferencias entre estos colisionadores y el colisionador agregado al vástago. En la figura 5.14 se muestran únicamente los colisionadores del vástago para hacer una distinción con la figura 5.13. También se requirió agregar un componente *Rigidbody* al vástago B de la empacadora para moverlo mediante la aplicación de una fuerza. La figura 5.15 muestra dicho componente y los parámetros elegidos para su funcionamiento, los cuales son semejantes a los elegidos para el vástago A en la figura 5.12, solo cambia la dirección del movimiento y la masa.

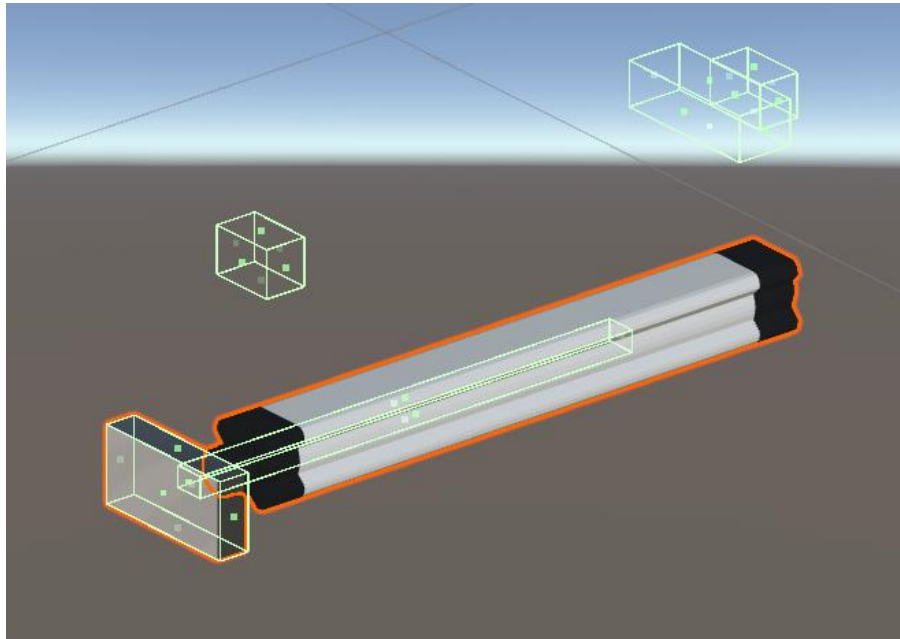


Figura 5.13. Colisionadores atados al vástago y al cilindro B de la empacadora. Los colisionadores superiores delimitan el movimiento del vástago. Los colisionadores inferiores sirven para efectos de colisión con los paquetes.

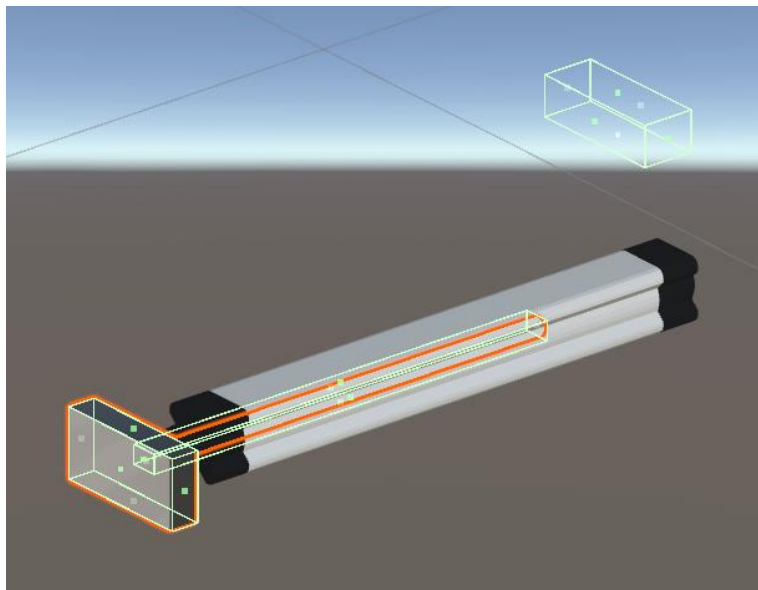


Figura 5.14. Se muestran los colisionadores del vástago del cilindro B de la empacadora. También se muestra el cilindro, pero no se muestran sus dos colisionadores extremos.

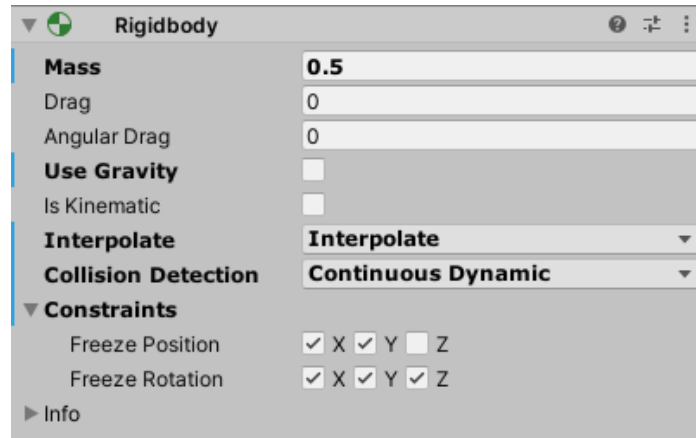


Figura 5.15. Componente *RigidBody* atado al vástago B de la empacadora. Se restringieron todos los movimiento a excepción del movimiento lineal en dirección Z. La masa se eligió arbitrariamente en 0.5 [Kg].

A las piezas de trabajo de la empacadora (paquetes) también se les agregó su colisionador y su componente *RigidBody*. La figura 5.16 muestra el colisionador atado a una de las piezas. La fricción estática de las piezas se estableció en 0.06 y la dinámica en 0.05 (fig. 5.17), estos valores fueron establecidos mediante prueba y error. Se estableció la combinación de fricción como *Average* para promediar las fricciones entre los distintos colisionadores que interactúan. El componente *RigidBody* agregado a los paquetes de la empacadora se muestra en el figura 5.18; donde se puede observar que la masa elegida es de 1 Kg. Para mover masas mayores se requieren fuerzas más grandes. Una fuerza muy grande provoca que los paquetes salgan disparados. Se encontró, después de una serie de pruebas, que una masa de 1[Kg] para los paquetes funciona bien para mantener los movimientos de los cilindros y de los paquetes de manera fluida (sin que las piezas se atoraran) y con la aplicación de fuerzas relativamente bajas (en combinación con las fuerzas de fricción establecidas). Más adelante se mostrarán las fuerzas aplicadas a los vástagos.

En la figura 5.18 también podemos observar que no se tienen restricciones de movimiento para la caja de modo que esta puede moverse libremente dependiendo de las fuerzas que se le apliquen. Para estas piezas de trabajo (paquetes) si fue necesario activar la casilla *Use Gravity* para que se viera afectada por la gravedad. La gravedad por defecto está establecida en 9.81 m/s^2 y no fue cambiada.

Para todos los elementos a los que se les agregó el componente *RigidBody*, se estableció la detección de colisiones en *Continuos Dinamic* y la interpolación en *Interpolate*. Esto fue determinado una vez más mediante prueba y error. Estos modos ofrecían los mejores resultados en el funcionamiento de todos los gemelos digitales. Otros modos provocaban que los paquetes tuvieran comportamientos erráticos en sus movimientos.

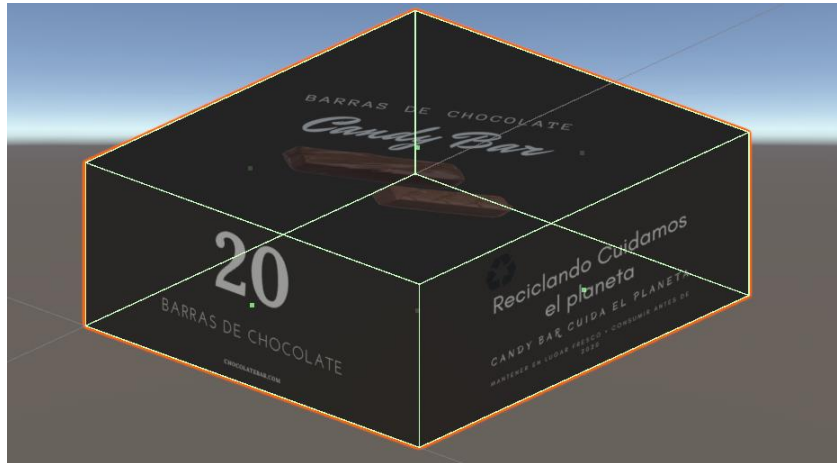


Figura 5.16. Colisionador atado a la caja de la empaedora.

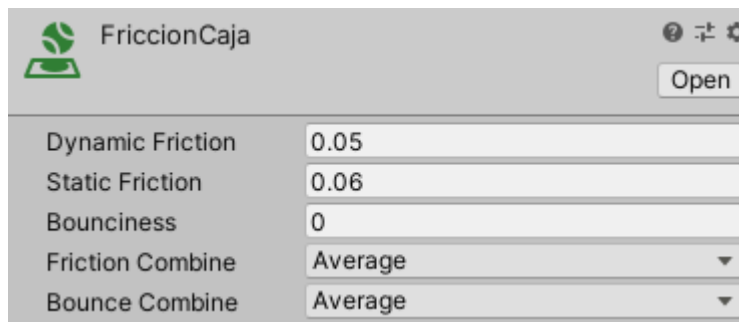


Figura 5.17. *PhysicsMaterial* donde se establece la fricción estática de los paquetes en 0.06 y la fricción dinámica en 0.05.

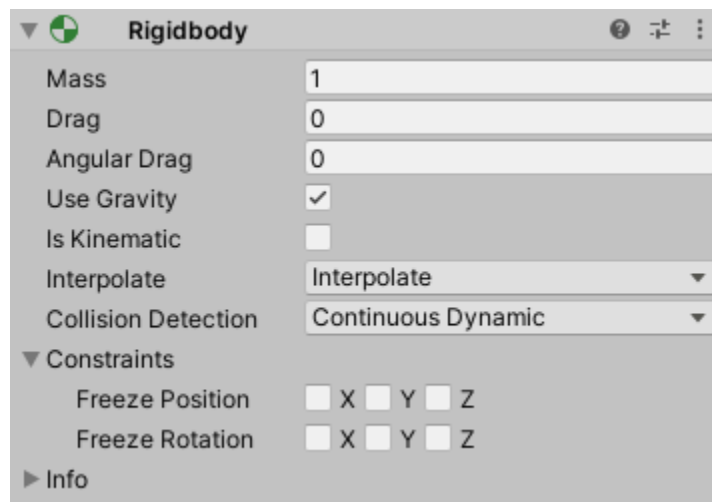


Figura 5.18. Componente *Rigidbody* para los paquetes de la empaedora. Se eligió una masa de 1 [Kg] para todos los paquetes.

Por último, se agregaron colisionadores a la caja contenedora de piezas. La figura 5.19 muestra como quedaron los colisionadores sobre la caja. Estos colisionadores se modificaron después de irlos agregando a la caja ya que en principio estos la cubren por completo. En este caso no importa tanto que los colisionadores sean aproximados al contorno de la caja ya que; en sí, los paquetes no deslizan por sus superficies, simplemente caen en ella. A las tapas de la caja no se les agregó colisionador ya que no eran requeridos. No se especificó ninguna fricción para estos últimos colisionadores agregados.

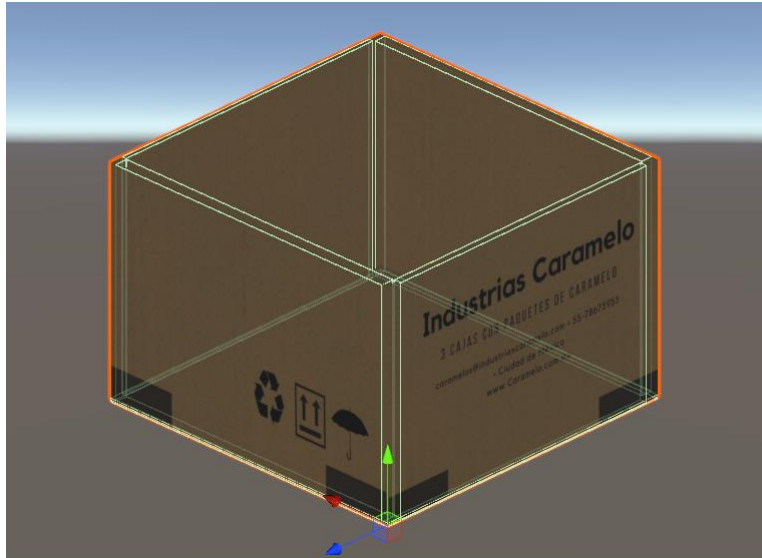


Figura 5.19. Colisionadores agregados a la caja contenedora (no se muestran las tapas).

Una vez que se han agregado los colisionadores (*Colliders*) y los cuerpos Rígidos (*Rigidbody*) a los componentes que lo requieren en el gemelo digital de la empaadora, se procede a programar los movimientos requeridos de los vástagos. Para sincronizar los movimientos de los cilindros de la mesa de trabajo neumática con los cilindros del gemelo digital se utilizó la variable "*VastDobAnimBool*" que es utilizada por el componente *Animator* encargado de la animación de extensión y retracción de los cilindros de la mesa de trabajo (el movimiento de los cilindros de las mesas de trabajo se logra mediante una animación). Cuando la variable *VastDobAnimBool* es verdadera el cilindro se extiende. En caso contrario el cilindro se contrae. La figura 5.20 muestra la llamada a la función que activa el movimiento de los cilindros, donde se puede observar que el cuarto parámetro que se envía a la función es la cadena de caracteres "*VastDobAnimBool*" (líneas 299 y 300). Esta cadena de caracteres identifica la variable del mismo nombre en la máquina de estados que contiene la animación del movimiento de los cilindros de la mesa de trabajo. La figura 5.21 muestra la máquina de estados que contiene la animación de los cilindros de la mesa de trabajo. Cuando la variable *VastDobAnimBool* es verdadera la máquina de estados pasa del estado *Metete_Vastago* al estado *Saca_Vastago*. Cuando la variable *VastDobAnimBool* es falsa la máquina de estados pasa del estado *Saca_Vastago* al estado *Metete_Vastago*.

```

297 // Movimiento de los Cilindros
298
299 ActivarPistonDoble(89, 90, animVastago2, "VastDobAnimBool", sonidoPiston, isActive, botones);
300 ActivarPistonDoble(95, 96, animVastago3, "VastDobAnimBool", sonidoPiston, isActive, botones);

```

Figura 5.20. Llamadas a la función que activa el movimiento de los cilindros de doble efecto.

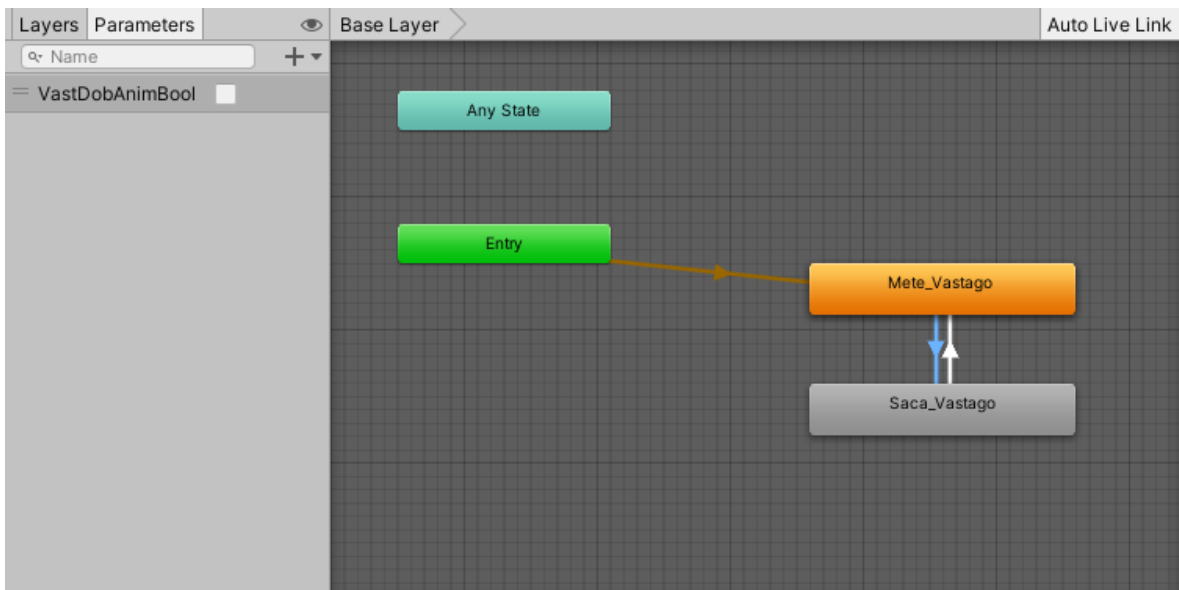


Figura 5.21. Máquina de estados (componente *Animator*) de las animaciones del movimiento de los cilindros de la mesa de trabajo.

El movimiento de los cilindros del gemelo digital se controló mediante la aplicación de una fuerza al vástago. La variable *VastDobAnimBool* se utiliza en este caso, para indicar la dirección de la fuerza que se aplica al vástago. La figura 5.22 muestra el código de la clase “Vastago” empleada en todos los gemelos digitales para mover los vástagos.

En la línea 6 de la figura 5.22 se observa la declaración de la clase “Vastago”.

La línea 8 declara la variable de tipo *GameObject* “*goVastago*” la cual sirve para almacenar el *GameObject* que contiene la animación del movimiento del vástago (cualquiera de los vástagos de las mesas de trabajo). En esta variable se tiene que colocar el vástago de la mesa de trabajo que se quiere que controle el vástago del gemelo digital. Se puede observar que se declaró como una variable pública para que el *GameObject* pueda asignarse desde el inspector (más adelante se verá cómo hacerlo).

La línea 9 declara la variable privada (*private*) “*animVastago*”. Esta variable se utiliza para acceder al componente *Animator* del *GameObject* que contiene la animación del movimiento del cilindro de la mesa. Se declara como privada para que solo este *script* pueda acceder a esta variable.

La línea 11 declara la variable de tipo *Rigidbody* “*vastago*”. Esta variable se utiliza para acceder al componente *Rigidbody* del *GameObject* al cual se le aplicará la fuerza y también es declarada como privada.

```

1  using UdonSharp;
2  using UnityEngine;
3  using VRC.SDKBase;
4  using VRC.Udon;
5
6  public class Vastago : UdonSharpBehaviour
7  {
8      public GameObject goVastago;
9      private Animator animVastago;
10
11     private Rigidbody vastago;
12     public Vector3 Fuerza;
13
14     void Start()
15     {
16         vastago = gameObject.GetComponent<Rigidbody>();
17         animVastago = goVastago.GetComponent<Animator>();
18     }
19
20     void Update()
21     {
22         if (animVastago.GetBool("VastDobAnimBool"))
23         {
24             vastago.AddForce(Fuerza, ForceMode.Force);
25         }
26
27         if (!animVastago.GetBool("VastDobAnimBool"))
28         {
29             vastago.AddForce(-Fuerza, ForceMode.Force);
30         }
31     }
32 }

```

Figura 5.22. Código empleado en los gemelos digitales para mover los vástagos de los cilindros neumáticos.

La línea 12 declara la variable “Fuerza” de tipo *Vector3* la cual sirve para establecer un vector de 3 dimensiones que representará la fuerza aplicada al vástago. Es declarado público para poder modificarlo desde el inspector.

Las líneas 14 a 18 forman parte de la función *Start* que se ejecuta al comienzo de la ejecución del simulador.

La línea 16:

$$vastago = gameObject.GetComponent < Rigidbody > ();$$

asigna el componente *Rigidbody* del objeto *gameObject* a la variable *vástago* (que es de tipo *Rigidbody*). El objeto *gameObject* (con la inicial en minúscula) hace referencia al *GameObject* al que se coloca el script (como componente), de modo que este script *Vastago*

se tienen que añadir como componente del vástago que se quiere controlar en el gemelo digital. El método *GetComponent* es utilizado para acceder al componente que se coloca entre las diples (<>), en este caso el componente *Rigidbody*.

La línea 17:

```
animVastago = goVastago.GetComponent < Animator > ();
```

asigna el componente *Animator* del *GameObject* guardado en la variable *goVastago* a la variable *animVastago* (que es de tipo *Animator*). El *GameObject* que se guarda en la variable *goVastago* debe ser el que contiene la animación del movimiento de los cilindros neumáticos de la mesa de trabajo (que son los vástagos de los cilindros en la mesa de trabajo y se verán más adelante).

Antes de iniciar la ejecución de la simulación se tiene que asignar el *GameObject* “*goVastago*” de lo contrario se mostrará un mensaje de error en la consola de Unity®.

Las líneas 20 a 31 representa la función *Update* la cual se ejecuta un determinado número de veces por cada cuadro (*Frame*) del juego; por lo que, funciona como un ciclo indefinido. En este caso las instrucciones *if* dentro de la función *Update* evalúan constantemente el estado de la variable “*VastDobAnimBool*” para determinar hacia donde debe moverse el vástago en el gemelo digital.

Dentro de la sentencia *if* de la línea 22, se tiene la condición:

```
animVastago.GetBool (“VastDobAnimBool”)
```

En esta condición se obtiene el valor de la variable booleana *VastDobAnimBool* declarada en la máquina de estados (*Animator*) que contiene la animación del movimiento de los cilindros de la mesa de trabajo (el componente *Animator* se guarda en la variable *animVastago*).

La línea 24:

```
vastago.AddForce ( Fuerza, ForceMode.Force );
```

se ejecuta si la instrucción *if* de la línea 22 es verdadera (si la variable *VastDobAnimBool* es verdadera). El método *AddForce* aplica una fuerza al *GameObject* guardado en la variable *vastago* (que corresponde al vástago que se quiere controlar en el gemelo digital). El método *AddForce* tiene dos parámetros. En el primero se especifica el vector fuerza que se quiere aplicar y en el segundo se especifica el tipo de fuerza que se aplicará; en este caso *ForceMode.Force* indica que se aplicará una fuerza constante. Otros modos de fuerza que podrían aplicarse incluyen torcas e impulsos. El método *AddForce* solo funciona si los *GameObjects* a los que se aplica el método tienen atado un componente *Rigidbody*.

En la línea 27, se tiene como condición de la instrucción *if*:

```
! animVastago.GetBool (“VastDobAnimBool”);
```

que es similar a la condición de la instrucción *if* de la línea 22, pero está negada, de modo que la instrucción *if* se ejecutará si la variable *VastDobAnimBool*, en la máquina de estados es falsa.

La línea 29, que se ejecuta si se cumple la condición de la instrucción *if* de la línea 27, utiliza el método *AddForce* para aplicar una fuerza constante al *GameObject* *vastago*, pero en dirección contraria a la fuerza aplicada en la instrucción de la línea 24.

Para el caso del control del movimiento del vástago A de la empacadora, lo primero que se requiere es asignar el *script* *Vastago*, cuyo código es mostrado en la figura 5.22, al vástago A de la empacadora como se muestra en la figura 5.23, donde se observa la selección del vástago A de la empacadora en las ventanas escena (*Scene*) y jerarquía (*Hierarchy*) y en la ventana inspector (*Inspector*) se remarca el componente (*Script Vastago*) agregado.

Una vez asignado el *script* al vástago, se mostrarán (dentro del inspector) las variables que se establecieron como públicas en el *script* (las variables *goVastago* y *Fuerza* en este caso). En la variable *goVastago* se asigna el *GameObject* que contiene el componente *Animator* que controla la animación del cilindro en la mesa de trabajo. La figura 5.24 muestra el *GameObject* que contiene el componente *Animator* requerido, que es el vástago del cilindro A de la mesa de trabajo (ya que se quiere controlar el vástago A del gemelo digital de la empacadora).

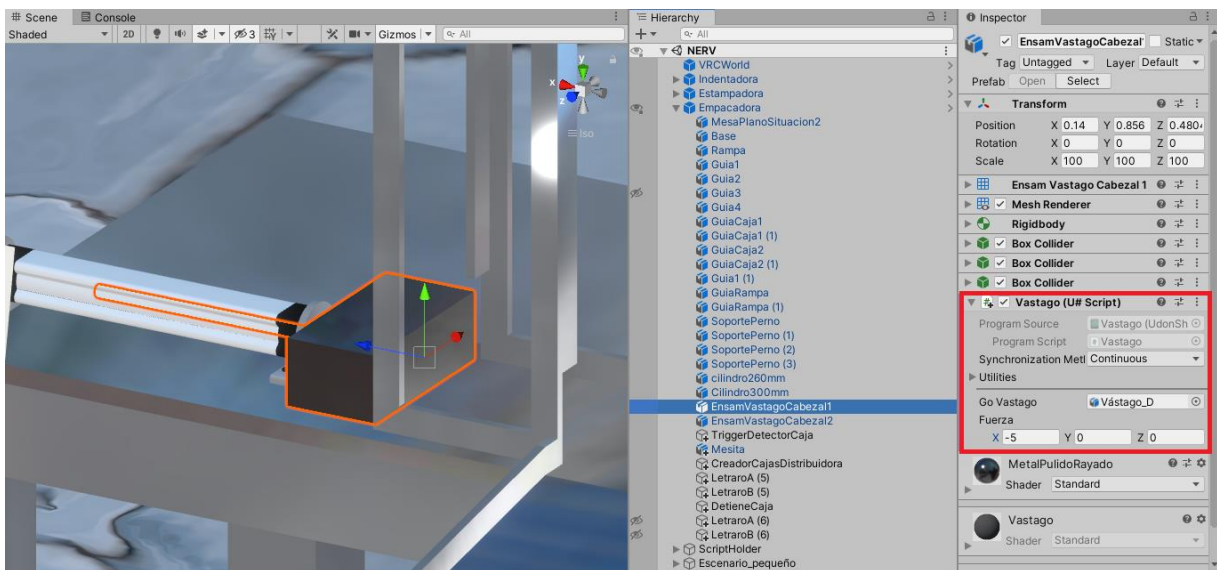


Figura 5.23. Se muestran las ventanas escena, jerarquía e inspector. Se ha seleccionado el vástago A de la empacadora. En la ventana inspector se enmarca el *Script Vastago* que se le ha asignado al vástago A.

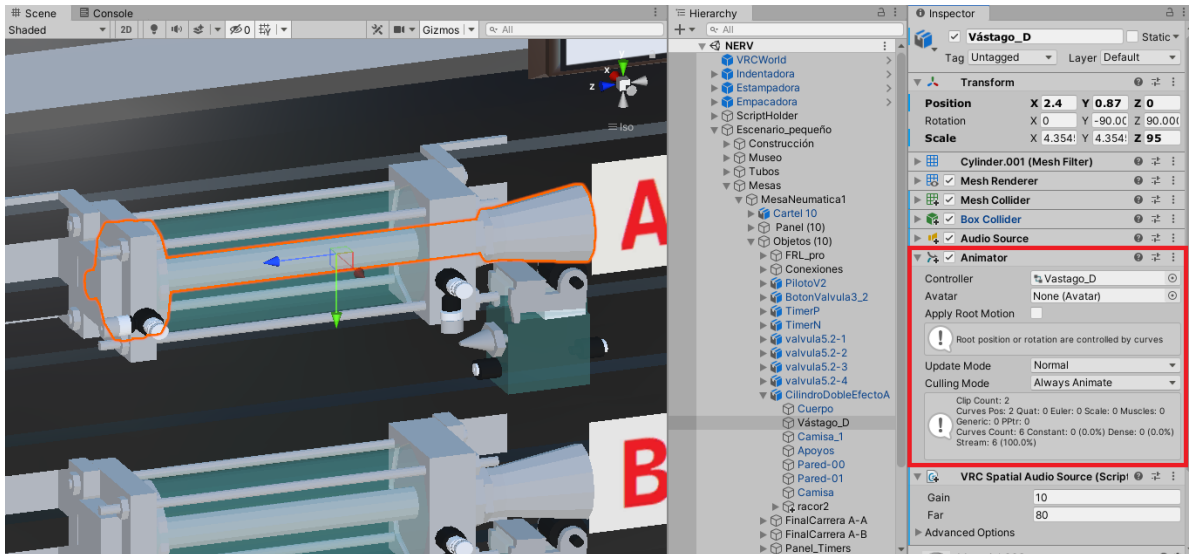


Figura 5.24. Se muestra la ventana escena (*Scene*) y jerarquía (*Hierarchy*) con la selección del *GameObject* (de nombre *Vastago_D*) que contiene el componente *Animator* que controla la animación que mueve el vástago de la mesa de trabajo. En la ventana inspector (*Inspector*) se remarca el componente *Animator* requerido.

La figura 5.25 muestra el componente correspondiente al *Script Vastago* que se remarcó en la figura 5.23, donde se puede observar de mejor manera la asignación del *GameObject Vastago D* a la variable *goVastago* además de que se ha establecido un vector $Fuerza = (-200, 0, 0)[N]$. Para asignar un *GameObject* a una variable pública de un *script* solo se tiene que arrastrar desde la ventana de jerarquía.

El mismo procedimiento se aplica para asignar los *GameObjects* y las fuerzas a los cilindros restantes. Por ejemplo, para el cilindro B de la empacadora, se tiene el *script* mostrado en la figura 5.26. Se observa que se han asignado la variable *goVastago* y también una fuerza. La variable asignada a *goVastago* tiene el mismo nombre que la utilizada para el vástago A; sin embargo, no hace referencia al mismo vástago, aunque tengan nombres iguales por lo que se tiene que tener cuidado en la asignación para que se haga referencia al vástago correcto (en este caso *Vastago_D* hace referencia al vástago B de la mesa de trabajo). Como Unity maneja código orientado a objetos, cuando se hizo una copia del cilindro de doble efecto original, también se copiaron sus animaciones asociadas que, aunque tengan el mismo nombre, cada una funciona independientemente de las copias. En la última figura también se asignó una fuerza que, en este caso, es: $Fuerza = (0, 0, 60)[N]$. Las direcciones de las fuerzas se asignaron de acuerdo a la orientación del gemelo digital.

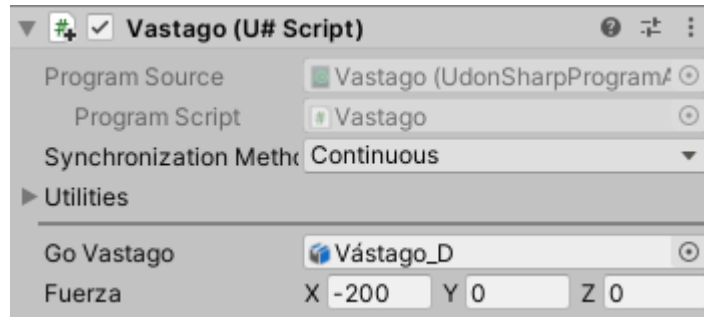


Figura 5.25. *Script* asignado al *GameObject* “*Vastago A*” de la empacadora. En este *Script* se asigna el *GameObject* “*Vástago_D*” a la variable *goVastago* y una fuerza.

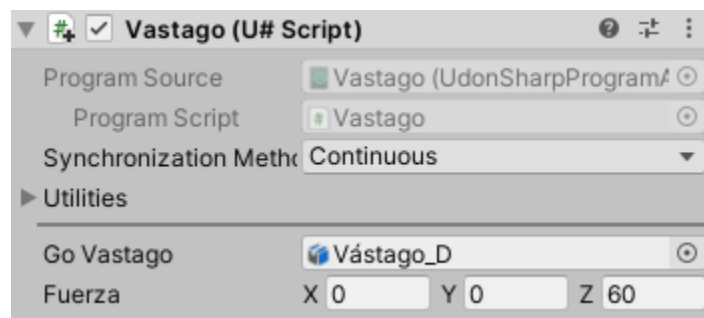


Figura 5.26. *Script* asignado al *GameObject* “*Vastago B*” de la empacadora. En este *Script* se asigna el *GameObject* “*Vástago_D*” a la variable *goVastago* y una fuerza.

Para continuar con la programación de la empacadora se agregaron dos *GameObjects* adicionales al gemelo digital de la empacadora. Estos nuevos *GameObjects* agregados son objetos vacíos y sirvieron para agregarles colisionadores *BoxColliders* de manera que trabajaran como de *Triggers* (detectores). A estos mismos *GameObjects* se les agregó el código que controla cada uno de los *Detectores*. La figura 5.27 muestra ambos detectores. El colisionador (detector) nombrado como *Detector 1* en la figura 5.27, detecta cuando un paquete ha dejado el almacén de piezas. El colisionador situado en la mesita (nombrado *Detector 2*) sirve para detectar las piezas que se empacan en la caja contenedora.

Los piezas que se empacarán, son creadas únicamente si están activados simultáneamente el botón enclavado y la válvula de paso de la unidad de mantenimiento en la mesa de trabajo correspondiente. Una vez activados se crean 8 paquetes (de los 8 paquetes prediseñados, figuras 3.104 y 3.105) y también se crea una caja contenedora donde son almacenadas las piezas (figura 3.110). Una caja nueva será creada, en la parte superior del soporte de almacenamiento de paquetes, cuando uno de los paquetes existentes deje el almacén de piezas (cuando el cilindro A de la empacadora empuje un paquete y este salga del *Detector 1* mostrado en la figura 5.27). Si el circuito neumático es correcto las piezas irán cayendo una por una en la caja contenedora. Una vez hayan caído 3 paquetes en la caja contenedora esta se cerrará, mediante una animación que recrea el cerrado de la caja, y se

destruirá esta caja contenedora junto con las piezas que estén dentro (las piezas que caen en la caja contenedora son detectadas por el *Detector 2* mostrado en la figura 5.27). Para controlar la destrucción de piezas (paquetes), que por alguna razón no hayan llegado al contenedor de piezas, se cuenta con dos arreglos que almacenan las piezas creadas en escena. Las piezas inicialmente creadas y las subsecuentes se almacenan primero en un arreglo. Si un arreglo se termina de llenar, entonces se empiezan a almacenar cajas en el otro arreglo. Antes de que se empiecen a almacenar piezas en un arreglo se eliminan todas las piezas que pudieran estar almacenadas en el arreglo. Esto permite que cualquier pieza que no haya sido destruida junto con el contenedor de paquetes, se destruya después de cierto tiempo, lo que evita que los paquetes puedan quedar en escena y acumularse. Los arreglos tienen suficientes elementos para permitir que siempre existan 8 piezas en el almacén. Cuando el usuario desactiva la unidad de mantenimiento y el botón enclavado; entonces todos los paquetes en escena se destruyen (incluidas las del almacén), se destruye la caja contenedora de paquetes y todas las variables son reiniciadas.

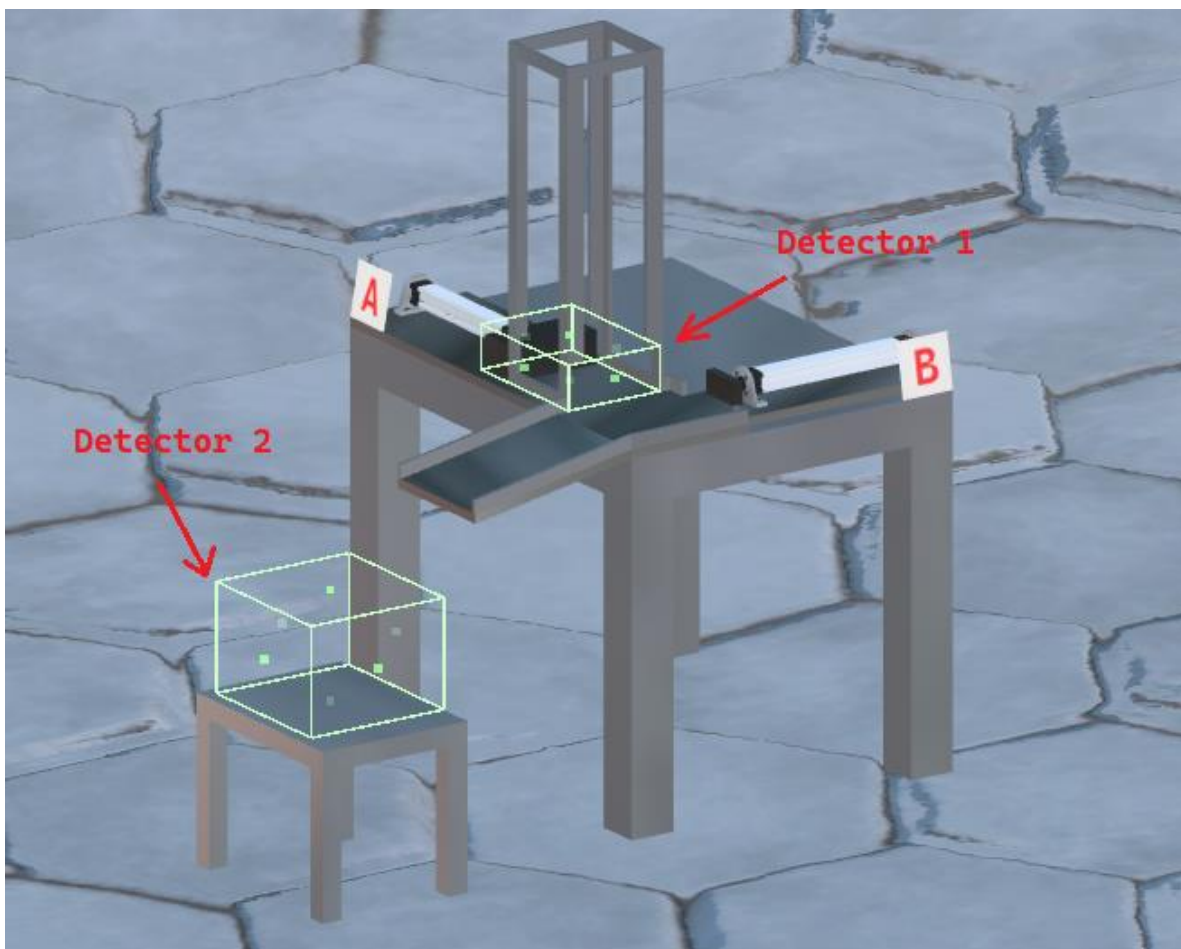


Figura 5.27. Colisionadores agregados a la empaquetadora para hacer funciones de detección (*Triggering*).

Las animaciones del botón enclavado y de la válvula de la unidad de mantenimiento se controlan con las variables “Boton1” y “UMBool” respectivamente. Estas variables están contenidas en las máquinas de estados (componentes *Animator*) que controlan las animaciones de apertura y cerrado del botón enclavado y de la válvula de la unidad de mantenimiento. En la figura 5.28 se muestra la ventana escena donde se puede observar el *GameObject* del botón enclavado. También se observa su selección en el ventana jerarquía lo que permite ver sus componentes en la ventana inspector. En esta última ventana se ha remarcado el componente *Animator* que contiene la animación requerida. La figura 5.29 muestra la máquina de estados correspondiente al componente *Animator* del botón enclavado. Se observa que la variable “Boton1” controla el movimiento de accionamiento del botón enclavado. Cuando la variable “Boton1” es verdadera se acciona el botón enclavado (movimiento de apertura) de lo contrario se desactiva el botón (movimiento de cerrado).

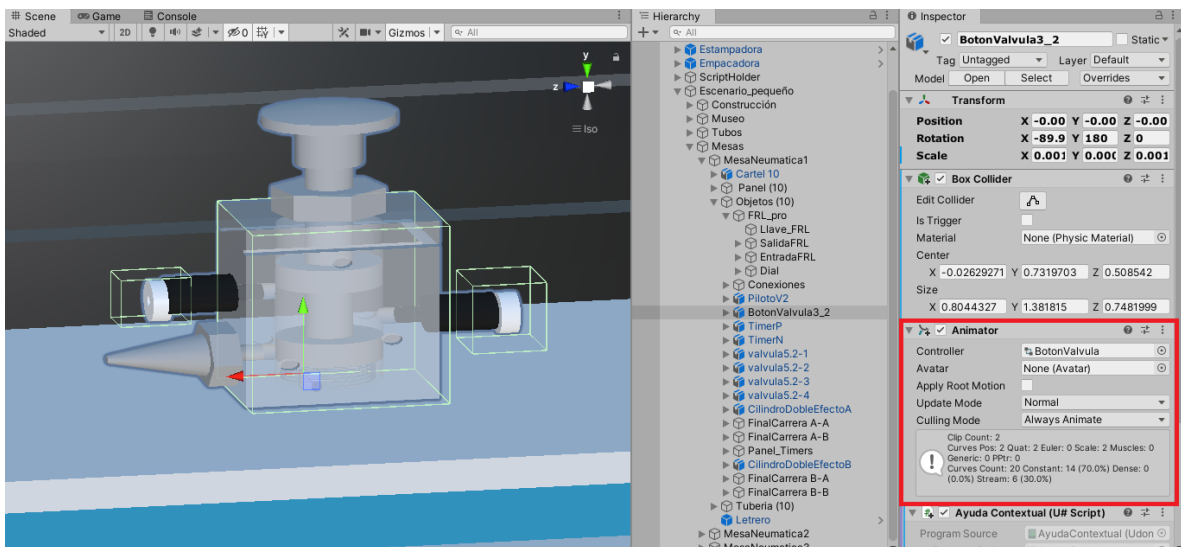


Figura 5.28. Ventanas: escena, donde se muestra el botón enclavado; Jerarquía donde se muestra la selección del botón enclavado; e inspector donde se muestran los componentes del botón enclavado y se remarca el componente *Animator*.

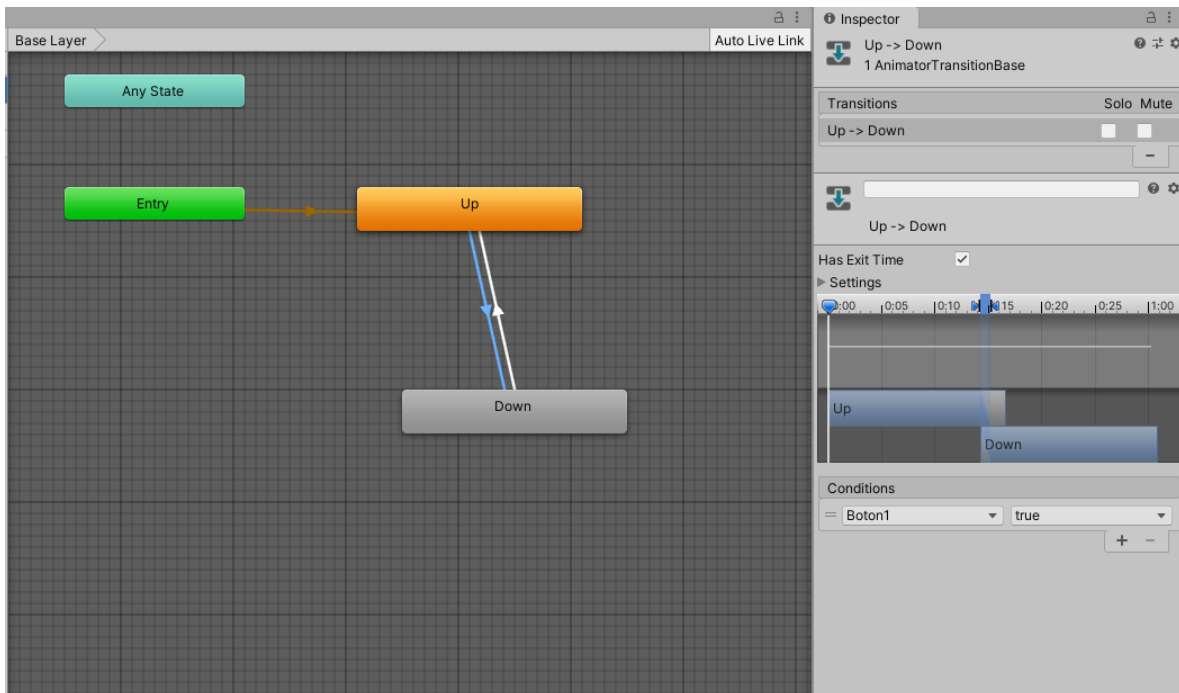


Figura 5.29. Máquina de estados (*Animator*) donde se observa que la variable “*Boton1*” que controla la animación del accionamiento del botón enclavado. Si “*Boton1*” es verdadero el botón se oprime (movimiento de apertura) de lo contrario el botón se suelta (movimiento de cerrado).

La figura 5.30 muestra el *GameObject* que contiene el componente *Animator* que controla la animación de apertura y cerrado de la válvula de la unidad de mantenimiento. Se puede observar que el componente *Animator* está atado solamente a la válvula y no a toda la unidad de mantenimiento. En la máquina de estados mostrada en la figura 5.31 se observa que la variable “*UMBool*” controla la animación de la apertura de la válvula de la unidad de mantenimiento. Si “*UMBool*” es verdadera la válvula se abre en caso contrario la válvula se cierra.

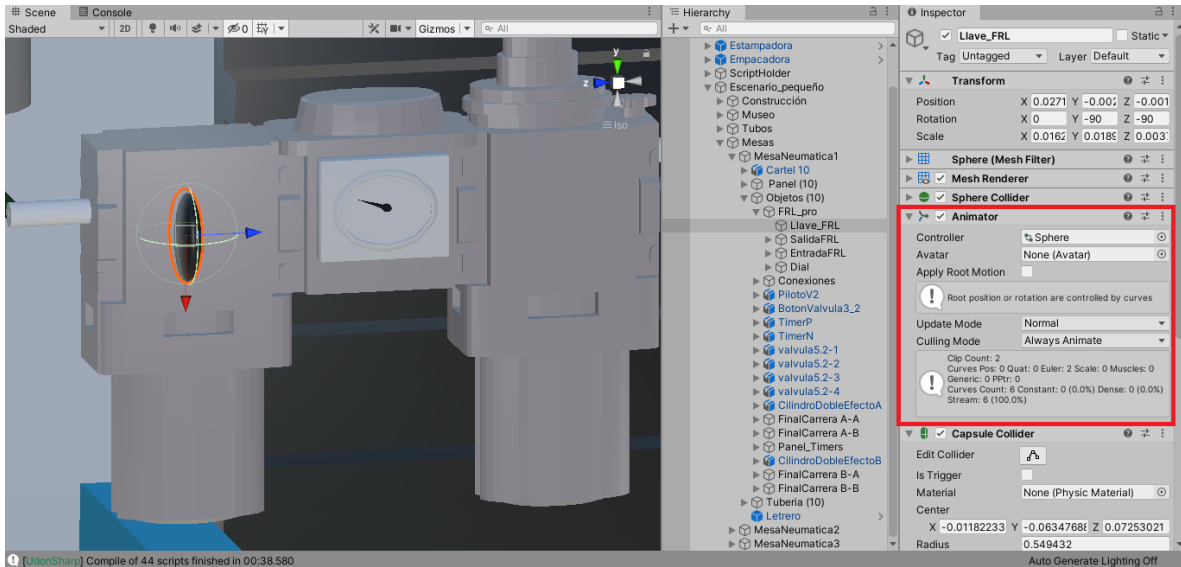


Figura 5.30. Ventanas: escena, donde se observa el *GameObject* que contiene el componente *Animator* que controla la animación de la apertura de la válvula de la unidad de mantenimiento; Jerarquía, donde se observa la selección del *GameObject*; e inspector donde se ha remarcado el componente *Animator*.

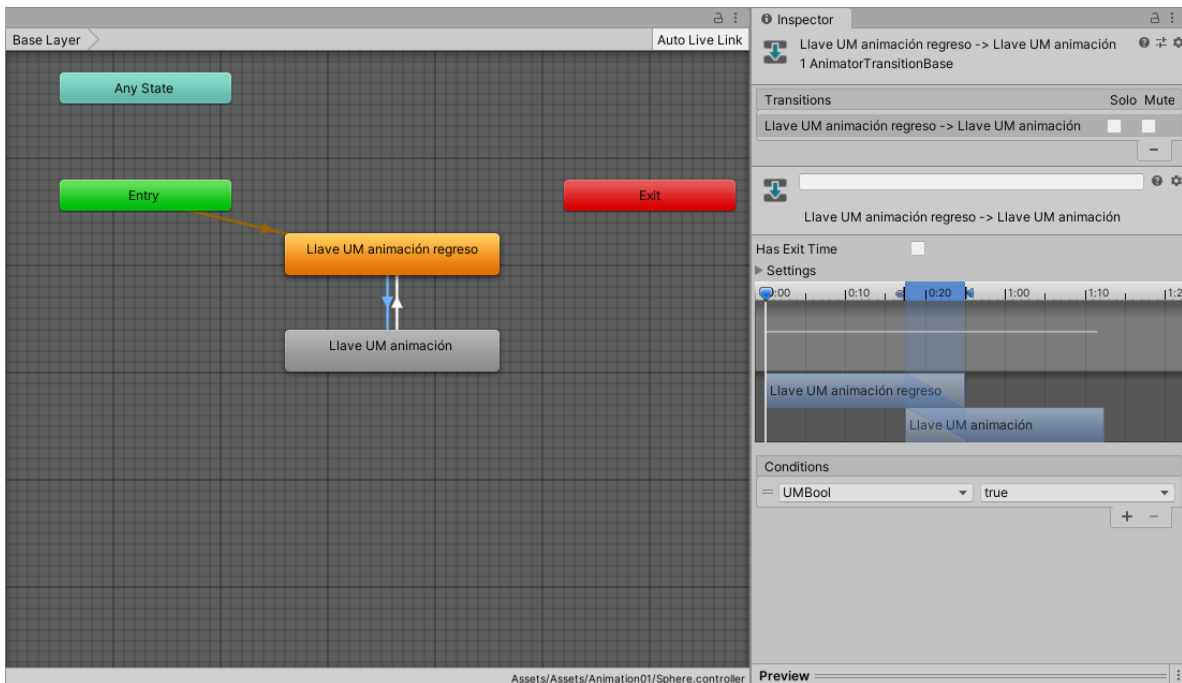


Figura 5.31. Máquina de estados donde se observa que la variable "UMBool" controla la animación del accionamiento de la válvula de la unidad de mantenimiento. Si "UMBool" es verdadero la válvula se abre de lo contrario la válvula se cierra.

En las figuras 5.32 a 5.39 se muestra el código del *script CreaPieza* empleado en el gemelo digital de la empacadora para la creación de las piezas de trabajo (paquetes), así como las cajas contenedoras. Este *script* también es utilizado para la creación de las piezas de trabajo de la estampadora, pues los modos de operación de la empacadora y la estampadora son bastante similares. Algunas líneas de este *script* no son utilizadas en el gemelo digital de la empacadora, y otras no son utilizadas por el gemelo digital de la estampadora. Aquí se hará una descripción detallada del código que abarcará también las partes correspondientes para la operación de la estampadora. En adelante, para la descripción de este código, se llamará a los paquetes de la empacadora y a las piezas de trabajo de la estampadora simplemente “piezas” o “piezas de trabajo”, haciendo notar que se refiere a los paquetes de la empacadora o las piezas de trabajo de la estampadora, según el contexto.

En la figura 5.32, la línea 6 crea la clase “*CreaPiezas*” cuyo cuerpo está formado por las líneas 7 a 216.

La línea 9 crea una variable pública de tipo “*DetectorPiezas*” nombrada “*refDetPiezas*” que se utiliza para acceder a las variables del *script DetectorPiezas*. El *script “DetectorPiezas”* se utiliza para detectar las piezas que han caído a la caja contenedora (más adelante se muestra el código de este *Script*). Las variables que se acceden desde otros *Scripts* deben ser públicas. En este código muchas de las variables se modifican desde otros *scripts* por lo que se declaran como variables públicas. También se hacen públicas las variables que se quieren modificar desde la ventana inspector de Unity. Esto ahorra trabajo cuando se requiere modificar los parámetros constantemente para lograr los resultados esperados.

Las líneas 13 y 14 crean dos variables públicas de tipo *GameObject* nombradas *goBoton* y *goFRL* que almacenan los *GameObjects* donde están contenidos los componentes *Animator* que contienen las animaciones de accionamiento del botón enclavado y de accionamiento de la válvula de la unidad de mantenimiento respectivamente.

La línea 18 crea un arreglo público nombrado “*piezas*” de tipo *GameObject*, que puede ser modificado desde el inspector. En este arreglo se guardan las piezas de trabajo que serán creadas en escena. Para el caso de la empacadora, como se tienen 8 paquetes distintos el arreglo deberá ser de 8, mientras que la estampadora solo tiene una pieza de trabajo por lo que el arreglo será de 1. Las piezas pueden ser creadas directamente con los elementos guardados en este arreglo, sin embargo, no será posible destruirlos (desaparecerlos de la escena). Solo se pueden destruir copias de los elementos y no los elementos directamente. Es por esto que las líneas 22 y 23 crean dos arreglos adicionales de tipo *GameObject* donde se guardarán las copias de las piezas que serán creadas y posteriormente destruidas. Los nombres de los arreglos son “*copiaPiezas*” y “*copiaPiezas2*”. Se crean dos arreglos para controlar el número de piezas en escena (en realidad pudo hacerse un solo arreglo, pero aquí se prefirió trabajar con dos).

La línea 26 crea una variable pública de tipo *GameObject* nombrada “*cajaContenedora*”, que almacena la caja contenedora de paquetes.

La línea 30 es una variable pública de tipo *GameObject* nombrada “*copiaCajaContenedora*”, que almacena una copia del *GameObject* de la caja contenedora para que pueda ser creada en escena y posteriormente ser destruida.

```

1  using UdonSharp;
2  using UnityEngine;
3  using VRC.SDKBase;
4  using VRC.Udon;
5
6  public class CreaPieza : UdonSharpBehaviour
7  {
8      // referencia al script DetectorPieza
9      public DetectorPieza refDetPieza;
10
11     // variables donde se guardan los GameObjects que contienen las animaciones
12     // de la llave de la unidad de mantenimiento y del botón enclavado
13     public GameObject goBoton;
14     public GameObject goFRL;
15
16     // Crea un arreglo de GameObjects donde se almacenaran los prefabs de
17     // las cajas que se empacan
18     public GameObject[] piezas;
19
20     // Crea dos arreglos de GameObjects donde se almacenan copias de los Prefabs
21     // para que puedan ser destruidos
22     public GameObject[] copiaPiezas;
23     public GameObject[] copiaPiezas2;
24
25     // crea un GameObject donde se guarda el prefab de la caja contenedora
26     public GameObject cajaContenedora;
27
28     // crea una GameObject donde se guarda una copia del prefab de
29     // la caja contenedora
30     public GameObject copiaCajaContenedora;
31

```

Figura 5.32. Código de creación de las cajas de la empacadora.

Las líneas 34 y 35 (fig. 5.33) son variables públicas de tipo *Animator* que se utilizan para acceder a los componentes *Animator* de los *GameObjects* del botón enclavado y de la válvula de la unidad de mantenimiento. Se ha nombrado “*animBoton*” y “*animFRL*” respectivamente.

La línea 39 crea una variable pública de tipo booleano llamada *empacadora*. Si la variable es verdadera, se crearán los paquetes correspondientes para el gemelo digital de la empacadora, si es falsa se crearán las piezas de trabajo del gemelo digital de la estampadora. La variable solo puede ser modificada desde el inspector.

La línea 42 crea una variable pública de tipo booleano nombrada “*creaPieza*” que sirve para indicar el momento en el que se debe crear una nueva pieza de trabajo (un paquete para la empacadora o una pieza de trabajo para la estampadora).

La línea 45 crea una variable pública de tipo booleano nombrada “*crearonPiezasInicio*” que se utiliza para indicar que ya han sido creadas las 8 piezas iniciales en el soporte almacenador de piezas (Ambos gemelos digitales, la empacadora y la estampadora, tienen soportes para almacenar sus respectivas piezas de trabajo). Tanto la empacadora como la estampadora, almacenan 8 piezas de trabajo en sus respectivos soportes contenedores.

La línea 48 crea una variable pública de tipo booleano llamada “*creaCajaContenedora*” utilizada para indicar el momento en el que se debe crear una nueva caja contenedora de piezas.

La línea 51 crea dos variables de tipo booleano utilizadas para indicar que arreglo de *GameObjects* (de los 2 creados) está guardando las copias de las piezas de trabajo creadas en escena. Se puede observar que la variable “*activaArreglo1*” inicia verdadera y “*activaArreglo2*” inicia falsa, esto para que inicialmente las cajas creadas empiecen a guardarse en el primer arreglo. La asignación se pudo establecer con una sola variable, pero por claridad se establecieron dos.

La línea 54 crea cinco variables públicas de tipo *Vector3* utilizadas para guardar la posición donde se crean las piezas y la caja contenedora, además de su rotación adecuada (tanto para los paquetes de la empacadora como para las piezas de trabajo de la estampadora). La variable “*posicionPieza*” guarda la posición inicial de creación de piezas (paquetes de la empacadora o piezas de trabajo de la estampadora), las cuales se crean de abajo hacia arriba. La variable “*posicionPieza2*” guarda una segunda posición de creación de piezas, esta vez las piezas se crean desde la parte superior del almacén. La variable “*posCajaContenedora*” guarda la posición donde se crean las cajas contenedoras de piezas. La variable “*rotPieza*” guarda el vector de rotación adecuado para la pieza y la variable “*rotCajaContenedora*” el vector rotación adecuado para la caja contenedora.

La línea 57 crea una variable pública de tipo entero nombrada “*contadorArreglo*” inicializada en cero. Se utiliza para contar las piezas creadas en cada uno de los arreglos que guardan las piezas en escena.

La línea 60 crea una variable pública de tipo entero nombrada “*contadorPaquetes*” utilizada para contar el número de paquetes que han sido creados entre los 8 disponibles (Esta variable solo se utiliza si la variable empacadora es verdadera). Se va creando un nuevo paquete de los 8 que se tienen disponibles con el incremento del contador.

La línea 63 es una variable pública de tipo *float* nombrada “*timer*” creada para funciones de temporizador.

```

32 // Variables para acceder a componentes Animator de
33 // la unidad de mantenimiento y del botón enclavado
34 public Animator animBoton;
35 public Animator animFRL;
36
37 //variable que indica si el script está activado para la empacadora
38 // o para estampadora (si empacadora entonces true, si estampadora entonces false)
39 public bool empacadora;
40
41 // variable que indica cuando se deben crear nuevas cajas
42 public bool creaPieza = false;
43
44 // variable que indica que se han creado las cajas iniciales
45 public bool crearonPiezasInicio = false;
46
47 // variable para la creación de la caja contenedora
48 public bool creaCajaContenedora = false;
49
50 // variables para activar los contadores de las cajas en escena
51 public bool activaArreglo1 = true, activaArreglo2 = false;
52
53 // Guarda las posiciones donde se crean las piezas y las rotaciones de las piezas
54 public Vector3 posPieza, posPieza2, posCajaContenedora, rotPieza, rotCajaContenedora;
55
56 // contadores Cajas en escena
57 public int contadorArreglo = 0;
58
59 // contador de cajas que indica cual caja es creada en el caso de la empacadora
60 public int contadorPaquetes = 0;
61
62 // variable para crear un Reloj
63 public float timer = 0f;

```

Figura 5.33. Código de creación de las cajas de la empacadora (continuación).

Las línea 65 (figura 5.34) declara la función *Start* que se ejecuta al inicio de la simulación. Las líneas 70 y 71, dentro de la función *Start*, asignan los componentes *Animator* de los *GameObjects* del botón enclavado y de la válvula de la unidad de mantenimiento a las variables “*animBoton*” y “*animFRL*” respectivamente. Estos componentes *animator* asignados contienen las animaciones que recrean el movimiento de accionamiento del botón enclavado y de la válvula de la unidad de mantenimiento. Antes de la ejecución de la simulación se deben asignar los *GameObjects* que contienen los respectivos componentes *Animator* a las variables “*goBoton*” y “*goFRL*”, arrastrando dichos *GameObject* a las casillas correspondientes en la ventana inspector.

La línea 70:

$$animBoton = goBoton.GetComponent < Animator > ();$$

Utiliza el método *GetComponent* para acceder al componente *Animator* del *GameObject* guardado en la variable *goBoton* y la instrucción a la derecha del signo igual se asigna a la variable *animBotom* que es de tipo *Animator*. La línea 71 asigna el componente *Animator* del *GameObject* guardado en la variable *goFRL* a la variable de tipo *Animator animFRL*.

```

65 void Start()
66 {
67     // Se obtienen los componentes Animator del botón enclavado y
68     // de la válvula de la unidad de mantenimiento y se guardan en las
69     // respectivas variables
70     animBoton = goBoton.GetComponent<Animator>();
71     animFRL = goFRL.GetComponent<Animator>();
72 }
73
74 // Indica lo que ocurre cuando un Collider sale del trigger
75 void OnTriggerExit(Collider colisionador)
76 {
77     creaPieza = true; // Si una pieza deja el Trigger se crea una caja nueva
78 }
79

```

Figura 5.34. Código de creación de cajas de la empacadora (continuación).

La línea 75 declara la función *OnTriggerExit* y las líneas 76 a 78 corresponden al cuerpo de la función. Esta función se ejecuta continuamente a lo largo de la simulación. Cuando un objeto que tenga atado un colisionador (*Collider*) salga del detector (*Trigger*) entonces se ejecutan las sentencias dentro de la función. En este caso cuando una pieza deje el detector, la variable “creaPieza” (línea 77) se hace verdadera. En ese momento se crea una pieza nueva en la parte superior del soporte almacenador de piezas (en la posición guardada en la variable *posPieza2*). Para que el colisionador pueda detectar las cajas se debe activar la casilla “is Trigger” (dentro de los parámetros del componente *BoxCollider*) y también se debe agregar el *script* que contiene la función “*OnTriggerExit*” al *GameObject* que tiene el colisionador en modo de *Trigger*; por lo que, este *script* “*CreaCajas*” se agrega como componente del *GameObject* vacío nombrado *Detector 1* en la figura 5.27.

Las líneas 81 a 214 (figuras 5.35 a 5.39) corresponden al cuerpo de la función *Update* declarada en la línea 80 (fig.5.35), la cual se ejecuta continuamente a lo largo de la simulación.

La línea 83 crea un temporizador para controlar el tiempo de creación de cajas contenedoras (permite un lapso de espera entre la destrucción y la creación de cajas). El método *Time.DeltaTime* se utiliza para contar el intervalo en segundos entre el fotograma actual y el anterior, por lo que la asignación *timer += Time.DeltaTime* lleva la cuenta en segundos desde el primer fotograma hasta el actual. Si se reinicia la variable *timer* en cero, entonces se llevará la cuenta en segundos desde que se realizó el reinicio de la variable hasta el fotograma actual en la simulación.

La línea 85 declara la instrucción *if* encargada de la creación de las piezas iniciales (también incluida la caja contenedora). El cuerpo de dicha sentencia está formado por las líneas 86 a 109. En la línea 85 se establece que si las variables “*Boton1*” y “*UMBool*” son verdaderas y no se han creado las piezas iniciales (*crearonPiezasInicio* es falsa), entonces se ejecutan las sentencias dentro de la instrucción *if*. La condición dentro del *if* de la línea 85:

animBoton.GetBool(“Boton1”)

```

80 void Update()
81 {
82     // Timer
83     timer += Time.deltaTime;
84     // creación de Piezas al inicio del ciclo
85     if (animBoton.GetBool("Boton1") && animFRL.GetBool("UMBool") && !crearonPiezasInicio)
86     {
87         while (contadorArreglo < 8)
88         {
89             // Si esta activada la empacadora
90             if (empacadora)
91                 copiaPiezas[contadorArreglo] = Instantiate(piezas[contadorArreglo],
92                                                             posPieza + contadorArreglo * 0.08f * Vector3.up,
93                                                             Quaternion.Euler(rotPieza));
94
95             // si no está activada la empacadora está activada al estampadora
96             if(!empacadora)
97                 copiaPiezas[contadorArreglo] = Instantiate(piezas[0],
98                                                             posPieza + contadorArreglo * 0.08f * Vector3.up,
99                                                             Quaternion.Euler(rotPieza));
100
101             contadorArreglo++;
102         }
103
104         // crea una caja contenedora en escena
105         copiaCajaContenedora = Instantiate(cajaContenedora, posCajaContenedora,
106                                           Quaternion.Euler(rotCajaContenedora));
107         // se indica que ya han sido creados los elementos iniciales
108         crearonPiezasInicio = true;
109     }
110 }

```

Figura 5.35. Código de creación de cajas de la empacadora (continuación).

obtiene el valor booleano de la variable *Boton1*, que controla la animación del accionamiento del botón enclavado y está guardada en el componente *Animator* almacenado en la variable *animBoton*. Para la condición *animFRL.GetBool("UMBool")* se tiene una interpretación similar, pero para la animación de la apertura de la válvula de la unidad de mantenimiento.

La línea 87 declara el ciclo *while* cuyo cuerpo está formado por las líneas 88 a 102 y que solo se ejecutará si la condición del *if* de la línea 85 es verdadera. El ciclo *while* se repetirá solo hasta que la variable *contadorArreglo* sea igual a 8. El ciclo *while* contiene dos sentencias *if* que se ejecutaran dependiendo de si se está trabajando con la empacadora o con la estampadora. Recordemos que si la variable *empacadora* es verdadera entonces se está trabajando con la empacadora y si es falsa se está trabajando con la estampadora. Si la variable *empacadora* se establece como verdadera (se está trabajando con la empacadora), entonces la sentencia *if* de la línea 90 es verdadera y se ejecutan las líneas 91 a 93 (que forman parte de una sola sentencia):

$$\begin{aligned}
 copiaPiezas[contadorArreglo] = & \mathbf{Instantiate}(piezas[contadorArreglo], \\
 & posPieza + contadorArreglo * 0.08f * Vector3.up, \\
 & Quaternion.Euler(rotPieza));
 \end{aligned}$$

El método *Instantiate* mostrado en la sentencia anterior se encarga de crear el *GameObject* que se especifica en el primer parámetro del método, en este caso *piezas[contadorArreglo]*. En cada una de las posiciones del arreglo *piezas* se guardan los paquetes disponibles para empacar por el gemelo digital de la empacadora (8 paquetes). Más adelante se muestra cómo se asignan estos paquetes (*GameObjects*) al arreglo. La variable *contadorArreglo* comienza en cero y se incrementa en uno, por lo que al final de 7 iteraciones se han creado 8 piezas distintas (paquetes). El segundo parámetro del método *Instantiate* especifica la posición donde se creará el *GameObject*. En este caso se tienen las posiciones:

$$posPieza + contadorArreglo * 0.08f * Vector3.up$$

Como *contadorArreglo* comienza en cero la primera posición la especifica el vector *posPieza*. Después para ciclos posteriores se incrementa proporcionalmente la altura de la creación de las piezas en un valor de 0.08 unidades (la letra *f* indica que es un valor float). Este valor de proporcionalidad corresponde a un valor ligeramente mayor a la altura de las piezas (paquetes). *Vector3.up* especifica un vector unitario en dirección Y y sentido positivo (equivalente al vector (0, 1, 0)). De este modo la creación de las piezas se hace una por encima de otra hasta completar los 8 modelos de paquetes disponibles. Por último, el tercer parámetro de la instrucción *Instantiate* corresponde a la rotación de las piezas. En este caso:

$$Quaternion.Euler(rotPieza)$$

La variable *rotPieza* especifica el vector rotación de tres dimensiones. Aún no se han especificado los valores de los vectores de posición y de rotación, esto se hace más adelante. Estos tres parámetros de la instrucción *Instantiate* especifican por completo dónde y cómo se crean las cajas. El método *Instantiate* se asigna a los componentes del arreglo *copiaPiezas*, de manera que las piezas se van almacenando en este primer arreglo como copias de las piezas guardadas en el arreglo *piezas*.

Si la variable empacadora es falsa, entonces la instrucción *if* que se ejecuta es la de la línea 96 que contiene una instrucción similar a la mostrada anteriormente que crea las piezas para la empacadora. En este caso se tiene:

$$\begin{aligned} copiaPiezas[contadorArreglo] &= Instantiate(piezas[0], \\ &posPieza + contadorArreglo * 0.08f * Vector3.up, \\ &Quaternion.Euler(rotPieza)); \end{aligned}$$

Como puede observarse solo cambio el primer parámetro del método *Instantiate* que ahora es: *piezas[0]*, lo que indica que solo se considera el primer elemento del arreglo *piezas*. Se verá más adelante que el tamaño del arreglo *piezas* puede especificarse desde el inspector, por lo que para el caso de la empacadora se especifica un tamaño de 8, correspondiente a las 8 piezas disponibles para empacar, y para la estampadora se especifica en 1, ya que solo se cuenta con una pieza para estampar.

Cualquiera que sea el caso la variable *contadorArreglo* de la línea 101 se incrementa en 1 por cada iteración del ciclo. Recordemos que esta variable cuenta el número de piezas en los arreglos donde se guardan las piezas instanciadas; que en este ciclo se guardan en el arreglo 1, por lo que al final del ciclo habrá un total de 8 piezas almacenadas en el arreglo 1.

La línea 105, que sigue perteneciendo a la sentencia *if* declarada en la línea 85, crea una caja contenedora de piezas con el método *Instantiate*. Como solo se tiene disponible una caja contenedora no se especifica un arreglo de *GameObjects* sino un elemento individual tanto para la caja "original" como para la copia (la asignación).

La línea 108 indica que ya se han creado las piezas iniciales con las que trabajará la empacadora o la estampadora; según sea el caso, y por lo tanto no se volverá a acceder a la instrucción *if* de la línea 85 hasta que se reinicien las variables.

La línea 112 (figura 5.36) declara una instrucción *if* cuyo cuerpo lo conforman las líneas 113 a 120. Dentro de esta instrucción se crean las subsecuentes cajas contenedoras de piezas. Una nueva caja contenedora será creada si la variable "*creaCajaContenedora*" es verdadera y el temporizador es mayor a 2.5 segundos. Las líneas 115 y 116, que forman una sola sentencia, se encargan de crear la caja contenedora con el método *Instantiate*. La línea 119 hace la variable "*creaCajaContenedora*" falsa para que solo se ejecute una vez la instrucción *if*. La instrucción se volverá a ejecutar cuando la variable "*creaCajaContenedora*" vuelva a ser verdadera (esto se hará desde otro *script*).

La línea 124 es otra instrucción *if* cuyo cuerpo está formado por las líneas 125 a 164. La instrucción *if* se ejecuta si tanto la variable "*creaPieza*" como la variable "*activaArreglo1*" son verdaderas. Las piezas que se crearon inicialmente son guardadas en el arreglo *copiaPiezas*. Las piezas restantes serán guardadas en este mismo arreglo hasta que se complete un total de 12 piezas en el arreglo (este valor se eligió arbitrariamente y posteriormente se mostrará la asignación). La línea 128 establece la variable *creaPieza* como falsa para que la instrucción solo se ejecute una vez hasta que *creaPieza* vuelva a ser verdadera.

La línea 131 (dentro de la instrucción *if* de la línea 124) es una instrucción *if* compuesta por las líneas 132 a 137. Esta instrucción se ejecutará únicamente si la variable *empacadora* es verdadera (si se está trabajando con la empacadora). Las líneas 134 y 135 dentro de la instrucción *if* se encargan de crear una nueva pieza (paquete) en la parte superior del almacén de piezas (en la posición guardada en la variable *posicionCaja2*). La línea 136 incrementa el contador "*contadorPaquetes*" en uno. Con el incremento de este contador se van creando cada una de las piezas (paquetes) que se tienen disponibles y que se almacenaron en el arreglo *piezas* (se tienen 8 paquetes distintos para trabajar con la empacadora).

La línea 140 (dentro de la instrucción *if* de la línea 124) declara una instrucción *if* que se ejecutará si la variable *empacadora* es falsa (si la estampadora está activada). Las líneas 141 a 145 forman el cuerpo de esta instrucción. Dentro de ella, las líneas 143 y 144 crean una nueva pieza de trabajo para la estampadora. En este caso no se tiene un contador ya que solo se cuenta con una pieza de trabajo.


```

111 // Si se debe crear una nueva caja contenedora y el tiempo es mayor a 2.5 unidades
112 if (creaCajaContenedora && timer > 2.5f)
113 {
114     // Se crea una nueva caja contenedora
115     copiaCajaContenedora = Instantiate(cajaContenedora, posCajaContenedora,
116                                     Quaternion.Euler(rotCajaContenedora));
117     // Se indica que se debe salir de la instrucción para que no se creen
118     // más cajas contenedoras
119     creaCajaContenedora = false;
120 }
121
122 // Si se indica que se debe crear una nueva pieza y está activa el
123 // almacenamiento en el arreglo 1
124 if (activaArreglo1 && creaPieza)
125 {
126     // Solo debe ejecutarse una vez la instrucción if, hasta que se indique de nuevo
127     // que debe crearse una pieza
128     creaPieza = false;
129
130     // si está activada la empacadora
131     if (empacadora)
132     {
133         // Se crea un nuevo paquete
134         copiaPiezas[contadorArreglo] = Instantiate(piezas[contadorPaquetes], posPieza2,
135                                                  Quaternion.Euler(rotPieza));
136         contadorPaquetes++;
137     }
138
139     // si está activada la estampadora
140     if (!empacadora)
141     {
142         // se crea una nueva pieza de trabajo
143         copiaPiezas[contadorArreglo] = Instantiate(piezas[0], posPieza2,
144                                                  Quaternion.Euler(rotPieza));
145     }

```

Figura 5.36. Código de creación de cajas de la empacadora (continuación).

Dentro del mismo cuerpo de la instrucción *if* de la línea 124, la línea 149 (fig. 5.37) incrementa la variable *contadorArreglo* en uno. Recordemos que esta variable lleva la cuenta de las piezas que se han creado en los arreglos que guardan las copias de las piezas de trabajo de la empacadora y de la estampadora.

La línea 154 declara una nueva instrucción *if*, que se ejecuta cuando la variable *contadorArreglo* es igual al número de elementos del arreglo *copiaPiezas*. Dentro del cuerpo de esta instrucción (líneas 155 a 163), se declara un ciclo *for* que se ejecuta desde cero hasta el valor del tamaño del arreglo *copiaCajas* en incrementos de uno. Dentro del ciclo *for* se destruyen las copias de las piezas almacenadas en el arreglo *copiaPiezas2*. Se debe notar que se eliminan las piezas del siguiente arreglo y no las del que recientemente ha estado almacenando las piezas. Esto permite que las piezas permanezcan en escena hasta que se depositen en la caja contenedora o después de cierto tiempo si las cajas, por alguna razón, no llegaron a su destino. Una vez eliminadas las piezas se declara que la variable *activaArreglo1* se hace falsa y la variable *activaArreglo2* se hace verdadera; con esto el primer arreglo deja de almacenar piezas y el segundo empieza a almacenarlas (líneas 160 y 161). La línea 162 reinicia la variable *contadorArreglo* a cero, para empezar a contar las piezas creadas en el siguiente arreglo.

```

146
147 // Aumenta en uno el contador que lleva la cuenta de las piezas que se crean en
148 // los arreglos
149 contadorArreglo++;
150
151 // Si el contadorArreglo es igual a número de espacios del arreglo
152 // se destruyen las piezas del siguiente arreglo 2(no del actual) y se activa el
153 // almacenamiento en el arreglo 2. El contador se reinicia en cero
154 if (contadorArreglo == copiaPiezas.Length)
155 {
156     for (int i = 0; i < copiaPiezas2.Length; i++)
157     {
158         Destroy(copiaPiezas2[i]);
159     }
160     activaArreglo2 = true;
161     activaArreglo1 = false;
162     contadorArreglo = 0;
163 }
164
165

```

Figura 5.37. Código de creación de cajas de la empacadora (continuación).

La línea 168 (fig. 5.38) declara una nueva instrucción *if* bastante similar a la de la línea 124. Solo hay algunos pequeños cambios. Esta nueva instrucción se ejecutará si se ha indicado que se cree una pieza nueva (*creaPieza* es verdadera) y si está activada la variable *activaArreglo2* (si la variable *activaArreglo2* es verdadera) por lo que las piezas ahora se guardan en el segundo arreglo. Este cambio se observa en las líneas 178 y 185 (respecto de las líneas 134 y 143) donde ahora la asignación de las piezas creadas se hace en el arreglo *copiaPiezas2* y no en el arreglo *copiaPiezas*.

```

166 // Si se debe crear una nueva pieza y está activo el almacenamiento
167 // en el arreglo 2
168 if (activaArreglo2 && creaPieza)
169 {
170     // Solo debe ejecutarse una vez la instrucción if, hasta que se indique de nuevo
171     // que debe crearse una pieza
172     creaPieza = false;
173
174     // si está activa la empacadora
175     if (empacadora)
176     {
177         // crea un nuevo paquete
178         copiaPiezas2[contadorArreglo] = Instantiate(piezas[contadorPaquetes], posPieza2,
179                                                     Quaternion.Euler(rotPieza));
180         contadorPaquetes++;
181     }
182     // Si está activa la empacadora
183     if (!empacadora)
184     {
185         copiaPiezas2[contadorArreglo] = Instantiate(piezas[0], posPieza2,
186                                                     Quaternion.Euler(rotPieza));
187     }
188     // Aumenta en uno el contador que lleva la cuenta de las piezas que se crean en
189     // los arreglos
190     contadorArreglo++;
191

```

Figura 5.38. Código de creación de cajas de la empacadora (continuación).

Otro cambio se aprecia en la línea 200 (fig. 5.39) donde ahora las piezas que se destruyen son las copias almacenadas en el arreglo *copiaPiezas* y no las del arreglo *copiaPiezas2* que es el que está actualmente almacenando las piezas. Las líneas 202 y 203 cambian para dar pie a la creación de piezas en el arreglo *copiaPiezas* y que se dejen de crear en el arreglo *copiaPiezas2*.

```
192 // Si el contadorArreglo es igual a número de espacios del arreglo
193 // se destruyen las piezas del siguiente arreglo 1 (no del actual) y se activa el
194 // almacenamiento en el arreglo 1. El contador se reinicia en cero
195
196 if (contadorArreglo == copiaPiezas.Length)
197 {
198     for (int i = 0; i < copiaPiezas.Length; i++)
199     {
200         Destroy(copiaPiezas[i]);
201     }
202     activaArreglo2 = false;
203     activaArreglo1 = true;
204     contadorArreglo = 0;
205 }
206
207
208 // si está activa la empacadora y el contador de paquetes es igual a 8
209 // se reinicia la cuenta en 0
210 if (contadorPaquetes == 8 && empacadora)
211 {
212     contadorPaquetes = 0;
213 }
214
215 }
```

Figura 5.39. Código de creación de cajas de la empacadora (continuación).

Se recalca que el *script CreaPieza* se agregó como componente del *GameObject Detector 1* mostrado en la figura 5.27 y que al colisionador de este *GameObject* se le activó el parámetro *is Triggering* ya que; el *script CreaPieza* tiene una función *OnTriggerExit* (fig. 5.34) que requiere que el *script* donde fue declarada esta función (el *script CreaPieza*), se agregue como componente del *GameObject* cuyo colisionador se quiere utilizar como detector.

Una vez agregado el *script CreaPieza* como componente del *GameObject Detector 1*, se pueden observar los parámetros del *script* dentro de la ventana del inspector. En esta ventana, que se muestra en la figura 5.40, se observa que aparecen todas las variables que se definieron como públicas dentro del *script*. En principio los espacios correspondientes a las variables aparecen vacíos, pero en esta imagen ya se han llenado algunos espacios con los *GameObjects* requeridos. Se vuelve a recalcar que aquí se agregan *GameObjects* aunque las variables; sean por ejemplo, de tipo *Animator* o *Rigidbody*, Unity se encargará de buscar los componentes requeridos dentro de los *GameObjects*. La primera variable declarada en el *scrip CreaPieza* fue la variable de tipo *DetectorPieza*, *refDetPieza*. Esta variable se puede observar en la parte superior de la figura 5.40. En esta ventana ya se agregó el *GameObject* que contiene el *script DetectorPieza* y de manera similar se hizo para las variables *goBoton*

y *goFRL* (Todavía no se ha descrito el *script Detector Pieza*, pero este *script* se agregó como componente del *Detector 2* mostrado en la figura 5.27).

Las variables *Piezas*, *copiaPiezas* y *copiaPiezas2* que son arreglos, tienen una flecha a su costado izquierdo que indica que se pueden expandir mostrando los espacios disponibles de los arreglos. La figura 5.41 muestra los arreglos mencionados una vez que se expanden para ver el número de sus elementos. El tamaño de los arreglos se puede especificar desde el recuadro *Size*. Para el caso del arreglo *Piezas* se especificó en 8 (ya que se tienen 8 paquetes distintos), y para los arreglos *copiaPiezas* y *copiaPiezas2* en 12. En la figura 5.41 se puede observar que ya se han agregado los 8 *GameObjects* (de los paquetes) al arreglo *Piezas*. También se pueden observar los espacios disponibles para el arreglo *copiaCajas*. De manera similar aparece el arreglo *copiaCajas2*, pero aquí no se ha expandido el arreglo para no hacer la figura más grande. Refiriéndonos de nuevo a la figura 5.40, también se agregó el *GameObject* de la caja contenedora, la cual ya contiene la animación para cerrarse. Las demás variables que aparecen como *None* (tanto en la figura 5.40 como en la 5.41) se inicializan durante la ejecución de la simulación, en la función *Start*, o durante la ejecución de la simulación, en la función *Update*. La variable *empacadora* debe estar activa (con la palomita que indica verdadero) para que funcione la creación de los 8 paquetes, en caso contrario solo se creará un solo paquete (que sería el caso que se requiere en la estampadora). Se observa, además, que la variable *ActivaArreglo1* inicia verdadera (palomeada) y la variable *activaArreglo2* falsa (sin paloma), como se especificó dentro del *script*. Los vectores de posición y de rotación para la creación y orientación de los paquetes también se inicializan. Estos valores se obtienen creando un paquete en escena en la posición y orientación requeridas y tomando sus valores desde el componente *Transform*.

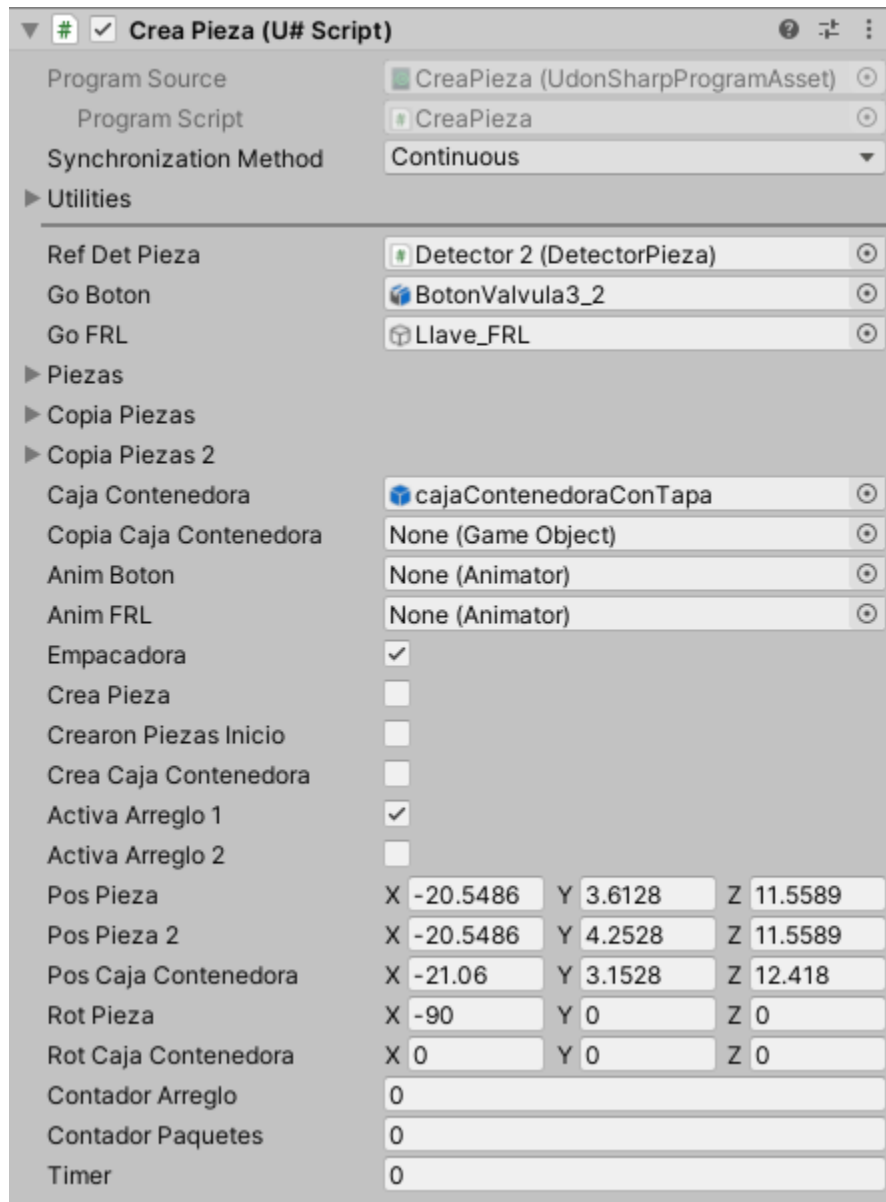


Figura 5.40. Script *CreaPiezas* agregado al *GameObject Detector 1* y cuyo colisionador es mostrado en la figura 5.27.

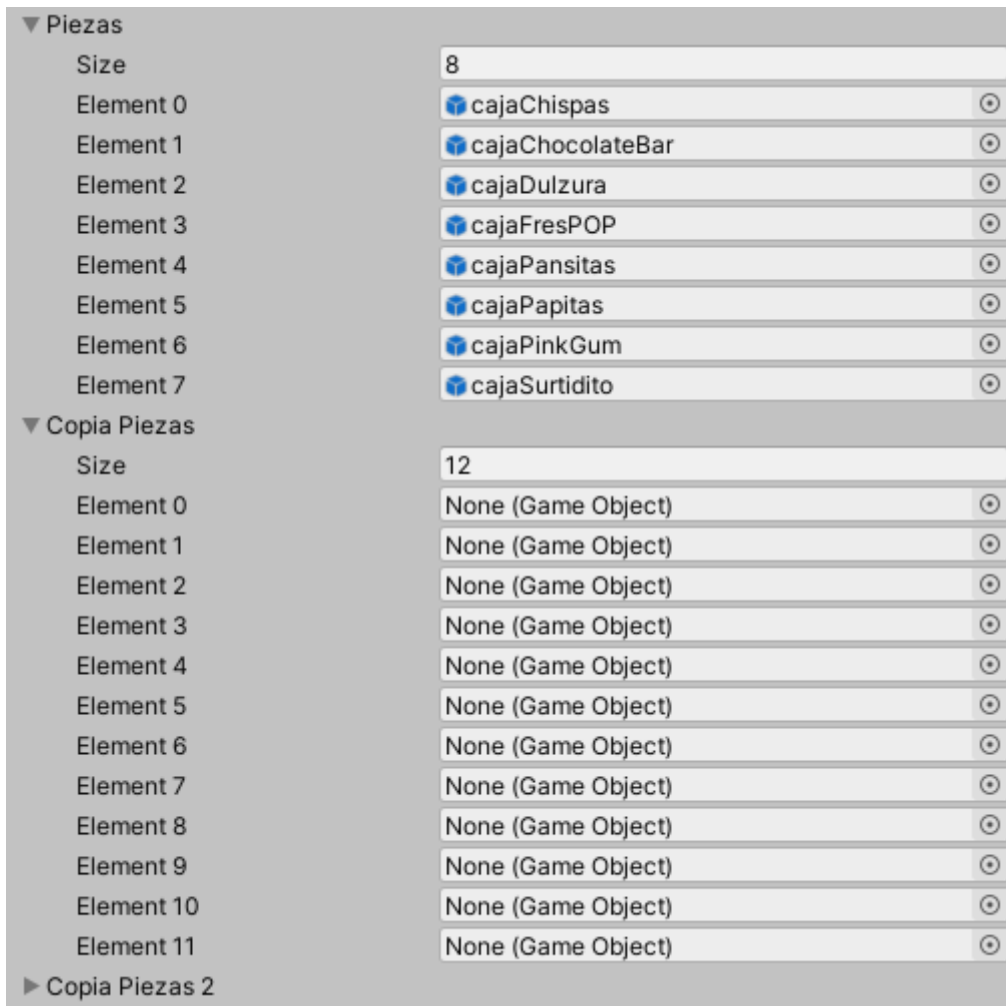


Figura 5.41. Se expanden las pestañas de los arreglos *piezas* y *copiaPiezas* para ver el tamaño de los arreglos y sus elementos (en el caso del arreglo *piezas*).

La figura 5.42 y 5.43 muestran el código del *script* *DetectorPiezas*. Este script se agrega como componente del *Detector 2* (figura 5.27). Al igual que con el *script* *CreaPieza* este *script* también se utiliza para el funcionamiento, tanto de la empacadora como de la estampadora. Este *script* se utiliza para detectar las piezas que han caído en la caja contenedora, las cuales pueden ser los paquetes de la empacadora o las piezas de trabajo de la estampadora (El *script* sólo asegura que una pieza a caído a la caja contenedora, no determina si fue un paquete de la empacadora o una pieza de trabajo de la estampadora). La línea 6 (fig. 5.42) crea la clase *DetectorPieza* cuyo cuerpo está conformado por las líneas 7 a 51.

La línea 9 crea una variable pública de tipo *CreaPieza* llamada *refCreaPieza*, la cual es utilizada para acceder a las variables del *script* *CreaPieza* visto anteriormente.

La línea 12 crea una variable pública de tipo *Animator* llamada *anim*, que se utiliza para acceder al componente *Animator* de las cajas contenedoras de paquetes. Este componente

```

1  using UdonSharp;
2  using UnityEngine;
3  using VRC.SDKBase;
4  using VRC.Udon;
5
6  public class DetectorPieza : UdonSharpBehaviour
7  {
8      // referencia al script CreaCajas
9      public CreaPieza refCreaPieza;
10
11     // referencia al animator de la caja contenedora
12     private Animator anim;
13
14     // variable para indicar cuando se debe activar la animación del
15     // cerrado de la caja contenedora
16     public bool activaAnimacionCajaContenedora;
17
18     // variable que sirve de contador de las cajas que se han empacado
19     public int contadorPiezasEmpacadas = 0;
20
21     // arreglo para guardar copias de las cajas que se han empacado
22     public GameObject[] copiaPiezas = new GameObject[3];
23

```

Figura 5.42. Código del *Script DetectorPiezas* utilizado para detectar las cajas que caen al contenedor de cajas.

Animator contiene la animación que cierra la caja cuando se llena con el número de piezas especificadas (para ambos gemelos digitales: empacadora y estampadora, son 3 piezas). La línea 16 crea una variable pública de tipo booleano llamada *activaAnimacionCaja-Contenedora*, la cual se utiliza para indicar cuando se debe activar la animación del cerrado de la caja contenedora de paquetes.

La línea 19 crea una variable pública de tipo entero nombrada *contadorPiezasEmpacadas* que es inicializada en cero y se utiliza para llevar un conteo de las piezas que se han empacado.

La línea 22 crea un arreglo público de 3 componentes de nombre *copiaPiezas*. Este arreglo es utilizado para guardar los *GameObjects* de las piezas que han caído a la caja contenedora. Las líneas 24 a la 37 (fig. 5.43) corresponden a la función *OnTriggerEnter* que también se ejecuta continuamente a lo largo de la simulación. Dentro del parámetro de la función *OnTriggerEnter* se especifica que al colisionador que entre al detector se le asignará la variable *colisionador* (línea 24):

Collider colisionador

Las líneas 26 a 36 forman el cuerpo de la función *OnTriggerExit*, donde la línea 26, asigna el *GameObject* que entra al detector a un elemento del arreglo *copiaCajas*. Para referirse al *GameObject* que tiene adjunto el colisionador que entró al detector se utiliza la sentencia

copiaPiezas[contadorPiezasEmpacadas] = colisionador.gameObject;

```

24 void OnTriggerEnter(Collider colisionador)
25 {
26     copiaPiezas[contadorPiezasEmpacadas] = colisionador.gameObject;
27     contadorPiezasEmpacadas++;
28     if (contadorPiezasEmpacadas == 3)
29     {
30         activaAnimacionCajaContenedora = true;
31         for (int i = 0; i <= 2; i++)
32         {
33             Destroy(copiaPiezas[i], 1f);
34         }
35         contadorPiezasEmpacadas = 0;
36     }
37 }
38
39 void Update()
40 {
41     if (activaAnimacionCajaContenedora)
42     {
43         activaAnimacionCajaContenedora = false;
44         anim = refCreaPieza.copiaCajaContenedora.GetComponent<Animator>();
45         anim.SetBool("cerrarCaja", true);
46         Destroy(refCreaPieza.copiaCajaContenedora, 2f);
47         refCreaPieza.timer = 0;
48         refCreaPieza.creaCajaContenedora = true;
49     }
50 }
51 }

```

Figura 5.43, Código de *script* *DetectorPiezas* utilizado para detectar las cajas que caen al contenedor de cajas (continuación).

donde la parte a la derecha del signo igual indica que debe referirse al *GameObject* que tiene atado el colisionador que entró al detector y la parte izquierda es la asignación a un elemento del arreglo, según especifique la variable *contadorPiezasEmpacadas*.

La línea 27 incrementa el contador *contadorPiezasEmpacadas* en una unidad para que la siguiente caja que se empaque se guarde en otro elemento del arreglo.

Las líneas 28 a 36 corresponden a una instrucción *if* que se ejecuta si el contador *contadorPiezasEmpacadas* es igual a 3. Si se cumple la condición entonces, la variable *activaAnimacionCajaContenedora* se vuelve verdadera y después se ejecuta el ciclo de repetición *for* de las líneas 31 a 34. Dentro de la instrucción de repetición *for* se destruyen las cajas por medio del método *Destroy* (línea 33):

$$\text{Destroy}(\text{copiaPiezas}[i], 1f);$$

El primer parámetro del método *Destroy* indica el *GameObject* que será destruido. El segundo parámetro indica el tiempo después del cual será destruido el *GameObject*, en este caso un segundo. Finalizando el ciclo *for* se reinicia el contador *contadorCajasEmpacadas* a cero (línea 35).

Las líneas 39 a 50 corresponden a la función *Update*. Dentro de la función hay una única instrucción *if* que se ejecuta si la variable *activaAnimacionCajaContenedora* es verdadera. Dentro de la función *if*, la línea 43 establece la variable *activaAnimacionCajaContenedora* como falsa, para que solo se ejecute una vez la instrucción *if*.

La línea 44:


```
anim = refCreaPiezas.copiaCajaContenedora.GetComponent < Animator > ();
```

Asigna el componente *animator* del *GameObject copiaCajaContenedora* que se ha declarado en el *script CreaPieza* a la variable *anim* declarada en el *script DetectorPieza*. Una vez asignado el componente *Animator* a la variable *anim* se puede enviar la instrucción que activa la animación del cerrado de la caja contenedora de paquetes. La instrucción:

```
anim.SetBool("cerrarCaja", true);
```

de la línea 45, indica que la variable *cerrarCaja*, declarada en la máquina de estados de la animación de la caja, se hará verdadera.

La línea 46 destruye la caja contenedora después de 2 segundos (tiempo suficiente para que se complete la animación del cerrado de la caja).

La línea 47 establece la variable *timer* declarada en el *script creaPieza* en cero. Esto impide que la nueva caja contenedora se cree antes de que se elimine la anterior y se provoque un efecto de apilamiento, ya que la nueva caja contenedora se creará si el *timer* es mayor 2.5 segundos (ver figura 5.36 línea 112).

La línea 48 establece la variable *creaCajaContenedora* declarada en el *script creaPieza* como verdadera, de modo que se puede crear una caja contenedora nueva desde el *script creaPieza*.

Se debe notar que se accede a las variables que se declararon en el *script CreaPieza* con la variable de tipo *CreaPieza* "*refCreaPieza*". Por ejemplo, en la línea 48 se establece, que la variable *creaCajaContenedora* declarada en el *script CreaPieza* es verdadera, a través de la instrucción:

```
refCreaPieza.creaCajaContenedora = true;
```

La figura 5.44 muestra el *script DetectorPieza*, una vez que se ha agregado como componente del *GameObject Detector 2*, donde se pueden observar las variables declaradas en el *script*. En particular fue asignado el *GameObject Detector 1* en la casilla correspondiente para el *script CreaPieza*, ya que este *script* se asignó como componente del *GameObject Detector 1*.

La figura 5.44 muestra la máquina de estados que controla la animación del cerrado de la caja contenedora de paquetes. La animación solo cierra la caja ya que después de cerrarla la caja desaparece y se crea una nueva caja abierta, no es necesario volver a abrirla. Notamos que la máquina de estados tiene 3 clips de animación. El primer clip llamado *animaciónCajaAbierta* se ejecuta cuando la caja es creada en escena. Esta animación en realidad no cambia con el tiempo y solo representa la caja contenedora abierta. Cuando la variable *cerrarCaja* se hace verdadera (línea 45, figura 5.43) se ejecuta el clip de animación *animacionCerrarCaja* (que es la animación del cerrado de la caja). Inmediatamente después de acabarse el clip del cerrado de la caja se ejecuta el clip de animación *animación-CajaCerrada* (no se requiere alguna condición adicional) que no tiene cambios en el tiempo y solo representa la caja cerrada. En las pruebas de funcionamiento se mostrarán imágenes del cerrado de la caja contenedora.

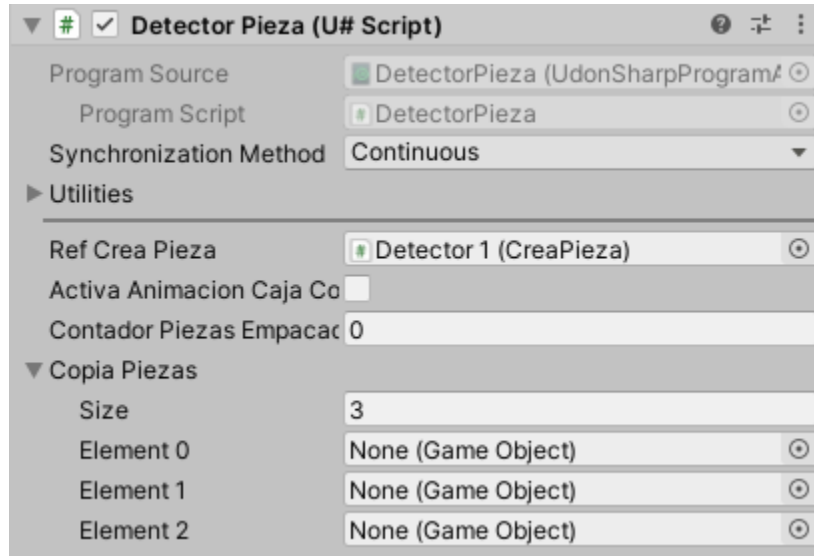


Figura 5.44. Script *DetectorPieza* agregado como componente del *GameObject Detector 2*.

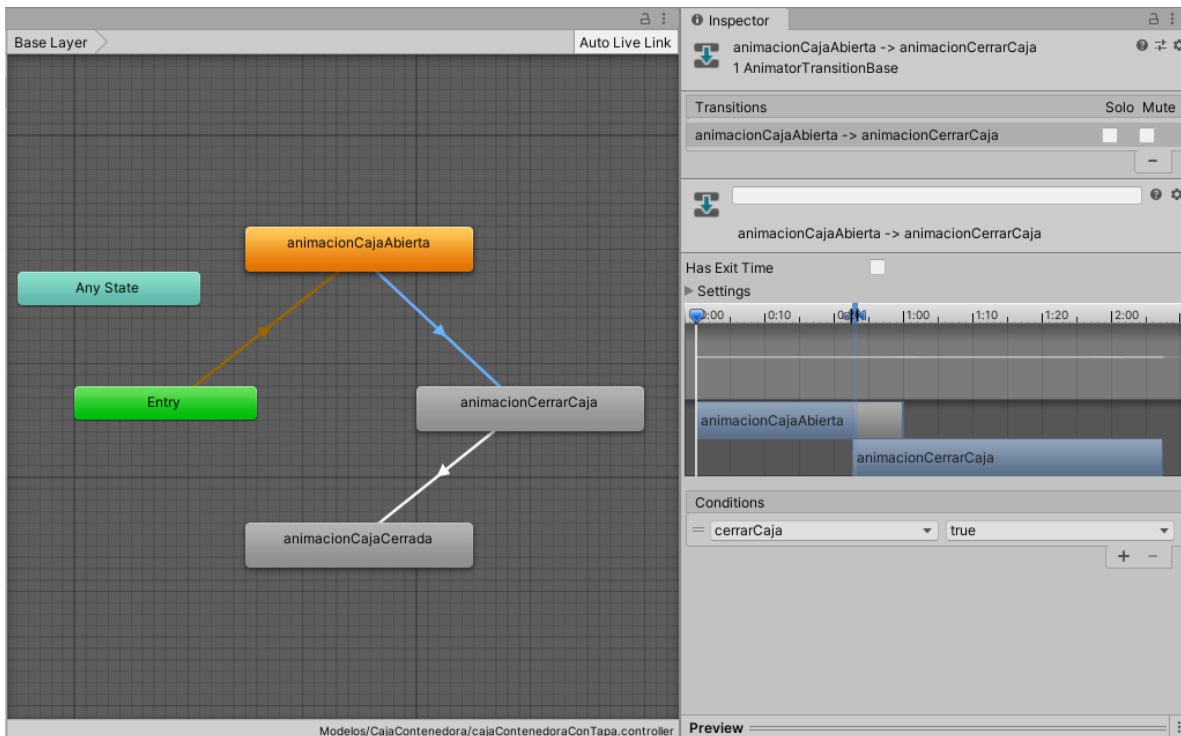


Figura 5.45. Máquina de estados para la animación del cerrado de la caja contenedora de paquetes.

La figura 5.46 muestra el código del *script DestructorPiezas*. Este *script* se encarga de borrar todas las piezas en escena y de reiniciar las variables del *script CreaPiezas*, por lo que también se utiliza para ambos gemelos digitales de la empacadora y de la estampadora. La línea 6 crea la clase *DestructorPiezas* cuyo cuerpo está formado por las líneas 7 a 29. La línea 9 crea una variable de tipo *CreaPieza* llamada *refCreaPieza* para acceder a las variables del *script CreaPieza*. Las líneas 11 a 28 corresponden a la función *Update* la cual solo tienen una sentencia *if* anidada. En esta sentencia *if* se establece que si ya se crearon las cajas iniciales:

refCreaPieza.crearonPiezasInicio, es verdadera

y si se desactivaron el botón enclavado y la válvula de la unidad de mantenimiento:

!*refCreaCajas.animBoton.GetBool("Boton1")*, es verdadera y
!*refCreaCajas.animFRL.GetBool("UMBool")*, es verdadera

entonces se ejecutan las instrucciones dentro de la sentencia *if* (líneas 15 a 27). Se debe notar que las dos sentencias anteriores son negaciones.

Las líneas 16 a 20 pertenecen al ciclo *for* (anidado dentro de la sentencia *if*) que se ejecuta desde cero hasta el tamaño del arreglo *copiaPiezas* en incrementos de 1.

Las líneas 18 y 19 (dentro de cuerpo de la instrucción *for*):

Destroy(refCreaPieza.copiaPiezas[i]);
Destroy(refCreaPieza.copiaPiezas2[i]);

borran cada una de las piezas que pudieran estar contenidas en los dos arreglos que almacenan las piezas declarados en el *script CreaPieza*. En este caso el método *Destroy* solo define un parámetro dentro de los paréntesis que es el *GameObject* que se quiere destruir, no se especifica tiempo de destrucción por lo que la destrucción ocurre instantáneamente. Las líneas 21 a 26 (dentro de la instrucción *if* de la línea 13) reinician todas las variables del *script CreaPieza* a sus valores iniciales. Como puede observarse todas las variables llevan la estructura

refCreaPieza.variable

Esto indica que las variables a las cuales se aplican las sentencias corresponden a variables en el *script CreaPieza*. Se recalca que antes de iniciar la ejecución de la simulación se deben colocar todos los *GameObjects* que contienen los *scripts* a los cuales se hizo referencia en los recuadros correspondientes (en este caso solo se hizo referencia a las variables del *script CreaPieza*). En la figura 5.47 se muestra el *GameObject Detector 1* colocado en el recuadro que se crea para la variable *refCreaPieza* que hace referencia al *script CreaPieza*, ya que dicho *script* es componente del *GameObject Detector1*.

```

1  using UdonSharp;
2  using UnityEngine;
3  using VRC.SDKBase;
4  using VRC.Udon;
5
6  public class DestructorPiezas : UdonSharpBehaviour
7  {
8      // referencia al script CreaCajas
9      public CreaPieza refCreaPieza;
10
11     void Update()
12     {
13         if (refCreaPieza.crearonPiezasInicio && !refCreaPieza.animFRL.GetBool("UMBool") &&
14             !refCreaPieza.animBoton.GetBool("Boton1"))
15         {
16             for (int i = 0; i < refCreaPieza.copiaPiezas.Length; i++)
17             {
18                 Destroy(refCreaPieza.copiaPiezas[i]);
19                 Destroy(refCreaPieza.copiaPiezas2[i]);
20             }
21             Destroy(refCreaPieza.copiaCajaContenedora);
22             refCreaPieza.crearonPiezasInicio = false;
23             refCreaPieza.contadorArreglo = 0;
24             refCreaPieza.contadorPaquetes = 0;
25             refCreaPieza.activaArreglo2 = false;
26             refCreaPieza.activaArreglo1 = true;
27         }
28     }
29 }

```

Figura 5.46. Código del *script DestructorPiezas* utilizado para eliminar todas las cajas en escena y reiniciar las variables.

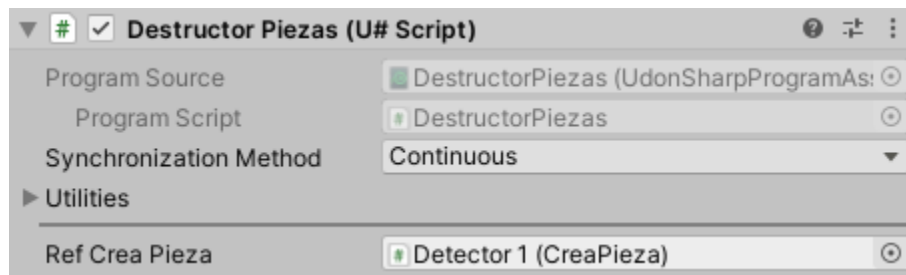


Figura 5.47. Se coloca el *GameObject* que contiene el *script CreaPieza (Detector 1)* en el recuadro correspondiente a la variable *refCreaCajas* de tipo *CreaPieza*.

5.2. Programación de la Indentadora

La programación de la Indentadora sigue un procedimiento similar al desarrollado para la empacadora. Primero se colocaron los colisionadores a los componentes del gemelo digital. En la figura 5.48 se muestra el colisionador agregado al *GameObject* correspondiente a la base horizontal de la indentadora.

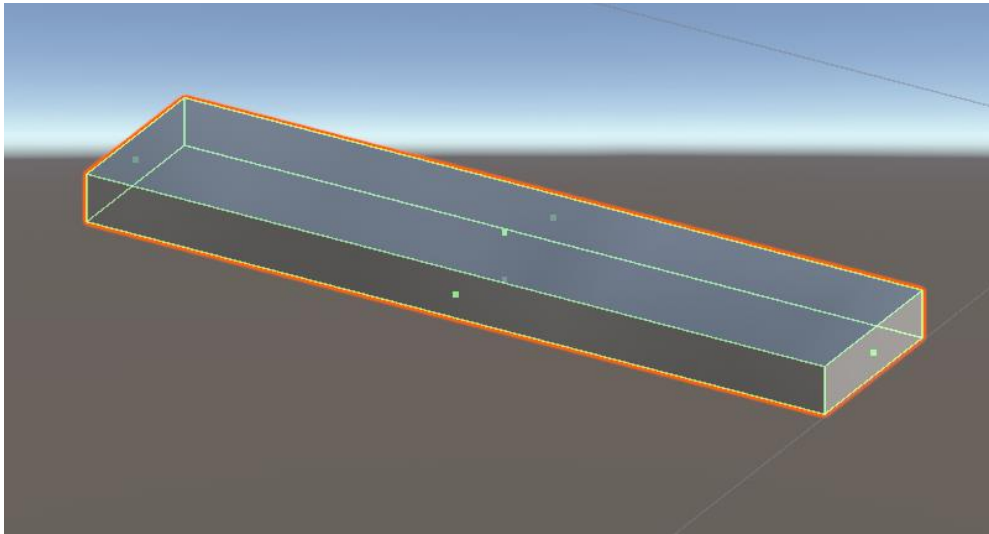


Figura 5.48 Colisionador atado a la base horizontal de la indentadora.

La figura 5.49 muestra los colisionadores que se ataron a las guías de la indentadora. El colisionador atado a la guía extrema se hizo más alto para que la pieza de trabajo no se inclinara al chocar con la guía.

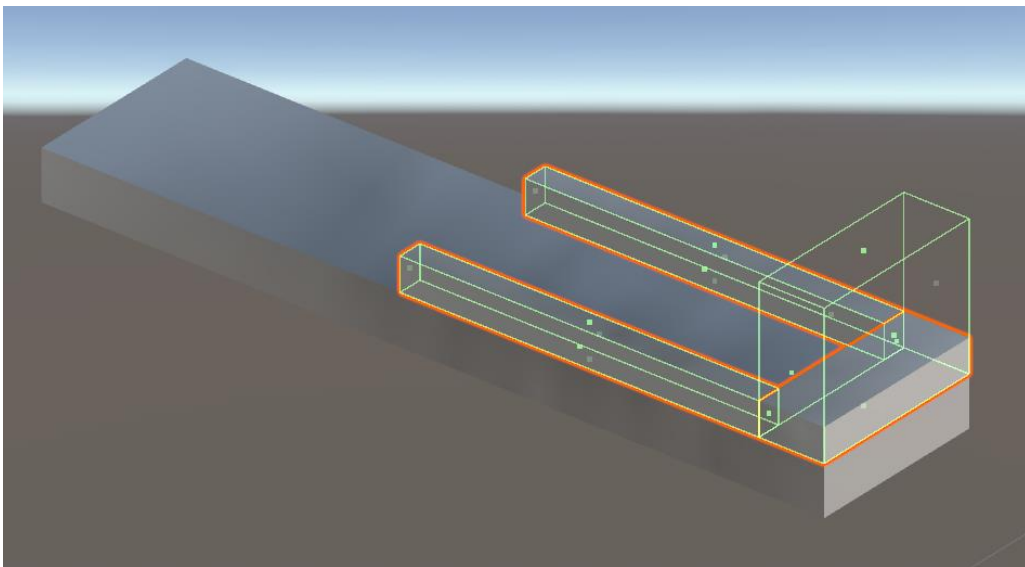


Figura 5.49. Colisionadores agregados a las guías de la indentadora.

La figura 5.50 muestra los colisionadores agregados al cilindro A de la indentadora y a su vástago. Los colisionadores extremos del cilindro y el central que pertenece al vástago cumplen el mismo propósito descrito para los cilindros de la empacadora que es básicamente el control de la carrera de los cilindros. Al vástago del cilindro A, se le agregó también un componente *Rigidbody* para controlar su movimiento mediante la aplicación de una fuerza. Dicho componente se muestra en la figura 5.51. La masa utilizada en este caso para el vástago fue de 0.1 [Kg]. Se desactivó la casilla *Use Gravity* ya que no se requiere que el vástago interactúe con la gravedad y además se restringieron los movimientos lineales en las coordenadas X y Y, además de todos los movimiento angulares.

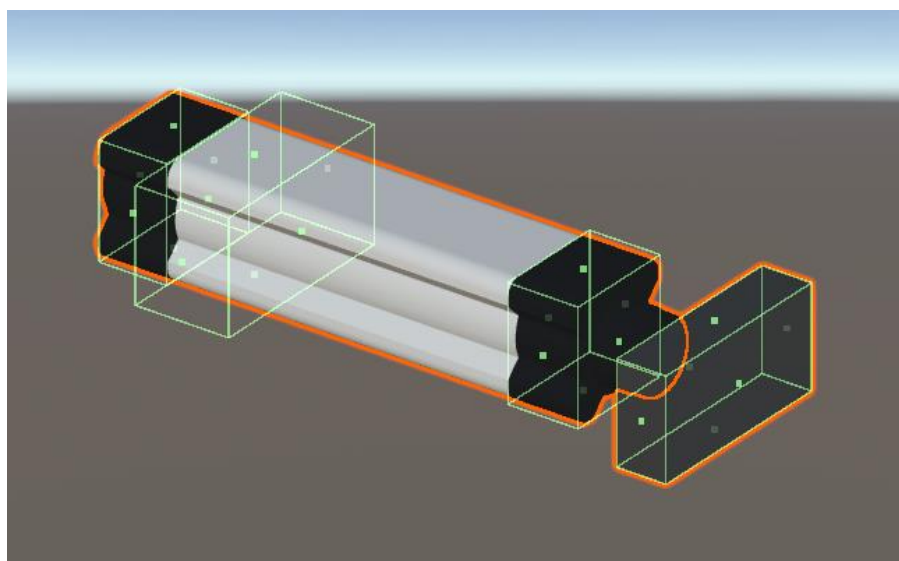


Figura 5.50. Colisionadores atados al cilindro A de la indentadora y su vástago.

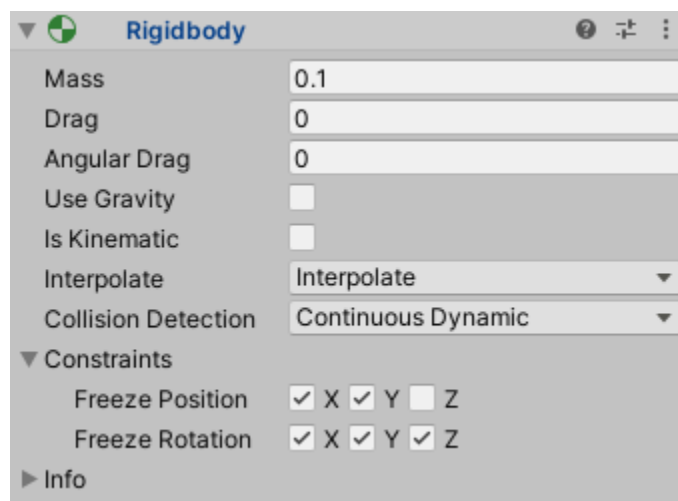


Figura 5.51. Componente *Rigidbody* agregado al vástago del cilindro A de la indentadora.

La figura 5.52 muestra los colisionadores atados al cilindro B de la indentadora y a su vástago. Los colisionadores que se agregaron tienen la misma función que los ya descritos anteriormente. En este caso se puede observar que uno de los colisionadores no cubre al cabezal, de hecho, está situado a un costado. Esto se hizo de este modo para que el cabezal penetrara la pieza de trabajo dando la impresión de que lo está perforando. Si se hubiera optado por cubrir el cabezal con un colisionador, entonces chocarían los colisionadores del cabezal y de la pieza de trabajo y no parecería que está ocurriendo la indentación de la pieza. El colisionador colocado a un costado sirve para que la pieza de trabajo choque con el colisionador y de la impresión de que el cabezal es una pieza sólida. Esto ocurriría si el usuario no realiza la secuencia de movimiento correcta y primero baja el vástago B y después activa el vástago A empujando la pieza directamente al cabezal del vástago B.

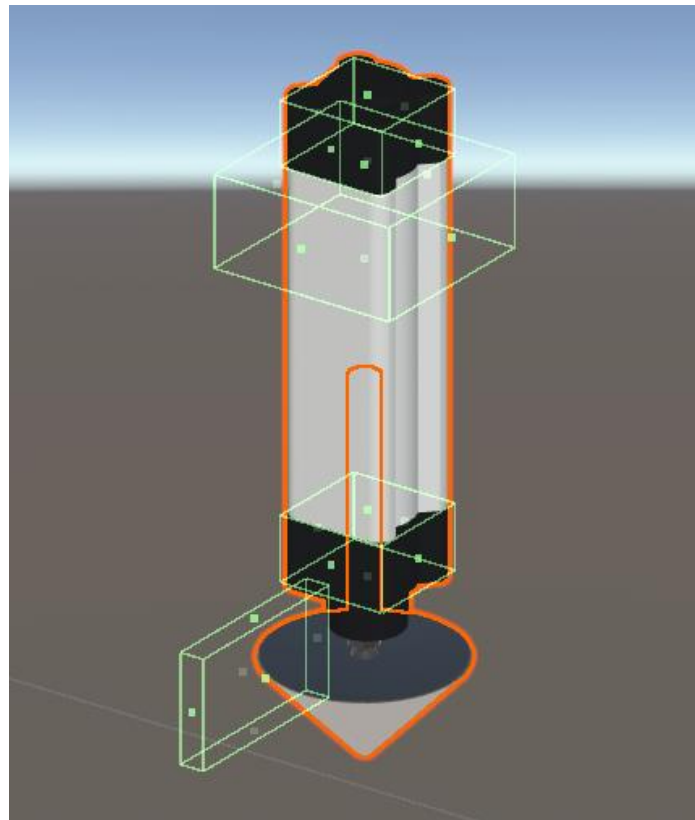


Figura 5.52. Colisionadores agregados al cilindro B de la indentadora y su vástago.

La figura 5.53 muestra el componente *Rigidbody* agregado al vástago B para poder moverlo mediante la aplicación de una fuerza. La masa utilizada es de $0.1 [Kg]$. También se restringieron los movimientos lineales y angulares en todas las direcciones a excepción del movimiento lineal en dirección Y . La casilla *Use Gravity* también fue desactiva ya que no se requiere que el vástago este sometido a la aceleración de la gravedad.

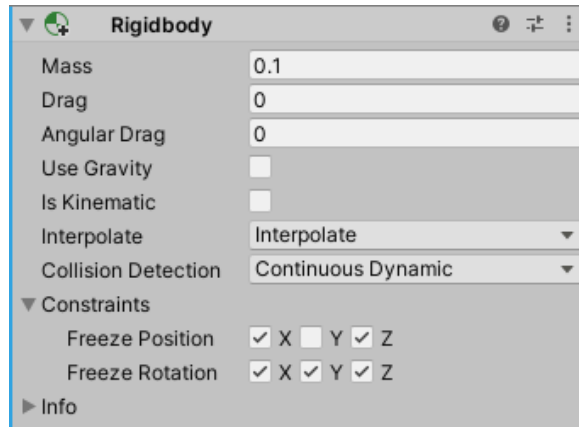


Figura 5.53. Componente *Rigidbody* agregado al vástago B de la indentadora.

Para la aplicación de la fuerza a los vástagos se utilizó el mismo *script* “*Vastago*” utilizado en los vástagos de la empacadora y solo se especificaron los vectores de fuerza y los *GameObjects* que contienen las animaciones de movimiento de los vástagos de la mesa de trabajo asociada al gemelo de la indentadora. La figura 5.54 muestra el *script* que se agregó al vástago A de la indentadora. Se observa que se ha establecido una fuerza del -1 [N] aplicada en la dirección Z. En la variable *goVastago* se agrega el *GameObject* del vástago del cilindro marcado como A en la mesa de trabajo (como es una copia todos los vástagos tienen el mismo nombre: *Vástago_D*, pero se refieren a vástagos distintos). La figura 5.55 muestra el *script* *Vastago* agregado al vástago B de la indentadora. En este caso la dirección de la fuerza se establece en dirección Y y tiene la misma magnitud que la aplicada al vástago A. El vástago de control es el marcado como B en la mesa de trabajo.

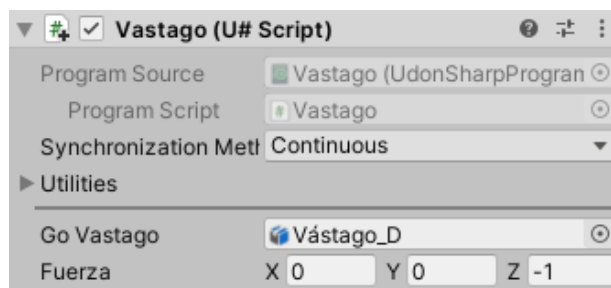


Figura 5.54. *Script* *Vastago* agregado al vástago A de la indentadora.

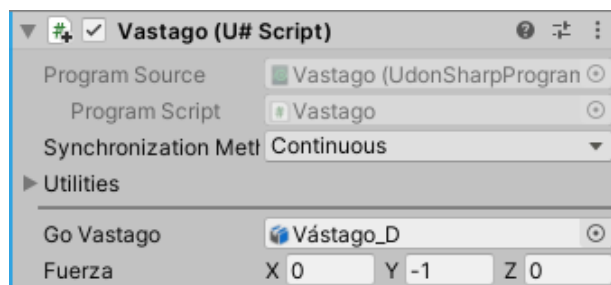


Figura 5.55. *Script* *Vastago* Agregado al vástago B de la indentadora.

Las figuras 5.56 y 5.57 muestran el código del *script CreaCubo* utilizado para crear el cubo que se indenta. Se crea un cubo cuando se acciona (expande) el cilindro A de la indentadora. Se destruye el cubo cuando ambos cilindros regresan a su posición retraída independientemente de si se hizo o no la indentación.

La línea 6 del código mostrado en la figura 5.56 crea la clase *CreaCubo*.

La línea 9 crea una variable de tipo *DetectorCubo* llamada *refDetectorCubo* para acceder a las variables del *script DetectorCubo*. El código de este *script* se verá más adelante.

Las líneas 12 y 13 declaran dos variables públicas de tipo *GameObject* llamadas *vastagoA* y *vastagoB* donde se guardan los *GameObjects* que contienen las animaciones de los cilindros neumáticos de la mesa de trabajo (los vástagos de los cilindros).

Las líneas 16 y 17 declaran dos variables privadas de tipo *Animator* llamadas *animVastagoA* y *animVastagoB* que se utilizan para acceder a los componentes *Animator* de los *GameObjects* guardados en las variables *vastagoA* y *VastagoB*.

La línea 20 declara una variable pública de tipo *GameObject* llamada *cubo* en la cual se guarda el *GameObject* del cubo que será creado en la simulación.

La línea 23 crea una variable pública de tipo *GameObject* llamada *copiaCubo* utilizada para guardar las copias de los *GameObjects* de los cubos que serán creados en escena y posteriormente destruidos.

La línea 26 crea una variable pública de tipo booleano llamada *creaCubo* que se utiliza para indicar el momento en el que se debe crear un cubo en escena.

La línea 29 crea una variable pública de tipo *Vector3* llamada *posCubo* en la cual se guarda la posición de creación del cubo.

```
1  using UdonSharp;
2  using UnityEngine;
3  using VRC.SDKBase;
4  using VRC.Udon;
5
6  public class CreaCubo : UdonSharpBehaviour
7  {
8      // referencia al Script Detector Cubo
9      public DetectorCubo refDetectorCubo;
10
11     // variables para guardar los GameObjects de los vástagos de la mesa de trabajo
12     public GameObject vastagoA;
13     public GameObject vastagoB;
14
15     // variables para acceder a los componentes Animator
16     private Animator animVastagoA;
17     private Animator animVastagoB;
18
19     // variable donde se guarda el prefab del cubo
20     public GameObject cubo;
21
22     // variable donde almacenamos una copia del prefab cubo para poder destruirla
23     public GameObject copiaCubo;
24
25     // variable que indica si debe crearse un cubo
26     public bool creaCubo = false;
27
28     // posicion creación del cubo
29     private Vector3 posCubo;
30
```

Figura 5.56. Código del *script CreaCubo* utilizado para crear la pieza cúbica que se indenta.

Las líneas 31 a 35 (fig. 5.57) corresponden a la función *Start* que se ejecuta al inicio de la simulación. Dentro de esta función, las líneas 33 y 34 asignan los componentes *Animator* contenidos en los *GameObjects* *vastagoA* y *vastagoB* a las variables *animVastagoA* y *animVastagoB* respectivamente (el método *GetComponent* se ha visto anteriormente).

Las líneas 37 a 53 forman parte de la función *Update*. Dentro de esta función, las líneas 39 a 43 corresponden a una instrucción *if* que se ejecuta si la variable "*VastDobAnimBool*", en el componente *Animator* del vástago A de la mesa de trabajo, es verdadera (si el cilindro se expande) y si no se ha creado ya un cubo (*creaCubo* es falsa). Si la instrucción *if* se ejecuta, entonces la sentencia de la línea 41 crea en escena una copia del cubo y la asigna a la variable *copiaCubo*. La estructura del método *Instantiate* se ha señalado anteriormente. Una vez que se ha creado el cubo la línea 42 establece que la variable *creaCubo* se hace falsa, por lo que la instrucción *if* se ejecutara de nuevo solo hasta que se vuelva a indicar que se debe crear un cubo (la variable *creaCubo* se haga de nuevo verdadera).

Dentro de la función *Update* se anida otra instrucción *if* (líneas 45 a 51) encargada de la destrucción del cubo. La instrucción se ejecutará si se contraen el cilindro A y el cilindro B de la mesa de trabajo y si además ya se ha creado un cubo. La línea 48 destruye la copia del cubo que se ha creado en escena, después de 0.5 segundos. La línea 49 establece que la variable *creaCubo* se vuelve falsa por lo que ahora se puede volver a crear otro cubo. La línea 50 establece como falsa la variable *decalCreado* declarada en el *script* *DetectorCubo* (se verá a continuación).

La figura 5.58 muestra el *script* *creaCubo* una vez que se ha agregado a un *GameObject* en escena. En este caso no se requiere que el *script* este atado a un *GameObject* en particular; aquí se eligió colocarlo como componente del *GameObject* de la mesa del Gemelo digital de la indentadora. Se observa que se han agregado los *GameObjects* requeridos además de un vector que especifica la posición de creación del cubo.

```
31 void Start()
32 {
33     animVastagoA = vastagoA.GetComponent<Animator>();
34     animVastagoB = vastagoB.GetComponent<Animator>();
35 }
36
37 void Update()
38 {
39     if (animVastagoA.GetBool("VastDobAnimBool") && !creaCubo)
40     {
41         copiaCubo = Instantiate(cubo, posCubo, Quaternion.Euler(-90f, 0f, 0f));
42         creaCubo = true;
43     }
44
45     if (!animVastagoA.GetBool("VastDobAnimBool") && creaCubo
46         && !animVastagoB.GetBool("VastDobAnimBool"))
47     {
48         Destroy(copiaCubo, 0.5f);
49         creaCubo = false;
50         refDetectorCubo.decalCreado = false;
51     }
52 }
53 }
```

Figura 5.57. Código del *script* *CreaCubo* utilizado para crear la pieza cúbica que se indenta (continuación).

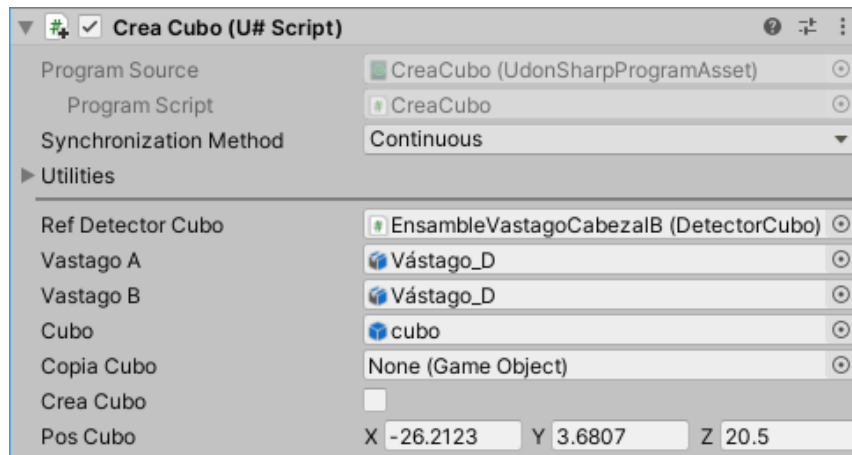


Figura 5.58. *Script CreaCubo* agregado como componente de un *GameObject* del gemelo digital de la indentadora.

5.2.1. Simulación de la indentación con Decals

El *script DetectorCubo* al que se hizo referencia anteriormente se utiliza para crear una estampa (*Decal*) en la parte superior del cubo, después de que el vástago B ha “indentado” la pieza. Esta estampa se utiliza para simular que el cabezal del vástago B ha dejado una marca después de que el cabezal atravesó la pieza (recordemos que el cabezal no tiene colisionador adherido por lo que puede atravesar el cubo). La estampa utilizada se muestra en la figura 5.59. Este tipo de estampas se utilizan en videojuegos para simular que las balas atraviesan las superficies. En este caso la estampa se utilizó para recrear la marca de la indentación.



Figura 5.59. Estampa del agujero de una bala. Obtenido de [42].

Para colocar la estampa sobre algún objeto primero se debe crear un *GameObject* que contenga la estampa. El uso de planos es ideal para este objetivo. Primero se crea un plano de algún tamaño predefinido y se coloca en escena. La estampa, que debe importarse a Unity, es entonces colocada sobre el plano. Se puede ajustar el tamaño de la estampa para que encaje a la perfección dentro del plano. Para simular la indentación se utilizó un plano del mismo tamaño que las caras del cubo. La figura 5.60 muestra el plano una vez que se le ha agregado la estampa del agujero de bala. Este plano ya puede colocarse sobre las superficies; sin embargo, no quedará del todo bien ya que las partes externas del agujero son de color oscuro y se prefiere que las partes externas sean transparentes y así también se observen las texturas de los objetos donde se pone la estampa. Para solucionarlo se debe cambiar el sombreador (*Shader*) de la imagen. Al agregar la imagen al plano, la ventana inspector del *GameObject* correspondiente al plano muestra un nuevo componente nombrado como la imagen que se agregó al plano. En este caso la imagen se llamó “agujero” y el componente, como se ve desde la ventana inspector, se observa en la figura 5.61. De manera predeterminada el sombreador se establece como *Default* (por defecto) y se obtiene el resultado mostrado en la figura 5.60. Para lograr el efecto deseado se requiere el sombreador *Sprites/Default*. Este sombreador permite eliminar las partes externas de la estampa. En este caso como es una implementación para VRChat, se utilizaron los sombreadores proporcionados por su SDK, los cuales se agregan a las opciones que se tienen de manera predeterminada en Unity. El sombreador elegido es, por tanto: *VRChat/Sprites/Default*, como se observa en la figura 5.61. La figura 5.62 muestra el resultado de elegir el sombreador *Sprites* para la imagen del agujero colocada sobre el plano, donde es evidente que el plano que contiene al agujero deja de ser perceptible. Este último plano con el agujero se establece como un *GameObject* predeterminado (también llamado prefab), el cual ya puede ser creado en escena para simular agujeros.

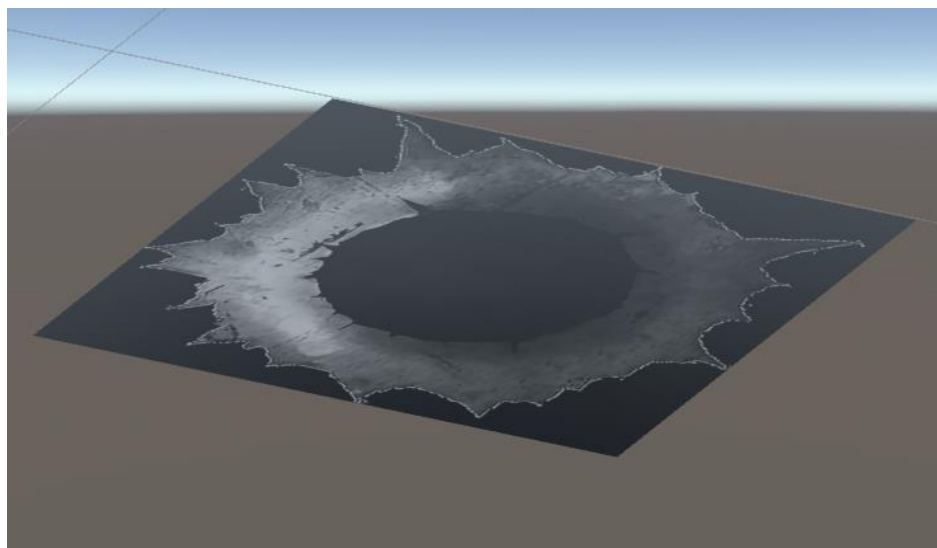


Figura 5.60. Estampa de un agujero colocado sobre un plano con el sombreador establecido en *Default*.

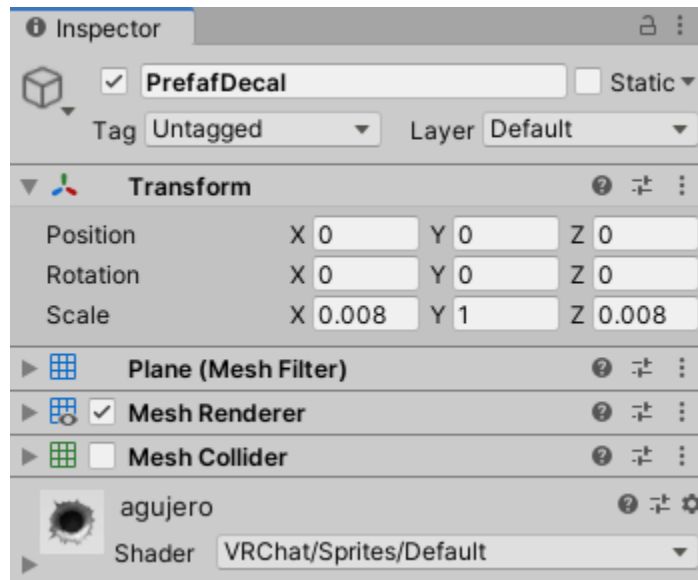


Figura 5.61. Ventana inspector del plano sobre el cual se coloca la estampa del agujero. El plano se nombra por defecto PrefafDecal. El sombreador (*Shader*) se ha cambiado de *Default* a *VRChat/Sprites/Default*.

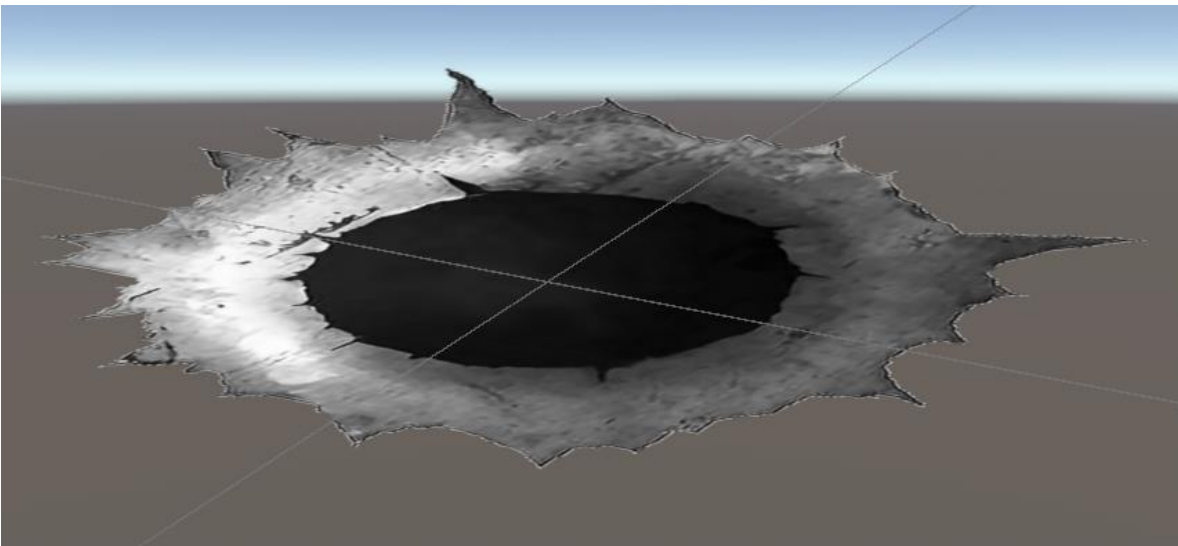


Figura 5.62. Estampa de un agujero colocado sobre un plano con el sombreador establecido en *VRChat/Sprites/Default*.

La figura 5.63 y 5.64 muestran el código utilizado para la creación de las estampas sobre la pieza de trabajo de la indentadora, y así poder simular la indentación. En la línea 6 se declara la clase *DetectorCubo* que está formada por las líneas 7 a 48.

La línea 9 crea una variable pública de tipo *CreaCubo* llamado *refCreaCubo* utilizada para acceder a las variables declaradas en el *script CreaCubo* visto anteriormente.

La línea 12 declara una variable privada de tipo *Ray* llamada *rayo* que se utiliza para especificar la dirección y el punto de origen del *Raycast*. Como se recordará el *Raycast* es un rayo invisible que se utiliza para propósitos de detección de colisionadores.

La línea 13 declara una variable privada de tipo *RaycastHit* llamada *hit*. Esta variable sirve para obtener información del *GameObject* cuyo colisionador se ha detectado mediante el *Raycast*.

La línea 16 crea una variable pública de tipo *GameObject* llamada *prefabDecal* que se utiliza para guardar el *GameObject* del plano con la estampa que se describió anteriormente.

La línea 17 crea una variable privada de tipo *GameObject* utilizada para guardar una copia del *GameObject* del plano con la estampa para que pueda ser creado y posteriormente destruido de la escena.

La línea 20 crea una variable pública de tipo booleano llamada *decalCreado* utilizada para indicar si ya ha sido creado el *GameObject* del plano con la estampa sobre la pieza de trabajo.

La línea 23 crea una variable privada de tipo *Vector3* llamada *posDecal* utilizada para guardar la posición donde debe crearse el *GameObject* del plano con la estampa.

La variable *rotDecal* declarada como privada en la línea 24 es utilizada para determinar la rotación adecuada del *GameObject* del plano con la estampa.

```
1 using UdonSharp;
2 using UnityEngine;
3 using VRC.SDKBase;
4 using VRC.Udon;
5
6 public class DetectorCubo : UdonSharpBehaviour
7 {
8     // Referencia al script CreaCubo
9     public CreaCubo refCreaCubo;
10
11     // variables para crear un rayo que detecte el cubo
12     private Ray rayo;
13     private RaycastHit hit;
14
15     // variables para guardar el decal
16     public GameObject prefabDecal;
17     private GameObject copiaPrefabDecal;
18
19     // variables que indican cuando crear el decal
20     public bool decalCreado = false;
21
22     // variables para indicar la posición y orientación del decal
23     private Vector3 posDecal;
24     private Quaternion rotDecal;
25
```

Figura 5.63. Script *DetectorCubo* utilizado para crear la estampa del agujero sobre la pieza de trabajo de la indentadora.

La línea 26 declara la función *Start* dentro de la cual se declara; en la línea 28, la sentencia:

```
rayo.direction = Vector3.down;
```

que asigna la dirección establecida en el vector: *Vector3.down*, que es equivalente al vector $(0, -1, 0)$, como dirección del *Raycast* guardado en la variable *rayo* a través del método *direction*. La dirección del rayo no cambia a lo largo de la ejecución del programa.

Las líneas 31 a 47 declara la función *Update*, dentro de la cual, la línea 34:

```
rayo.origin = gameObject.transform.position + new Vector3(0f, -0.04f, 0f);
```

asigna el punto de origen del *Raycast* a la variable *rayo* a través del método *origin*. Esta asignación se define como la suma vectorial de la posición del *GameObject* al cual está atado el *script DetectorCubo* más un vector de 3 dimensiones. En este caso el *script* se ata al vástago B de la indentadora y por lo tanto el punto de origen está contenido en el vástago. Los *GameObjects* tienen puntos de referencia que sirven para posicionarlos en el mundo. Estos puntos de referencia se pueden definir desde Blender cuando se editan las texturas. Para el caso particular de los ensamblados de los vástagos y los cabezales los puntos de referencia se definieron en los centroides de los modelos 3D, definidos por el software Blender.

La instrucción *gameObject.transform.position* obtiene la posición del punto de referencia del *GameObject* al cual está atado el *script*, en este caso el centroide del vástago B de la indentadora. Después a esta posición (que es un vector de 3 dimensiones), se le suma el vector: *new Vector3(0f, -0.04f, 0f)*, para mover el punto de origen a la punta del indentador. Esta instrucción se coloca dentro de la función *Update* para que se actualice continuamente con el movimiento del vástago.

```
26 void Start()
27 {
28     rayo.direction = Vector3.down;
29 }
30
31 void Update()
32 {
33     // origen del rayo
34     rayo.origin = gameObject.transform.position + new Vector3(0f, -0.04f, 0f);
35
36     // dibuja el rayo
37     Debug.DrawRay(rayo.origin, rayo.direction * 0.025f, Color.red);
38
39     if (Physics.Raycast(rayo, out hit, 0.025f) && !decalCreado && refCreaCubo.creaCubo)
40     {
41         posDecal = hit.point + hit.normal * 0.001f;
42         rotDecal = Quaternion.FromToRotation(Vector3.up, hit.normal);
43         copiaPrefabDecal = Instantiate(prefabDecal, posDecal, rotDecal);
44         copiaPrefabDecal.transform.parent = refCreaCubo.copiaCubo.transform;
45         decalCreado = true;
46     }
47 }
48 }
```

Figura 5.64. *Script DetectorCubo* utilizado para crear la estampa del agujero sobre la pieza de trabajo de la indentadora (continuación).

La línea 37 dibuja el rayo en la ventana escena, pero solo es útil para fines de desarrollo ya que no es visible durante la ejecución de la simulación.

La línea 39 declara una instrucción *if* que se ejecuta si el rayo detecta un colisionador, si la variable *decalCreado* es falsa (no se ha creado una estampa) y si la variable *refCreaCubo.creaCubo* es verdadera (ya se creó el cubo). Dentro de esta instrucción *if*, la condición:

$$\text{Physics.Raycast}(\text{rayo}, \text{out hit}, 0.025f)$$

es la encargada de detectar los colisionadores por medio del *Raycast*. El primer parámetro: *rayo*, especifica la dirección y el punto de origen del rayo los cuales fueron especificados en las líneas 28 y 34 respectivamente. El parámetro *out hit*; se encarga de regresar información del colisionador con el que choca el rayo, como: el nombre del *GameObject* que tiene el colisionador que se detectó, la distancia a la que se encuentra el colisionador o la dirección de la cara del colisionador que se toca con el rayo (la normal al plano de la cara del colisionador). El tercer parámetro es la distancia del rayo medida desde el origen del rayo. Si todas las condiciones del *if* de la línea 39 se cumplen entonces se ejecutan las líneas 41 a 45. La línea 41 establece el punto donde se creará el prefab del decal:

$$\text{posDecal} = \text{hit.point} + \text{hit.normal} * 0.001f;$$

La instrucción *hit.point* indica el punto donde el rayo interseca el colisionador. La instrucción *hit.normal* indica la dirección de la normal del plano que forma la cara del colisionador con el que interseca el rayo. La sentencia completa indica que la posición del *Decal* será 0.001 unidades arriba del punto de intersección del rayo y el plano del colisionador. Si se colocará el *Decal* justo en la posición de intersección del plano y el rayo, el *Decal* y el *GameObject* se encimarían lo que provoca efectos visuales extraños como que no se logre apreciar por completo el *Decal*.

La línea 42 establece la rotación adecuada del *Decal* a través de la asignación:

$$\text{rotDecal} = \text{Quaternion.FromToRotation}(\text{Vector3.up}, \text{hit.normal});$$

donde los parámetros del método *FromToRotation* establecen el ángulo entre las direcciones del primer y segundo parámetros. En este caso se obtiene el ángulo entre la dirección Y positiva (hacia arriba) y la normal de la cara del colisionador con el que chocó el rayo (la instrucción *Vector3.up* es equivalente al vector (0, 1, 0)).

La línea 43 asigna una copia del *Decal* colocado sobre el cubo a la variable *copiaDecalPrefab*. El primer parámetro del método *Instantiate* (línea 43) indica que el *GameObject* guardado en la variable *prefabDecal* será creado en escena; el segundo establece la posición, que en este caso sería la guardada en la variable *posDecal*; y el tercero establece la rotación, que en este caso es la guardada en la variable *rotDecal*.

La línea 44:

```
copiaDecalPrefab.transform.parent = refCreaCubo.copiaCubo.tranform;
```

Se utiliza para asignar el *Decal* creado como hijo de la copia del cubo que se creó en escena y que se está indentando. Esto para que al momento de destruir el cubo se destruya de igual manera el *Decal* que se creó por encima del cubo.

Por último, la línea 45 establece la variable *decalCreado* como verdadero lo que indica que ya se ha creado el *Decal*. El resultado de la impresión del *Decal* sobre el cubo se verá en la sección posterior de pruebas de la indentadora.

5.3. Programación de la estampadora

La programación de la estampadora comienza, al igual que con los otros gemelos digitales, con la adición de colisionadores y cuerpos rígidos. La figura 5.65 muestra el colisionador que se agregó a la base de la estampadora. Al igual que con la base de la empacadora, el colisionador cubre partes vacías, pero no supone problema porque las piezas de trabajo no tocan esas partes del colisionador y si las llegasen a tocar la mesa por debajo estaría cumpliendo la misma función de soporte de las piezas de trabajo.

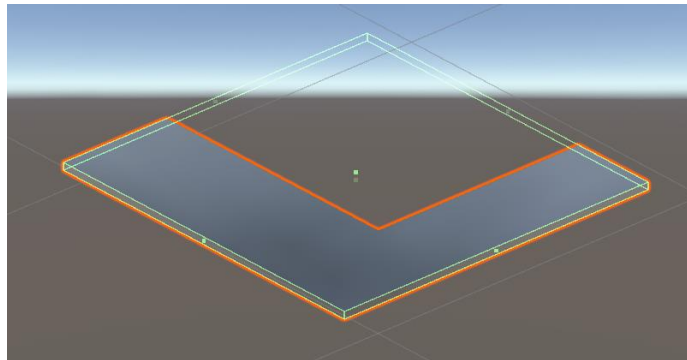


Figura 5.65. Colisionador *Box Collider* agregado a la base de la estampadora.

La figura 5.66 muestra la rampa de la estampadora a la cual se le agregó un colisionador *Box Collider*. Fue requerido hacer el colisionador un poco más delgado que el *GameObject* para evitar que la pieza de trabajo chocara con el colisionador y en su lugar callera sobre él. Tanto para la base como para la rampa se utilizaron los mismos *Physics Materials* utilizados para la base y la rampa de la empacadora (fig. 5.3).

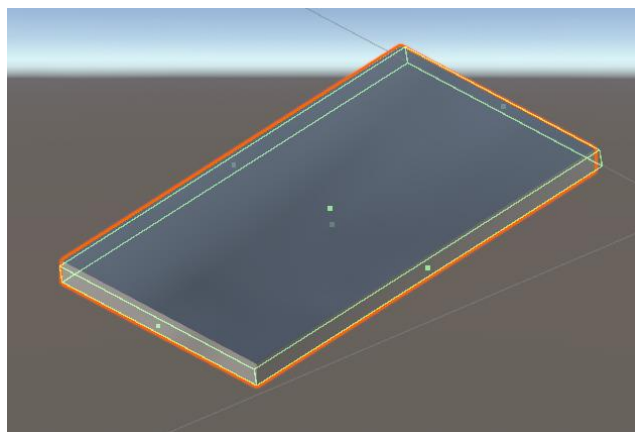


Figura 5.66. Rampa de la estampadora a la que se agregó un colisionador *BoxCollider*.

La figura 5.67 muestra el almacén de piezas de trabajo una vez que se agregaron los colisionadores. El modelo está formado por piezas individuales como en el caso del almacén de piezas de la empacadora, por lo que sus colisionadores se agregaron individualmente a cada pieza.

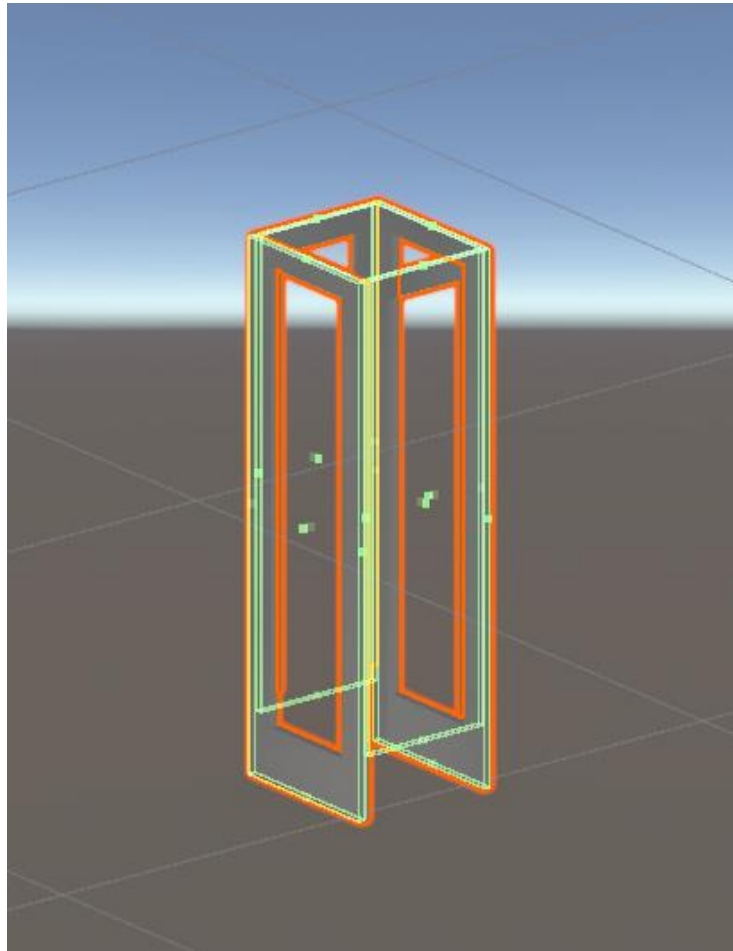


Figura 5.67. Soporte almacenador de piezas de trabajo de la empacadora con sus colisionadores.

La figura 5.68 muestra los colisionadores agregados a las guías de la empacadora. Las guías se han montado en la base y en la rampa para mostrar su posición relativa a estas piezas. Se agregó el mismo *Physics Material* al soporte de almacenamiento de piezas de trabajo y a las guías que el agregado al soporte de almacenamiento de paquetes y las guías de la empacadora (fig. 5.7).

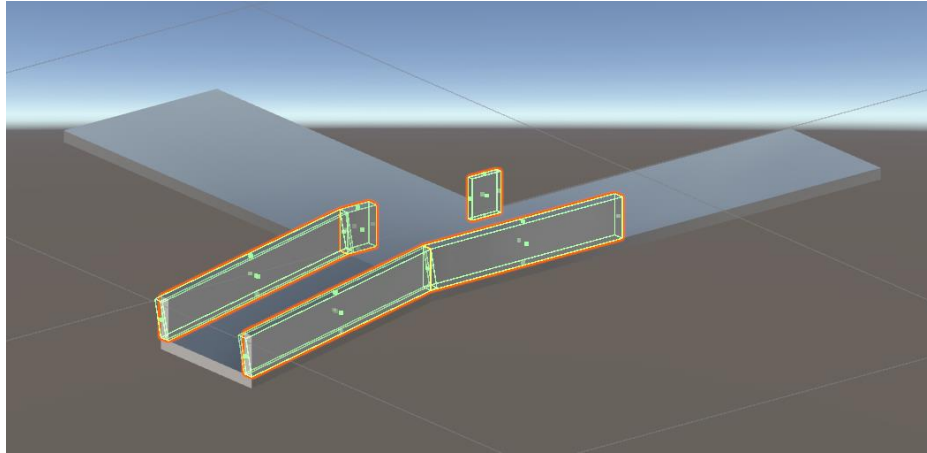


Figura 5.68. Colisionadores agregados a las guías de la impresora.

La figura 5.69 muestra los colisionadores agregados al cilindro A de la impresora. Al igual que en el caso de la empaquetadora se tienen colisionadores extremos en el cilindro que controlan el rango de movimiento del vástago (la carrera del cilindro). Para el cabezal se tienen dos colisionadores que en conjunto empujan las piezas de trabajo y sostienen las restantes en el soporte de almacenamiento. En este caso se optó por no cubrir todo el cabezal con los colisionadores, además de que uno de los colisionadores está inclinado. Este acomodo de los colisionadores fue resultado de un gran número de pruebas en las que el funcionamiento no resultaba según lo previsto. El principal problema surgió en las piezas de trabajo las cuales penetraban otros colisionadores de las mismas piezas de trabajo (figura 5.70). Como se puede observar de la figura 5.70, si el cabezal del cilindro se extendiera este empujaría dos piezas en lugar de una y como la pieza superior no tiene permitido moverse horizontalmente ocurre un atascamiento. Hacer un colisionador más pequeño para el cabezal proporciona una solución temporal, pero no resuelve el problema por completo, ya que en ocasiones la inclinación es más grande. Cambiar la forma del cabezal no fue opción en vista de que se querían utilizar colisionadores predefinidos y estos seguirían siendo cuadrados sin importar la forma que se tuviera. Se tienen otro tipo de colisionadores que se ajustan a la forma de los objetos, pero con un mayor consumo de recursos computacionales, por lo que se desechó la idea. Al final los colisionadores del cabezal mostrados en la figura 5.69 arreglaron el problema sin que fuera perceptible que el cabezal tenía esta disposición especial de colisionadores (no hay efectos visuales extraños en el acomodo de las piezas, como que la pieza atraviesa parte del cabezal). Cabe señalar que la inclinación de las piezas no es tan pronunciada como para empujarlas desde la posición del colisionador en el cabezal mostrado en la figura 5.69, pero colocarlo en esta posición fue garantía de que no ocurrirían atascamientos que pudieran impedir el buen funcionamiento del modelo. Existe otra razón para colocar los colisionadores sin cubrir todo el cabezal. Resulta que hay un error en la programación (no se sabe si de Unity o del SDK de VRChat) que, en ocasiones, después de poner un colisionador a un *GameObject*, estos comienzan a moverse por sí solos aun cuando no se les haya especificado que deben hacerlo. Al colocar colisionadores a los vástagos se presentó este efecto en varias ocasiones

(solo se presentó este error con los vástagos). Después de muchas pruebas se encontró que el error está asociado al acomodo de los colisionadores. En el caso de los colisionadores del cabezal de la figura 5.69, este acomodo permitió no solo evitar atasques en el acomodo de las piezas, sino que además solucionó el error de movimiento del vástagos en direcciones no permitidas. Al colisionador inclinado de la figura 5.69 se le colocó un *Physics Material* para controlar la fricción con las piezas de trabajo de cero (ver figura 5.7).

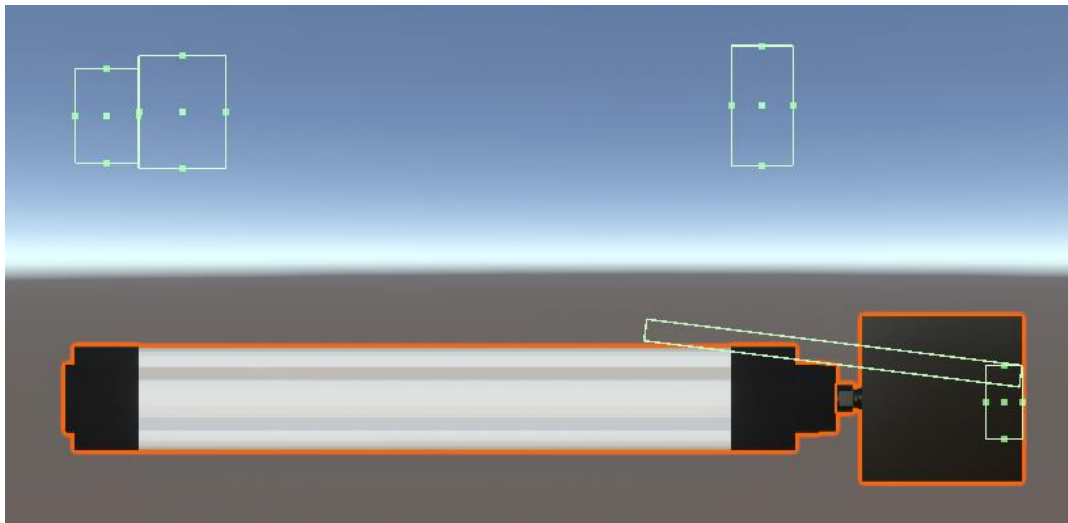


Figura 5.69. Colisionadores agregados al cilindro A de la estampadora.

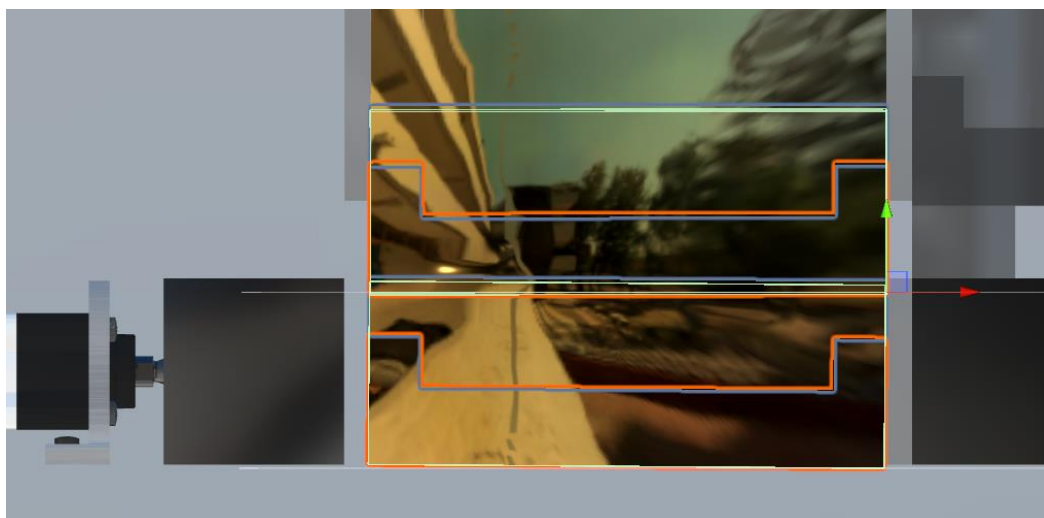


Figura 5.70. Se muestran dos piezas de trabajo de la estampadora donde se puede observar que el colisionador superior penetra una parte del colisionador inferior.

La figura 5.71 muestra los colisionadores agregados al cilindro B de la estampadora, el cilindro que realiza el estampado. Al igual que para el cilindro A, se tienen colisionadores extremos en el cilindro y un colisionador central en el vástago que controlan la carrera del cilindro. Se puede notar que el cabezal tiene colisionadores extremos y no uno que lo recubra por completo. Esto es para evitar que el cabezal choque con la pieza de trabajo al momento de bajar. Si se optaba por colocar un colisionador que cubriera el cabezal, podrían ocurrir colisiones entre los colisionadores del cabezal y el de la pieza de trabajo, lo que provocaría un efecto de rebote poco deseado. Los colisionadores colocados en los extremos del cabezal tienen el objetivo de evitar que la pieza de trabajo se salga de los límites donde ocurre la deformación del material (que es una animación), ya que esto a menudo ocurre si no se evita que la pieza se mueva de estos límites.

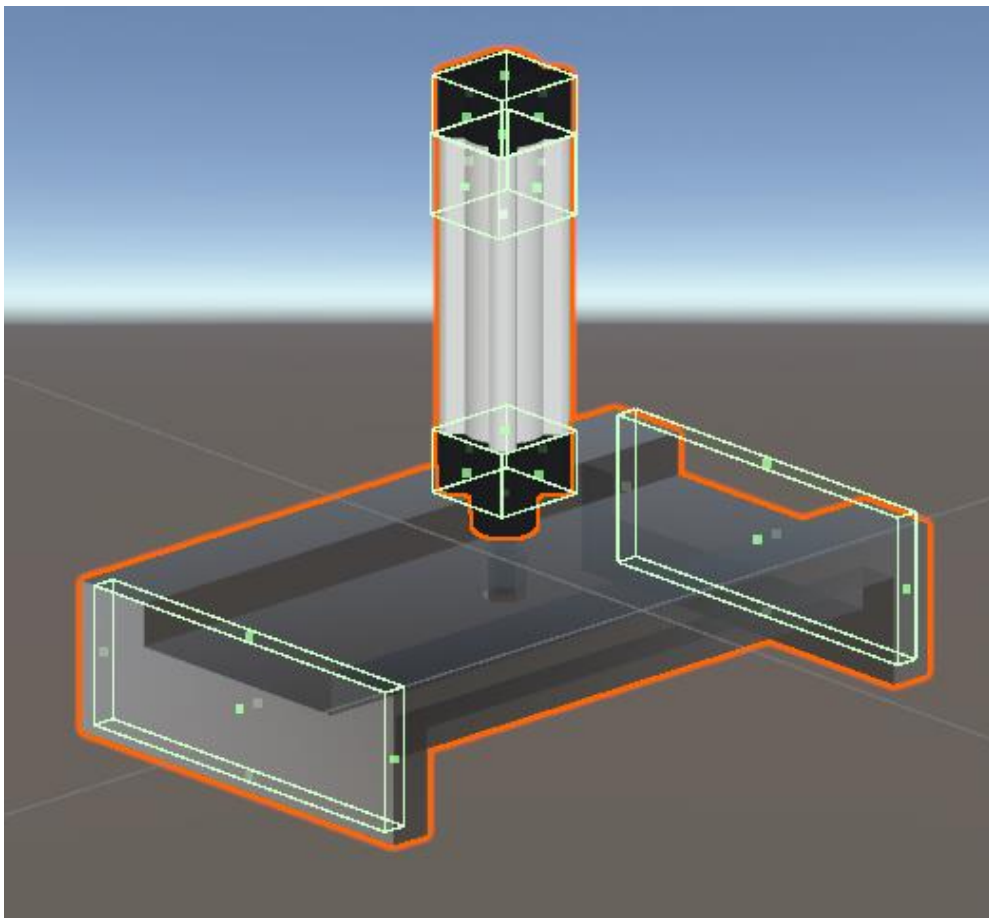


Figura 5.71. Colisionadores agregados al cilindro B de la estampadora.

La figura 5.72 muestra los colisionadores agregados al cilindro C de la estampadora. Se puede observar que los colisionadores que controlan la amplitud del movimiento del vástago se han colocado arriba del cilindro. Esto se hizo así para poder colocar un colisionador adicional al vástago para que, en circuitos construidos de manera incorrecta en los que primero se active el cilindro C y luego el A, la pieza de trabajo choque con el colisionador del vástago y provoque el efecto que se esperaría en la realidad. Para los colisionadores de los cilindros no fue necesario agregar *Physics Materials* ya que estos no tienen deslizamiento con otros colisionadores, son para restringir el movimiento de los vástagos.

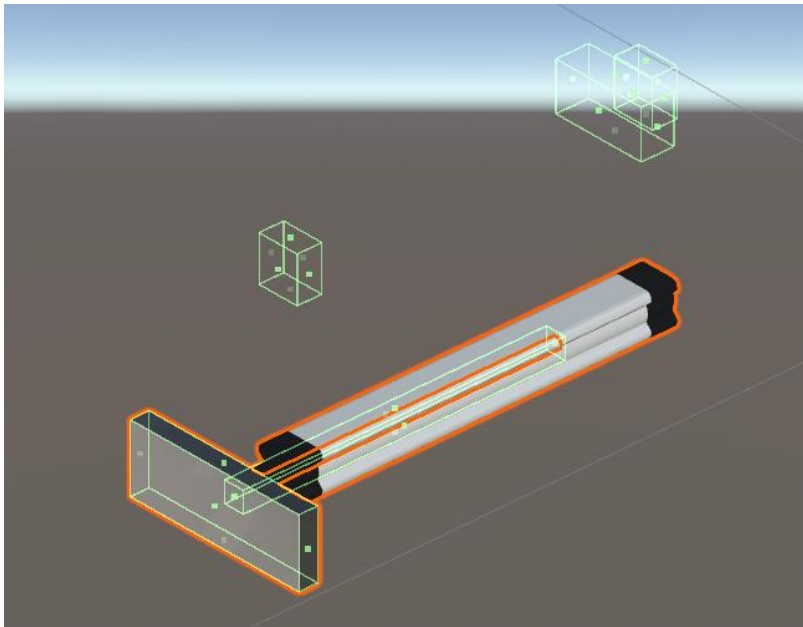


Figura 5.72. Colisionadores agregados al cilindro C de la estampadora.

Se agregaron componentes *Rigidbody* a los ensambles de vástago y cabezal de los cilindros para poder controlar su movimiento por medio de física (aplicación de un fuerza). La figura 5.73 muestra el componente *Rigidbody* agregado al vástago A de la empacadora. Para este ensamble se eligió una masa de 5 [kg] y se deshabilitó la opción *Use Gravity* para que no lo afectara la gravedad. Se bloquearon los movimiento lineales y angulares en todas las direcciones a excepción del movimiento lineal en la coordenada Z (su dirección de movimiento). Para el vástago B también se agregó un componente *Rigidbody* en este caso con una masa de 1 [kg], se deshabilitó la opción *Use Gravity* y se restringieron los movimientos lineales y angulares en todas direcciones a excepción del movimiento lineal en la dirección Y (fig. 5.74). Para El vástago C se eligió una masa de 2 [kg] y también se desactivó la opción *Use Gravity*. Fueron deshabilitadas los movimientos en todas direcciones a excepción del movimiento lineal en la dirección X (fig. 5.75). Todas las masas fueron elegidas arbitrariamente

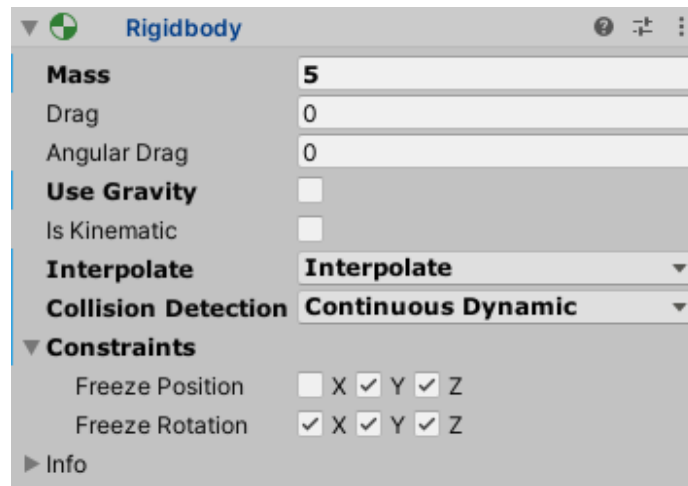


Figura 5.73. Componente *RigidBody* agregado al vástago A de la estampadora.

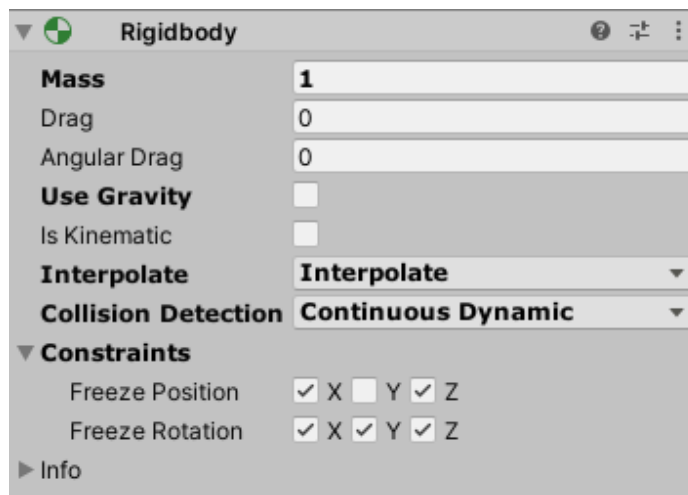


Figura 5.74. Componente *RigidBody* agregado al vástago B de la estampadora

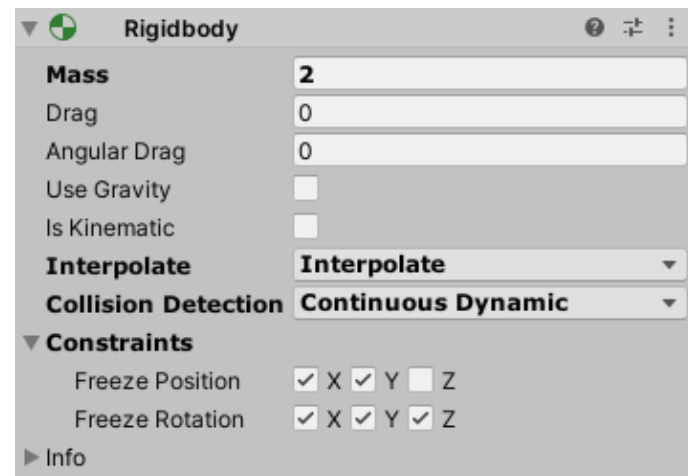


Figura 5.75. Componente *RigidBody* agregado al vástago C de la estampadora.

Para el control del movimiento de los vástagos se utilizó el mismo *script Vastago* mostrado en la figura 5.22, simplemente se modificaron los vectores de fuerza y se colocaron los respectivos *GameObjects* responsables del control del movimiento de los vástagos del gemelo digital (los vástagos de la mesa de trabajo). La figura 5.76 muestra el *script Vastago* agregado al vástago A de la empacadora donde se observa que se agregó una fuerza de 200 [N] en dirección X. La variable *goVastago* se llena con el *GameObject* del vástago del cilindro A de la mesa de trabajo. La figura 5.77 muestra el *script Vastago* agregado al vástago B, en este caso la fuerza es de -10 [N] en dirección Y y la variable *goVastago* se llena con el *GameObject* del vástago del cilindro B de la mesa de trabajo. Para el vástago C se tiene una fuerza de -100 [N] en dirección Z y la variable *goVastago* se llena con el *GameObject* del vástago del cilindro C de la mesa de trabajo (fig. 5.78).

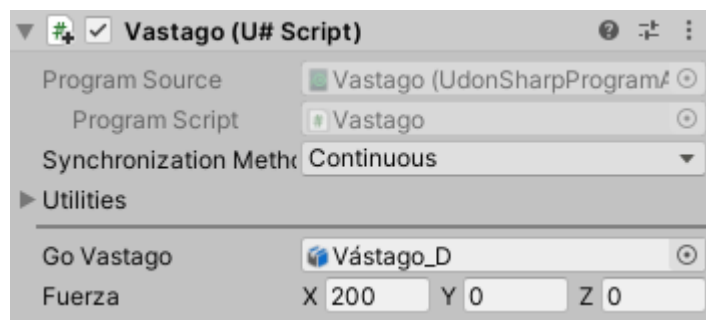


Figura 5.76. *Script Vastago* agregado al vástago A de la empacadora.

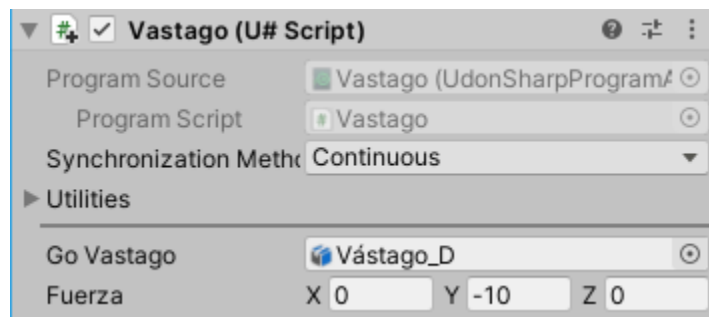


Figura 5.77. *Script Vastago* agregado al vástago B de la empacadora.

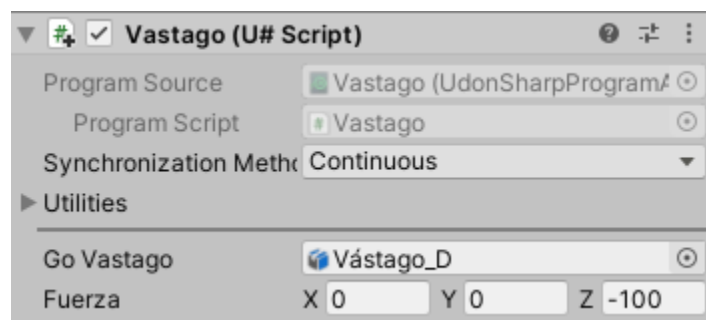


Figura 5.78. *Script Vastago* agregado al vástago C de la empacadora.

La creación de las piezas de trabajo de la estampadora se realizó mediante los scripts *CreaPieza*, *DetectorPieza* y *DestructorPiezas* vistos en la programación de la empacadora. Como en el caso del gemelo digital de la empacadora, se agregaron *GameObjects* (a los que se les agregaron colisionadores con el parámetro *is Trigger* activado) para que sirvieran de elementos de detección de las piezas de trabajo. La figura 5.79 muestra los colisionadores de los *GameObjects* agregados a la estampadora para funciones de detección, de manera similar a como fueron agregados para la empacadora en la figura 5.27. En la misma figura 5.79 observamos los nombres con los que nos referiremos a estos *GameObjects* para simplificar la descripción. Los scripts *CreaPieza* y *DestructorPieza*, se agregaron al *GameObject* del Detector 1; mientras que el script *DetectorPieza* se agregó al *GameObject* Detector 2. Esto es exactamente lo mismo que se hizo para el caso de la empacadora.

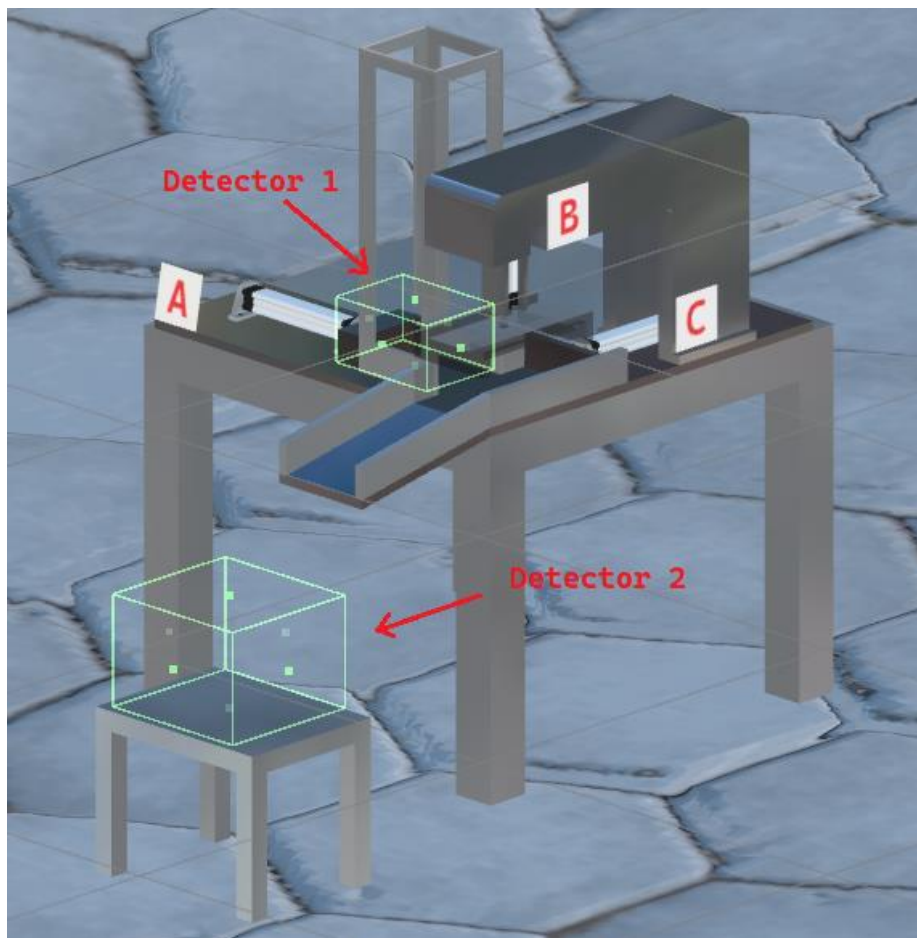


Figura 5.79. Colisionadores agregados a la estampadora, para hacer funciones de detección de las piezas de trabajo.

El *script CreaPieza* se muestra en la figura 5.80 una vez agregado al *GameObject* Detector 1, donde ya se han agregado los *GameObjects* necesarios para las variables: La variable *refDetPieza* que es de tipo *DetectorPieza* y se utiliza para acceder a las variables del *script DetectorPieza*, se le agregó el *GameObject* Detector2 ya que en este se contiene dicho *script*. La variable *goBoton* se llena con el *GameObject* que contiene la animación del botón enclavado. La variable *goFRL* se llena con el *GameObject* que tiene la animación de la válvula de paso de la unidad de mantenimiento. La variable *cajaContenedora* se llena con el *GameObject* de la caja contenedora. También se han inicializado los vectores de posición y rotación de la pieza de trabajo de la estampadora. Los arreglos: *Piezas*, *copiaPiezas* y *copiaPiezas2* se muestran en la figura 5.81. Se puede observar que el arreglo *Piezas* solo tiene un elemento, ya que solo se cuenta con una pieza de trabajo para estampar, a diferencia de la empacadora que contaba con 8 distintos paquetes para empacar. Los arreglos *copiaPiezas* y *copiaPiezas2* tiene una dimensión de 12 para guardar las piezas que se van creando en escena (en la figura solo se observa el tamaño del arreglo *copiaPiezas* para no hacer más grande la figura). Además, se debe poner como falsa la variable empacadora para que se ejecuten las partes del código correspondientes a la estampadora.

La figura 5.82 muestra el *script DestructorPiezas* una vez que se agregó al *GameObject* Detector1. La variable *refCreaPieza* se llena con el *GameObject* que contiene el *script CreaPieza*, que es el mismo *GameObject* donde se agregó el *script DestructorPiezas*, Detector1.

La figura 5.83 muestra el *script DetectorPieza* una vez que se agregó como componente del *GameObject* Detector2. En este *script* se utilizan las variables del *script CreaPieza* por lo que se coloca el *GameObject* (Detector1) donde se agregó este *script* en la casilla para la variable *refCreaPieza*. También se inicializó el arreglo *copiaPiezas* en 3. De este modo se pueden almacenar 3 piezas de trabajo antes de que se borren todas las piezas del mismo.

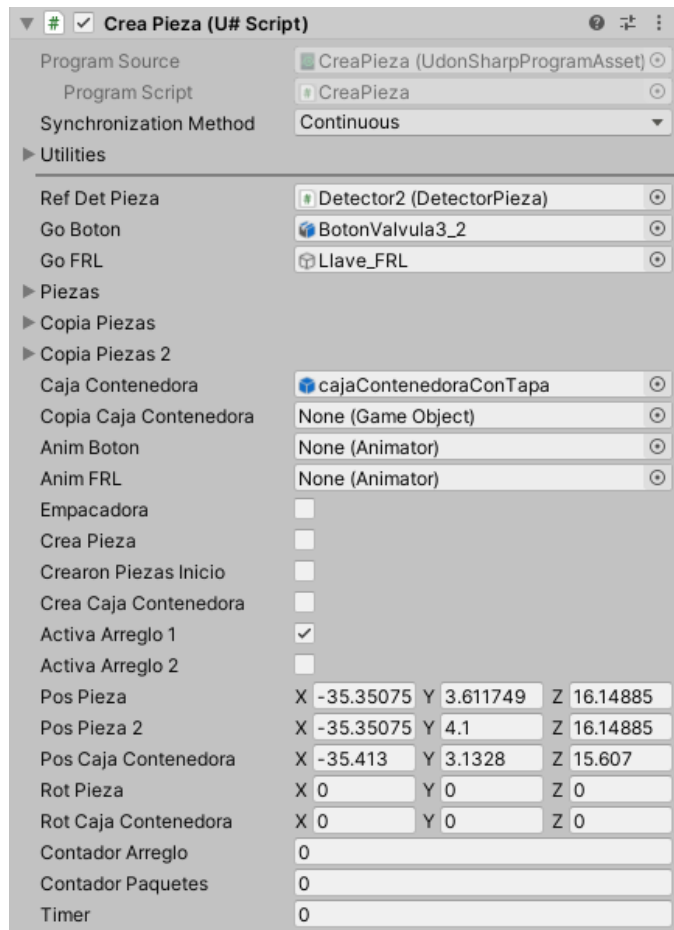


Figura 5.80. Script *CreaPieza* agregado al *GameObject Detector1*.

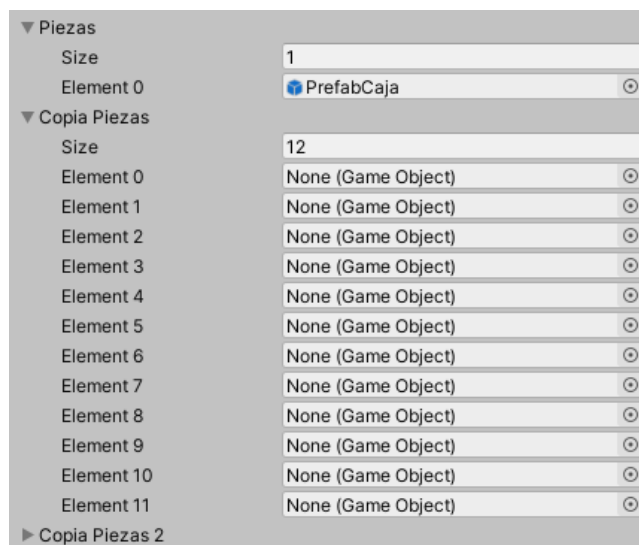


Figura 5.81. Script *CreaPiezas* con los arreglos *Piezas* y *copiaPiezas* expandidos para ver sus componentes. El arreglo *copiaPiezas2* tiene la misma dimensión que el arreglo *copiaPiezas*.

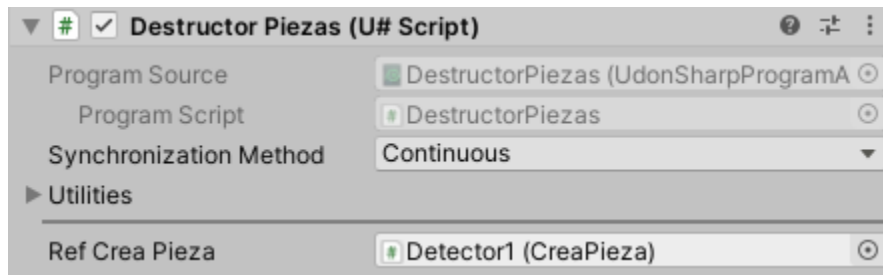


Figura 5.82. *Script DestructorPiezas* una vez agregado como componente del *GameObject Detector1*.

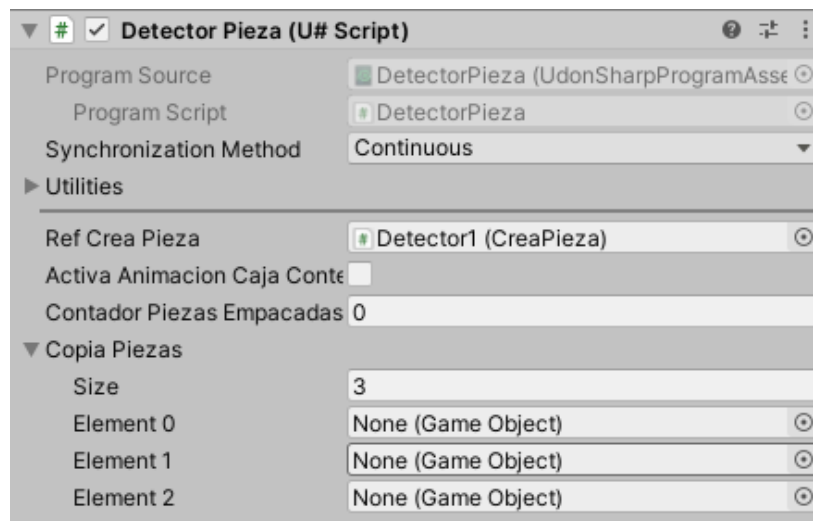


Figura 5.83. *Script DetectorPieza* una vez agregado como componente del *GameObject Detector2*.

5.3.1. Animación de la deformación de la pieza de trabajo

Se recordará de la sección 3, que la pieza de trabajo de la estampadora se realizó en piezas y no como una pieza solida (ver figura 3.45). Esta pieza se hizo de esta manera para poder hacer una animación del estampado de la pieza. Hay diversas maneras de hacer una animación; aquí se modificó únicamente el escalado de las partes componentes de la pieza para lograr la simulación de deformación. La figura 5.84 muestra la pieza de trabajo de la estampadora separada en sus partes componentes. Estas partes componentes no se mueven de manera relativa unas a otras, ya que forman parte del mismo *GameObject* (como hijos del *GameObject*), por lo que al impulsarlas con el cabezal del vástago se moverán como una sola pieza. La animación se realizó directamente en Unity. La escala de un *GameObject* se puede modificar en las tres direcciones que conforman el espacio local del objeto. En la figura 5.84, se puede observar los ejes coordenados respecto de la pieza de trabajo y en la esquina superior derecha se observa el nombre de los ejes coordenados. La animación consiste en 3 clips: El primero representa la pieza sin deformar, el segundo la pieza deformándose y el tercero la pieza deformada.

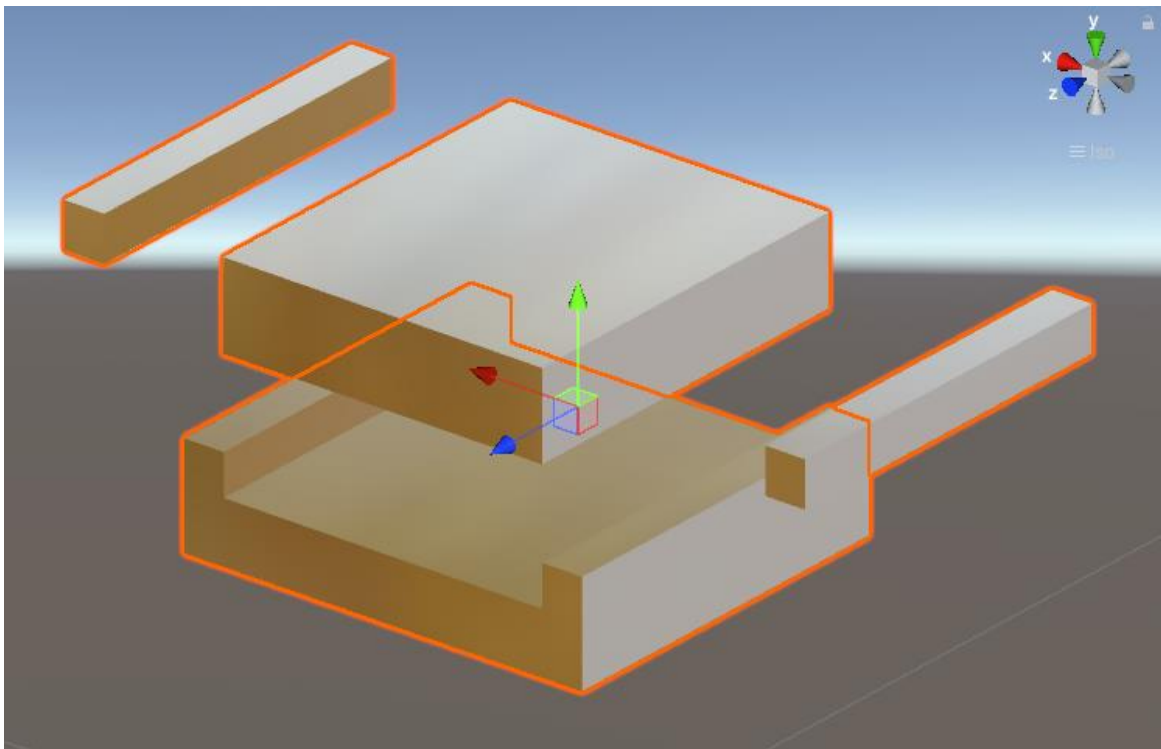


Figura 5.84. Pieza de trabajo de la estampadora separada en sus partes componentes.

Cuando una pieza de trabajo se crea en escena, inmediatamente comienza la primera animación, la pieza sin deformar o la pieza “normal”. La figura 5.85 muestra las ventanas *Scene* y *Animation* donde se observa el estado de la animación “Sin deformar”. Cabe resaltar que la pieza de trabajo se ha mantenido en partes separadas para observar el cambio en las piezas individuales, pero éstas deben permanecer juntas. Durante este estado “sin deformar” la pieza no tiene ningún cambio y solo está a la espera de que se reciba la indicación de que se debe iniciar la deformación. La figura 5.86 muestra las ventanas *Scene* y *Animation* pero ahora las piezas se ven “deformadas”, pero como se mencionó, simplemente se hizo un escalado de las componentes. Las componentes superiores de la pieza se escalaron en las direcciones Y y Z. El componente inferior solo se escaló en la dirección Z ya que esta pieza tiene la forma que se quiere obtener del estampado. El escalado en las direcciones Z de todas los componentes se aumentó en 50% de los componentes originales y la reducción en la dirección Y de las componentes superiores se realizó en 99%. Esta última reducción no se hizo del 100% porque las componentes tienden a superponerse unos sobre otros y durante la ejecución de la simulación se observa un efecto de distorsión del material que tienen asignados los componentes. En la parte inferior de la figura 5.86 se puede observar una escala de tiempo. En un tiempo cero la escala de los componentes de la pieza se deja sin cambio (se dejan las dimensiones originales de los componentes de la pieza). En un tiempo de un segundo se escalan los componentes como se indicó anteriormente. Unity interpola entre las posiciones extremas de la escala para completar la animación, por lo que no se requiere más trabajo.

El último clip de animación, que corresponde a la pieza de trabajo deformada, se ejecuta inmediatamente después de ocurrido el estampado de la pieza (después de que se escalaron los componentes de la pieza) y se repite de manera continua hasta que la pieza de trabajo desaparece de escena. En esta última animación las escalas de los componentes de la pieza de trabajo se mantienen constantes e iguales a las escalas de la pieza deformada. No se muestra una imagen correspondiente a este último clip de animación ya que se ve exactamente igual a la figura 5.86. Para las animaciones de la pieza sin deformar y la totalmente deformada no es necesario agregar modificaciones en la escala de tiempo. En este caso para la pieza sin deformar de la figura 5.85 se colocaron dos estados: uno inicial en 0 segundos y uno final en 1 segundo, pero en ambos casos se mantuvo la escala de la pieza constante. Si solo se colocase un estado en 0 segundos la animación funciona de manera similar.

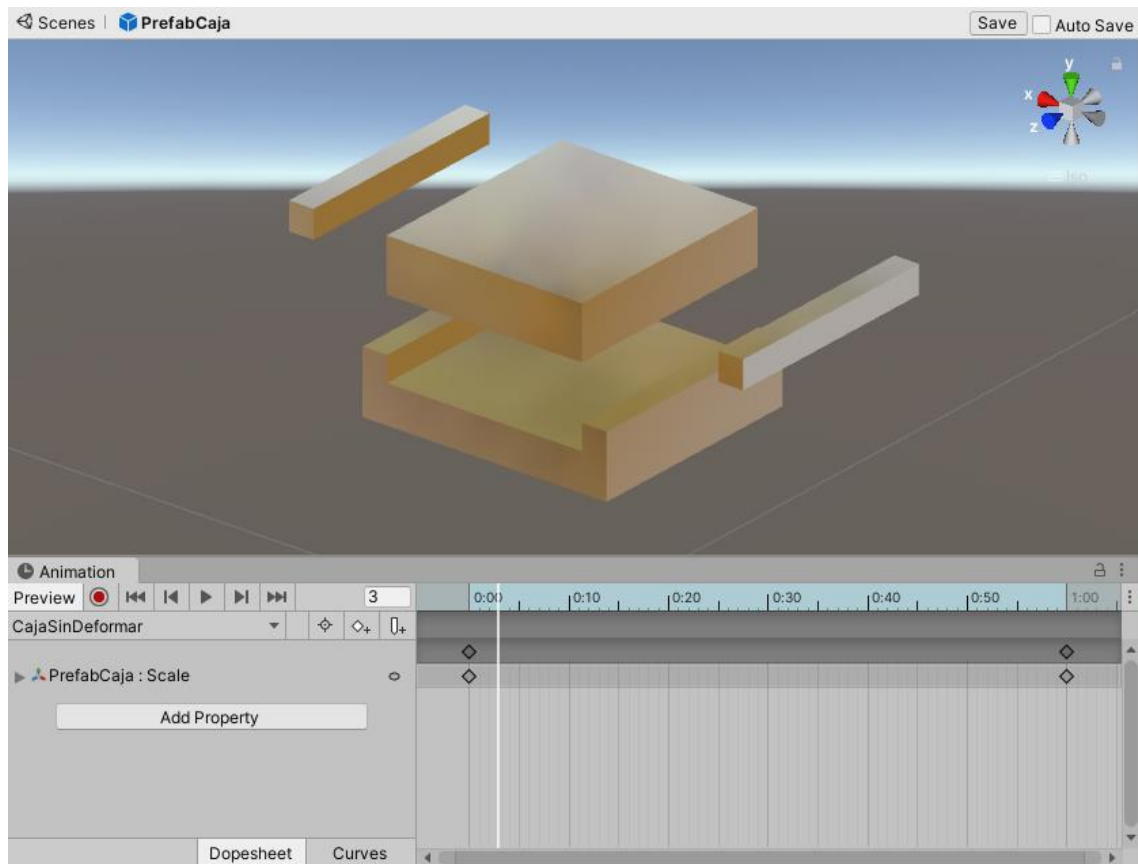


Figura 5.85. Ventanas *Scene* y *Animation* en la cuales se realiza la animación del estampado de la pieza de trabajo. La pieza no está deformada (escalada).

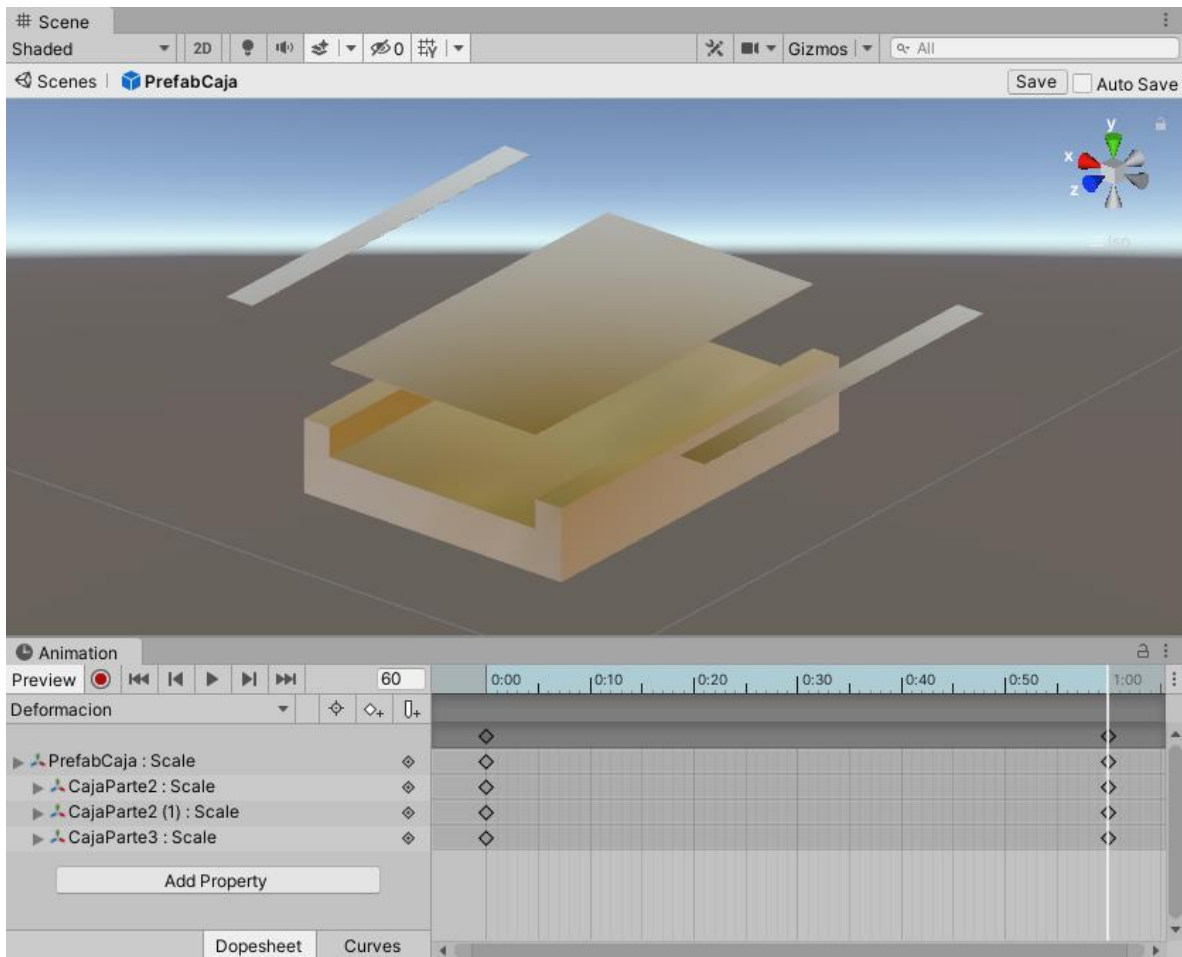


Figura 5.86. Se han escalado los diversos componentes de la pieza de trabajo para que durante la animación parezca que esta se está deformando.

La figura 5.87 muestra la ventana *Scene* con la pieza de trabajo deformada y con sus componentes en los lugares adecuados, por lo que se observa una sola pieza. La figura 5.88 muestra la ventana *Animator* donde se pueden observar los tres clips de animación antes mencionados. Al crearse una pieza en escena se ejecuta de inmediato el primer clip de animación (la caja sin deformar). Este clip permanece activo hasta que se indique que se ejecute la animación de la deformación. Inmediatamente después de que se ejecuta la animación de la deformación se inicia la animación de la pieza deformada y se queda en este clip hasta que se destruye la pieza de la escena (ya no regresa a su estado inicial).

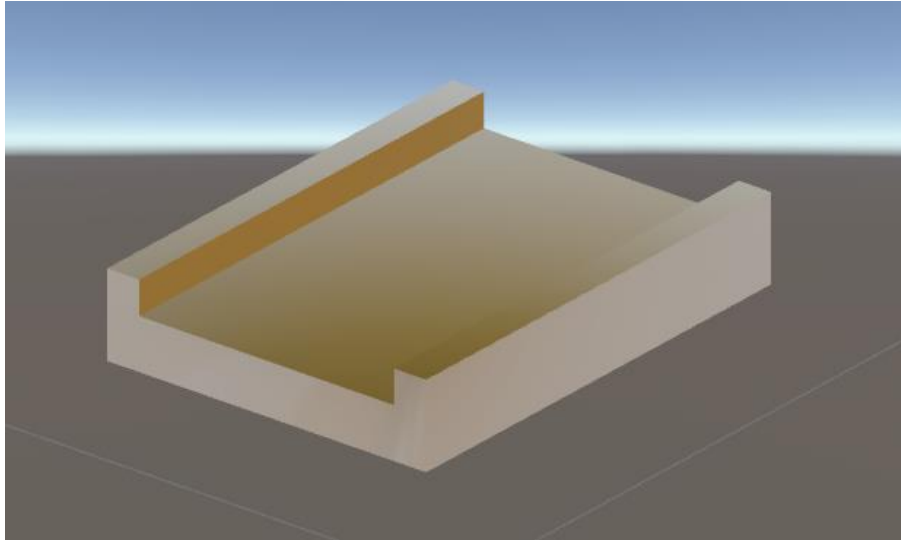


Figura 5.87. Pieza de trabajo de la estampadora “deformada”. Los componentes de la pieza no son notorios.

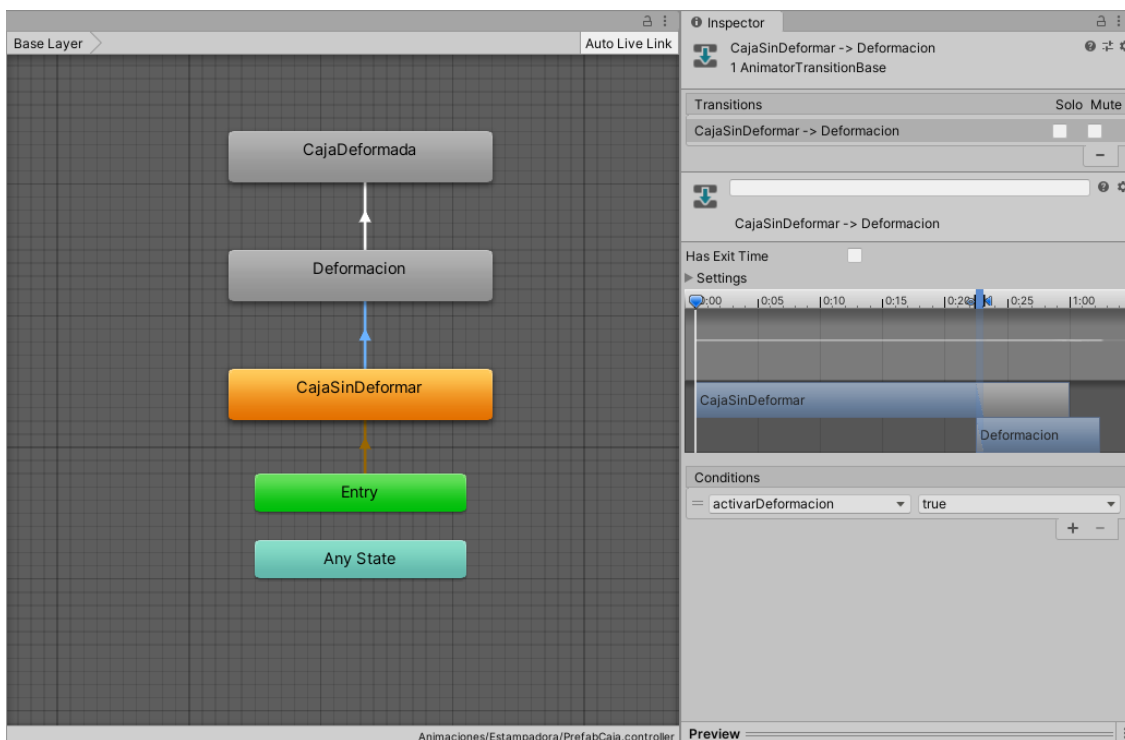


Figura 5.88. Ventana *Animator* e *Inspector*. En la ventana *Animator* se observan los diversos clips que conforman la animación de la deformación de la pieza de trabajo de la estampadora. En la ventana *Inspector* se observa que la variable en el *Animator* que controla el inicio de la animación de la deformación se llama *activarDeformación*.

Para controlar la deformación de la pieza de trabajo de la estampadora, se requiere agregar un *script* que indique el momento adecuado para iniciar la animación y un nuevo colisionador que sirva de detector de las piezas de trabajo. La figura 5.89 muestra un colisionador en modo *Trigger* para hacer labores de detección. Este colisionador se agregó a un nuevo *GameObject* vacío y dicho *GameObject* se colocó como hijo del vástago B de la estampadora. Con esto se logra que cuando se mueva el vástago también se mueva el colisionador (detector) ya que pertenecen al mismo *GameObject*. A este nuevo *GameObject* se le agregó el *script* mostrado en la figura 5.90, como uno de sus componentes, para controlar el inicio de la animación. Cuando una pieza de trabajo se encuentre por debajo del colisionador situado en el cabezal B y el cabezal se mueva hacia abajo, en el momento que el colisionador detecte la pieza (la toque), en ese momento se empezará a ejecutar la animación de la deformación. La animación solo se ejecutará si el objeto que se detecta tiene la etiqueta correcta, como se verá en breve.

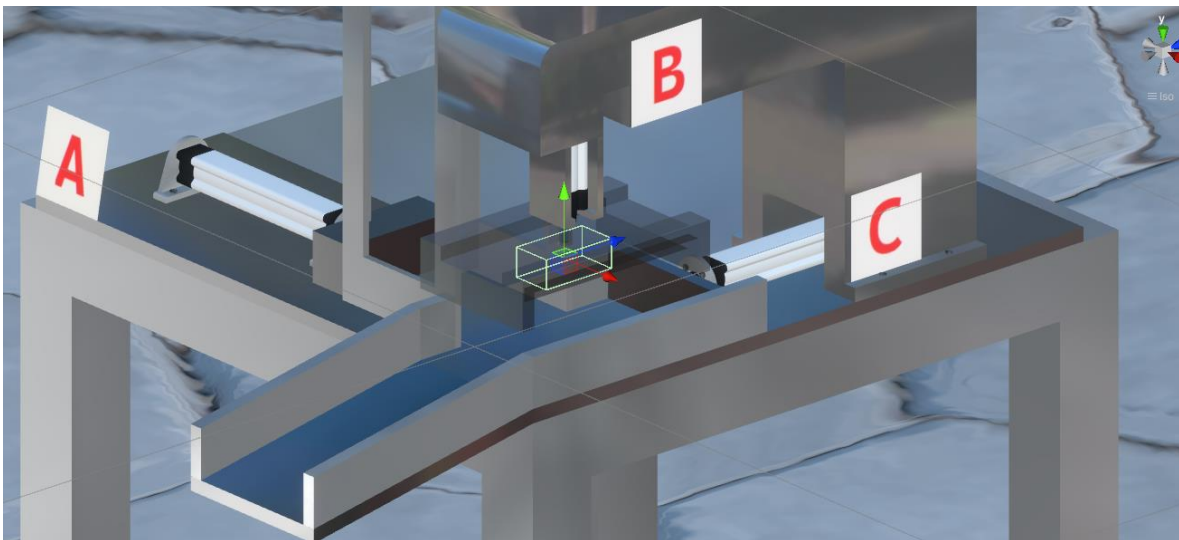


Figura 5.89. Se agregó un *GameObject* vacío con un colisionador en modo *Trigger* al cabezal del vástago B de la estampadora para detectar la pieza de trabajo.

En la figura 5.90, la línea 6 crea la clase *Deformacion* que se compone de las líneas 7 a 32. La línea 8 declara la variable privada de tipo *Animator* *anim*. Esta variable se utiliza para acceder al componente *Animator* de las piezas de trabajo que se van creando en escena (una a la vez cuando son detectadas).

La línea 13 declara una variable pública de tipo booleano llamada *animacionActivada* la cual se inicializa como falsa y se utiliza para indicar el momento en el que se debe activar la animación de la deformación de la pieza.

La línea 15 declara la función *OnTriggerEnter* la cual se encarga de detectar los objetos que entran al detector (*Trigger*). Dentro de los parámetros de la función se encuentra la instrucción:

Collider colisionador

Este parámetro ya fue explicado, pero se recalca que al colisionador que sea detectado se le asignará la variable de tipo *Collider colisionador*.

En la línea 17 se declara la instrucción *if*:

if(colisionador.gameObject.name == "PrefabCaja(Clone)")

En la cual se indica que si el nombre del colisionador que se detectó es igual a la cadena de caracteres: *PrefabCaja(Clone)*, entonces se ejecutan las sentencias en el cuerpo de la instrucción. Cabe señalar que cada una de las piezas de trabajo que se crean en escena llevan este nombre ya que *PrefabCaja* es el nombre del *GameObject* de la pieza de trabajo de la estampadora. Cuando se crea una nueva pieza de trabajo en escena a partir de esta pieza de trabajo predefinida, a cada una de las copias se les agregara el distintivo (*Clone*). De este modo se completa el nombre que se muestra dentro de la instrucción *if*.

La línea 19, dentro de la instrucción *if* de la línea 17, indica que se debe activar la animación (la variable *animaciónActivada* se hace verdadera).

La línea 20; dentro de la misma instrucción *if* de la línea 17, asigna el componente *Animator* del *GameObject* cuyo colisionador se ha detectado (la pieza de trabajo) a la variable *anim*, de modo que se pueda acceder a la animación de la deformación de la pieza de trabajo desde otras funciones (en este caso desde la función *Update*).

La línea 24 corresponde a la función *Update* cuyo cuerpo está formado por las líneas 25 a 31. Dentro de la función *Update* se encuentra un *if* que se activa cuando la variable *animacionActivada* es verdadera (cuando se detectó una pieza de trabajo).

La línea 28 dentro de la instrucción *if*:

anim.SetBool("activarDeformacion", animacionActivada);

indica que la variable "*activarDeformacion*" toma el valor de la variable *animacionActivada*, que si llega a este punto debe ser verdadera. La variable "*activarDeformacion*" de tipo booleano es declarada dentro del componente *Animator* del *GameObject* de la pieza de trabajo (ver figura 5.88) y al ser verdadera dará paso a la animación de la deformación para la pieza de trabajo correspondiente (la que fue detectada).

Por último, la línea 29 establece que la variable *animacionActivada* se vuelve falsa por lo que la instrucción *if* de la línea 26 se volverá a activar hasta que se detecte una nueva pieza de trabajo.

```

1  using UdonSharp;
2  using UnityEngine;
3  using VRC.SDKBase;
4  using VRC.Udon;
5
6  public class Deformacion: UdonSharpBehaviour
7  {
8      // variable para acceder a la animación de la deformación de la pieza de trabajo
9      private Animator anim;
10
11     // variable para indicar cuando se debe activar la animación de
12     // la deformación de la pieza de trabajo
13     public bool animacionActivada = false;
14
15     void OnTriggerEnter(Collider colisionador)
16     {
17         if (colisionador.gameObject.name == "PrefabCaja(Clone)")
18         {
19             animacionActivada = true;
20             anim = colisionador.gameObject.GetComponent<Animator>();
21         }
22     }
23
24     void Update()
25     {
26         if (animacionActivada)
27         {
28             anim.SetBool("activarDeformacion", animacionActivada);
29             animacionActivada = false;
30         }
31     }
32 }

```

Figura 5.90. Script *Deformacion* utilizado para controlar el inicio de la deformación de la pieza de trabajo de la estampadora.

Para continuar con la programación de la estampadora se tiene que agregar un componente *Rigidbody* a la pieza de trabajo. La figura 5.91 muestra el componente *Rigidbody* agregado a la pieza. Se eligió una masa de 1 *Kg* y ninguna restricción de movimiento fue requerida, ya que esta pieza debe moverse libremente. La opción *Use Gravity* fue activada para que la pieza fuera afectada por la gravedad. Como la pieza de trabajo está formada por varios componentes se requiere establecer uno de ellos como padre de los demás, ya que de esta forma se logra que todas las piezas se muevan juntas como un único cuerpo. El componente *Rigidbody* se debe agregar solo al objeto que sirve como padre de los demás. A la pieza de trabajo también se colocó un colisionador para que pudieran interactuar con los otros colisionadores de la estampadora (fig. 5.92). En este caso se agregó un colisionador global al padre del *GameObject* de la pieza de trabajo que cubre la pieza en su totalidad. A este colisionador le fue agregado un *Physics Material* para controlar la fricción con las superficies externas a la pieza. La figura 5.93 muestra el *Physics Material* colocado al colisionador de la pieza de trabajo donde se observa que se ha elegido una fricción estática de 0.055 y una dinámica de 0.05. Una vez más se establecieron los valores, tanto para la masa como para la fricción de las piezas de trabajo, después de una serie de pruebas en las que se determinó que los movimientos de las piezas eran suaves y estables.

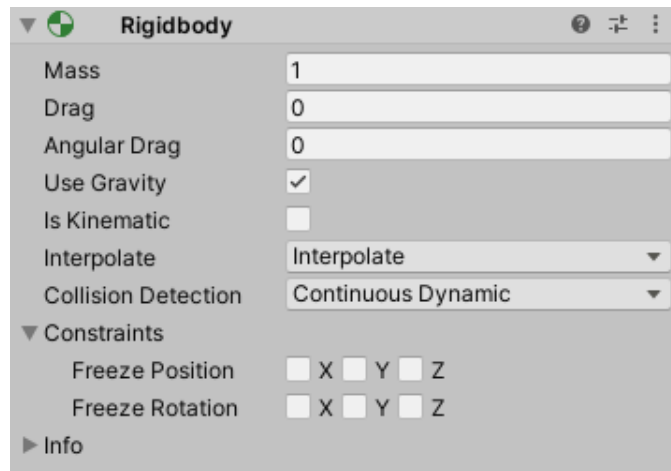


Figura 5.91. Componente *Rigidbody* agregado a la pieza de trabajo de la estampadora.

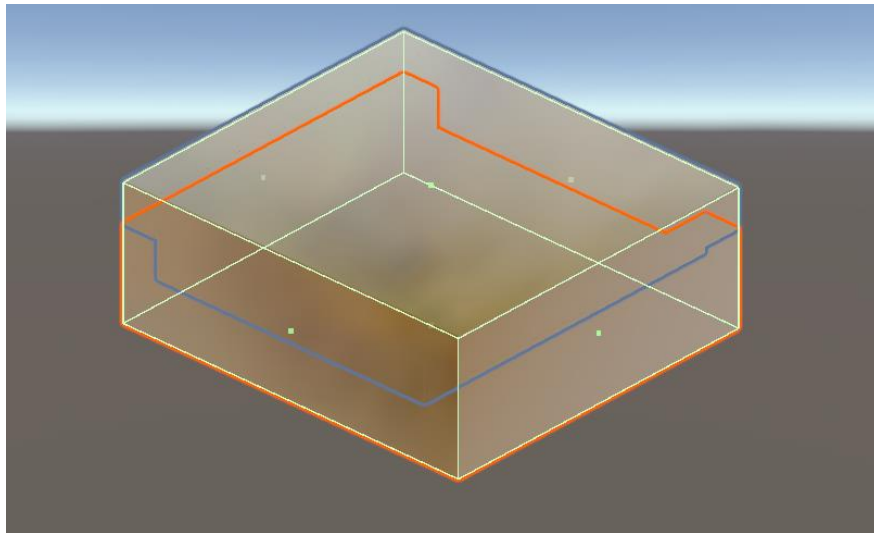


Figura 5.92. Colisionador *Box Collider* agregado a la pieza de trabajo de la estampadora.

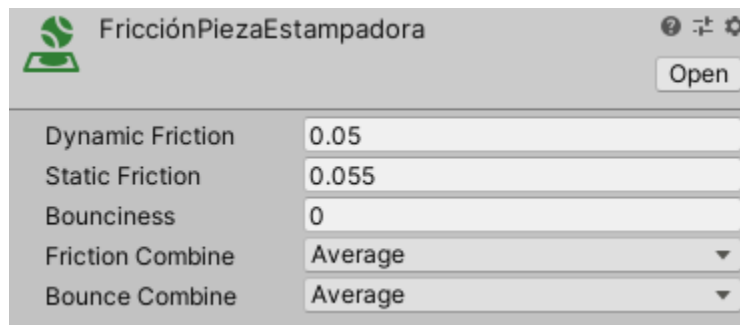


Figura 5.93. *Physics Material* agregado al colisionador de la pieza de trabajo de la estampadora para controlar la fricción de la pieza.

En la figura 5.71 se observó que el cabezal del vástago B no tiene colisionadores que lo cubran por completo. Como el cabezal no tiene colisionador, ocurre un problema cuando los usuarios no ingresan una secuencia de movimiento correcta. Por ejemplo, si el usuario primero activa el movimiento B+ y luego el A+, entonces el cabezal del cilindro B debería impedir el paso de la pieza de trabajo a la posición de estampado, pero esto no ocurrirá en vista de que no hay colisionador en el cabezal del cilindro B que lo impida. Durante las pruebas de desarrollo se intentó agregar colisionadores al cabezal del cilindro B de tal forma que se impidiera este movimiento, sin embargo, la adición de colisionadores provocaba el efecto indeseado de rebote cuando llegaban a chocar los colisionadores del cabezal y el de la pieza de trabajo. En vez de usar un colisionador situado en el cabezal del vástago se agregó código adicional y otros colisionadores por fuera del gemelo digital para lograr el comportamiento deseado. Los colisionadores agregados se pueden observar en la figura 5.94. En esta figura los colisionadores A y B fueron agregados como componentes de nuevos *GameObjects* vacíos los cuales se hicieron hijos de los vástagos de los cilindros A y B respectivamente, por lo que los colisionadores (y los *GameObjects*) se mueven junto con los vástagos de los cilindros A y B. El detector es también componente de un *GameObject* vacío e hijo del vástago del cilindro A, por lo también se mueve junto con el colisionador A. Para el funcionamiento se establece que si las piezas de trabajo han sido creadas (si están activadas la válvula de la unidad de mantenimiento y el botón enclavado), entonces el colisionador B está activado y puede impedir el movimiento del colisionador A. Si no han sido creadas las piezas de trabajo, entonces el colisionador B está desactivado y no impide el movimiento del colisionador A. Por lo tanto, si hay piezas de trabajo creadas y se expande el cilindro B (B+), el colisionador B baja (junto con el cabezal B) y esto obstruye el movimiento del colisionador A; como se puede observar en la figura 5.95, lo que produce el efecto deseado en el que el cilindro A no se expande porque hay una pieza de trabajo entre el cabezal del vástago B y el cabezal del vástago A.

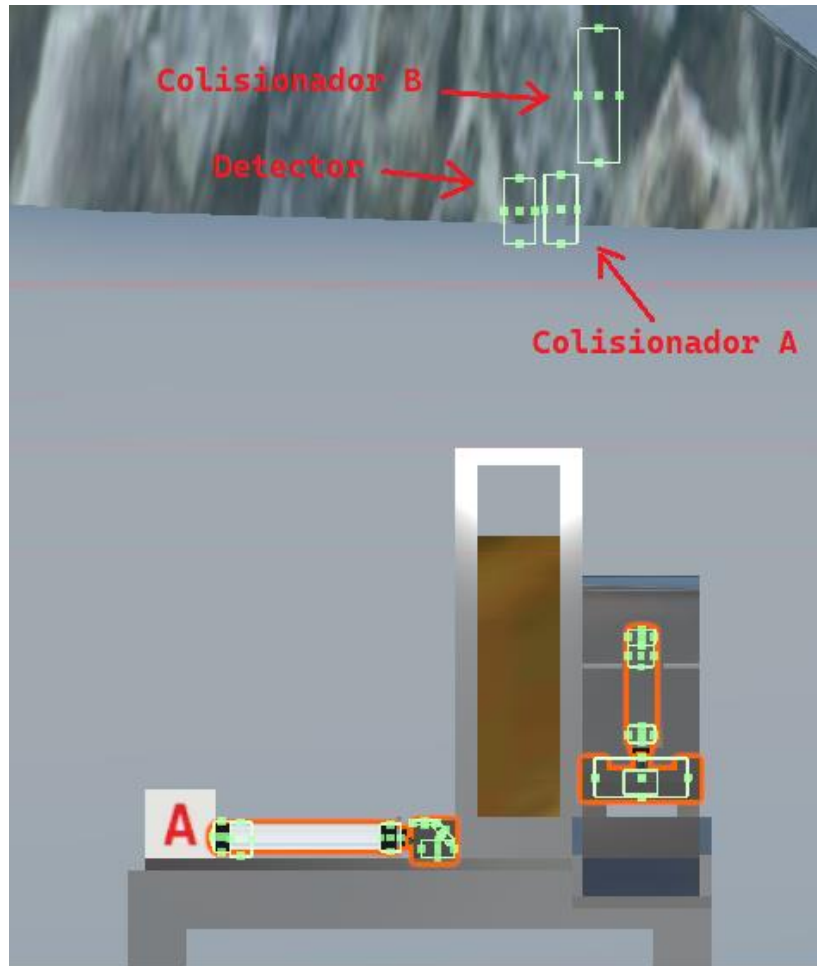


Figura 5.94. Colisionadores adicionales agregados a los cilindros A y B para controlar el choque de las piezas de trabajo con el cabezal del cilindro B de la estampadora.

Si no hay piezas de trabajo, entonces el colisionador B está desactivado y aunque baje no impedirá el movimiento de expansión del cilindro A. De este modo si no se han creado las piezas de trabajo y se expandió el cilindro B no habrá colisionador que impida la expansión del cilindro A (lo que se esperaría que pasara ya que no hay pieza de trabajo que obstruya la expansión del cilindro A).

En el caso en el que primero se expande el cilindro A y luego el B (lo que ocurriría si el usuario establece la secuencia de movimiento correcta) y además ya se crearon las piezas de trabajo, el colisionador A se posiciona del otro lado del colisionador B como se muestra en el figura 5.96. Si el usuario intentara regresar el vástago A antes de regresar el B, entonces ocurriría otra colisión en la que el colisionador B impediría el retroceso completo del vástago A. Para este caso se agregó el tercer colisionador llamado *Detector*. En la figura 5.96 se observa que el *Detector* queda en medio de los colisionadores A y B. Si el cilindro A retrocede antes que el B, entonces el detector tocará el colisionador B y este se desactivará momentáneamente para permitir la contracción completa del cilindro A, esto se logra mediante el código que se verá a continuación.

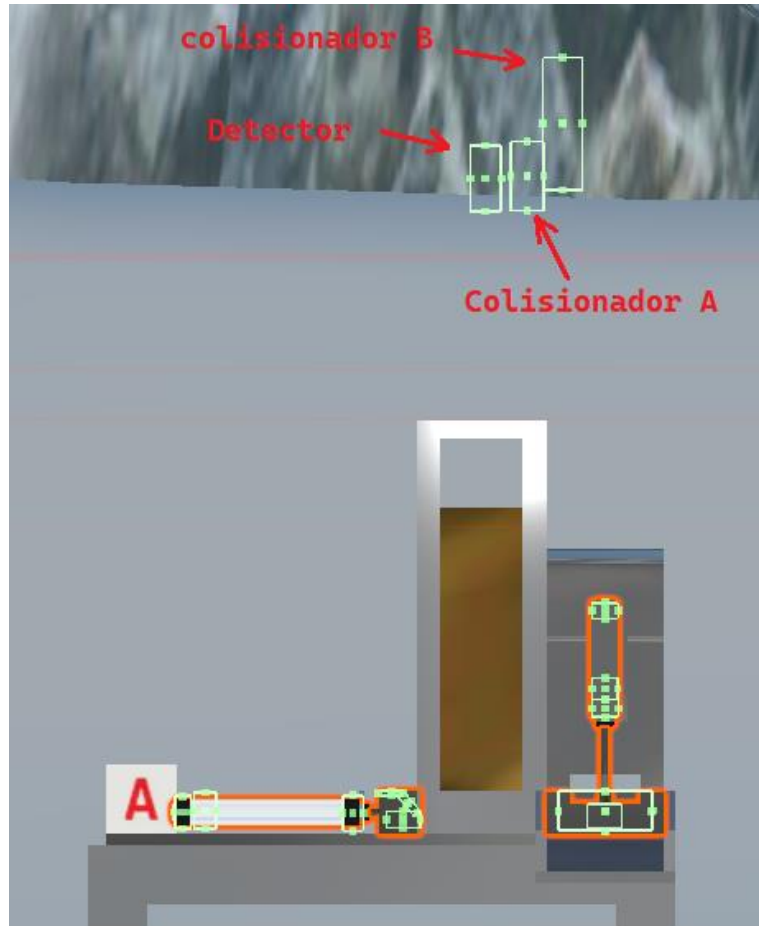


Figura 5.95. El colisionador B ha bajado lo que obstruye el movimiento del colisionador A, y se produce el efecto en el que el cilindro A no se expande porque hay una pieza de trabajo atorada entre los cabezales A y B.

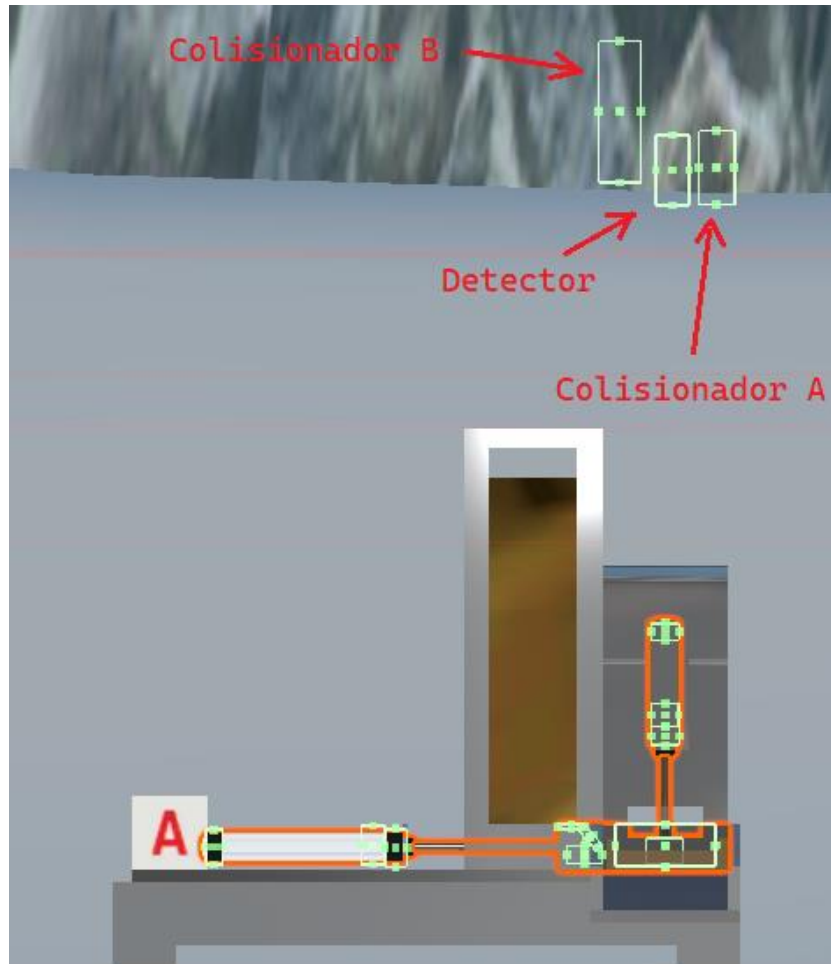


Figura 5.96. Primero se expandió el cilindro A y luego el B. Se observa que, si el cilindro A se contrae, ocurriría un choque entre los colisionadores A y B que impediría el retroceso completo del cilindro A.

Para controlar el activado del colisionador B, se creó el *script* mostrado en la figura 5.97. En esta figura, la línea 6 crea la clase *Freno*, donde el nombre indica que el colisionador sirve de freno del movimiento del colisionador A.

La línea 9 crea una variable pública de tipo *CreaPieza* que se utiliza para acceder a una variable declarada en el *script* *CreaPieza*.

La línea 12 declara una variable privada de tipo *Collider* utilizada para guardar el *Colisionador B* (vea las figuras 5.94 a 5.96).

Las líneas 14 a 17 forman parte de la función *Start*, en donde la línea 16:

```
colisionador = gameObject.GetComponent < Collider > ();
```

obtiene el componente *Collider* del *GameObject* al cual se asocia el *script* *Freno*, por lo que este *script* debe colocarse como componente del *GameObject* que contiene el componente *Collider* del *GameObject* *Colisionador B*.

Las líneas 19 a 30 forma parte de la función *Update*. La función tiene dentro dos instrucciones *if* las cuales determinan si se han creado o no las piezas de trabajo. La línea 21:

```
if(refCreaPieza.crearonPiezasInicio)
```

Indica que, si la variable *crearonPiezasInicio* declarada en el script *CreaPieza* es verdadera, entonces se ejecuta la sentencia declarada en la línea 23:

```
colisionador.enabled = true;
```

Esta última instrucción indica que el colisionador guardado en la variable *colisionador* debe activarse (el colisionador B se activa).

La instrucción *if* de la línea 26 es exactamente lo opuesto a lo declarado en la instrucción *if* de la línea 21 y podría ser simplemente sustituida por una instrucción *else*. En este caso si no se han creado las piezas de trabajo, entonces el colisionador no está activado como indica la línea 28:

```
colisionador.enabled = false;
```

```
1  using UdonSharp;
2  using UnityEngine;
3  using VRC.SDKBase;
4  using VRC.Udon;
5
6  public class FrenC : UdonSharpBehaviour
7  {
8      // variable para acceder a las variables del script CreaPiezas
9      public CreaPieza refCreaPieza;
10
11     // crea una variable para guardar el componente Collider del GameObject
12     private Collider colisionador;
13
14     void Start()
15     {
16         colisionador = gameObject.GetComponent<Collider>();
17     }
18
19     void Update()
20     {
21         if (refCreaPieza.crearonPiezasInicio)
22         {
23             colisionador.enabled = true;
24         }
25
26         if (!refCreaPieza.crearonPiezasInicio)
27         {
28             colisionador.enabled = false;
29         }
30     }
31 }
```

Figura 5.97. Código implementado para controlar la activación del colisionador B de las figuras 5.94 a 5.96.

La figura 5.98 muestra el código implementado para el control del detector mostrado en las figuras 5.94 a 5.96. En esta figura la línea 6 declara la clase *Detector* formada por las líneas 7 a 34.

La línea 9 declara una variable privada de tipo *float* llamada *timer* utilizada como temporizador.

La línea 12 declara una variable privada de tipo booleano llamada *activarCollider* utilizada para activar el colisionador B (el *script* no especifica que solo el colisionador B debe activarse, pero debido a la posición de estos colisionadores, el colisionador B es el único que se detecta).

La línea 15 crea una variable privada de tipo *Collider* llamada *detector* utilizada para guardar el colisionador que será detectado por el *Detector* (que será el colisionador B mostrado en las figuras 5.94 a 5.96).

La línea 17 declara una función *OnTriggerEnter*. Dentro de los parámetros de la función se indica que el colisionador que entre al detector se le asignará la variable *colisionador* (esto ya se ha visto anteriormente).

Dentro del cuerpo de la función *OnTriggerEnter* (líneas 18 a 23), la línea 19:

```
detector = colisionador;
```

asigna la variable *colisionador* a la variable *detector*. De este modo se guarda el colisionador detectado para su uso dentro de la función *Update* (se verá a continuación).

La línea 20:

```
colisionador.enabled = false;
```

desactiva el colisionador que toca el detector (que deberá ser el colisionador B de la figura 5.95).

La línea 21 establece el temporizador en 0 segundos justo en el momento en el que se detecta el colisionador B.

La línea 22 indica que la variable *activarCollider* se hace verdadera.

Las líneas 25 a 33 forman parte de la función *Update*. Dentro de esta función, la línea 27:

```
timer += Time.deltaTime;
```

establece el temporizador. El método *Time.deltaTime* se ha visto anteriormente.

Las líneas 28 a 32 (dentro de la función *Update*) forman una instrucción *if* que se ejecuta si la variable *activarCollider* es verdadera y además el temporizador (*timer*) es mayor a 1.5 segundos. Dentro de la instrucción *if* se vuelve a activar el colisionador B guardado en la variable *detector* (línea 30) y la variable *activaCollider* se hace falsa (línea 31) de este modo la instrucción *if* solo se ejecuta una ocasión.

Estos últimos dos *scripts* completan el código generado para el funcionamiento de la estampadora. En la siguiente sección se aborda el funcionamiento de los gemelos digitales con los códigos que se propusieron en esta sección.

```

1  using UdonSharp;
2  using UnityEngine;
3  using VRC.SDKBase;
4  using VRC.Udon;
5
6  public class Detector : UdonSharpBehaviour
7  {
8      // crea una variable de tipo float para propósitos de temporización
9      private float timer = 0;
10
11     //crea una variable booleana para indicar cuando activar el colisionador
12     private bool activarCollider = false;
13
14     // crea una variable para acceder al colisionador y volver a activarlo
15     private Collider detector;
16
17     void OnTriggerEnter(Collider colisionador)
18     {
19         detector = colisionador;
20         colisionador.enabled = false;
21         timer = 0;
22         activarCollider = true;
23     }
24
25     void Update()
26     {
27         timer += Time.deltaTime;
28         if (activarCollider && timer > 1.5f)
29         {
30             detector.enabled = true;
31             activarCollider = false;
32         }
33     }
34 }

```

Figura 5.98. Código implementado para el Detector de las figuras 5.94 a 5.96.

6. Resultados

Se realizaron un gran número de pruebas de funcionamiento durante la construcción del simulador virtual, la mayoría de las cuales fueron realizadas dentro de Unity. Sin embargo, las pruebas que aquí se presentan son las realizadas dentro de la aplicación de VRChat una vez que el mundo construido fue subido al metaverso y como tal se puede acceder desde cualquier dispositivo con acceso a la plataforma VRChat. Para acceder al mundo se puede buscar dentro de la ventana de configuración de la plataforma en la pestaña mundos (worlds), como se remarca en la figura 6.1. Dentro de la ventana de mundos se debe activar los *Community Labs* para poder visualizar el mundo de nombre: "LOR Mundo". Hay una gran cantidad de mundos publicados dentro de *Community Labs*; para poder encontrar fácilmente el mundo se puede buscar con el nombre desde la pestaña de búsqueda (pero para encontrarlo primero se deben activar los *Community Labs*). La figura 6.2 muestra el mundo de nombre "LOR Mundo" como se visualiza desde la ventana de búsqueda de VRChat.



Figura 6.1. Ventana de VRChat donde se accede a la ventana de configuración. Se remarca en rojo la pestaña worlds.

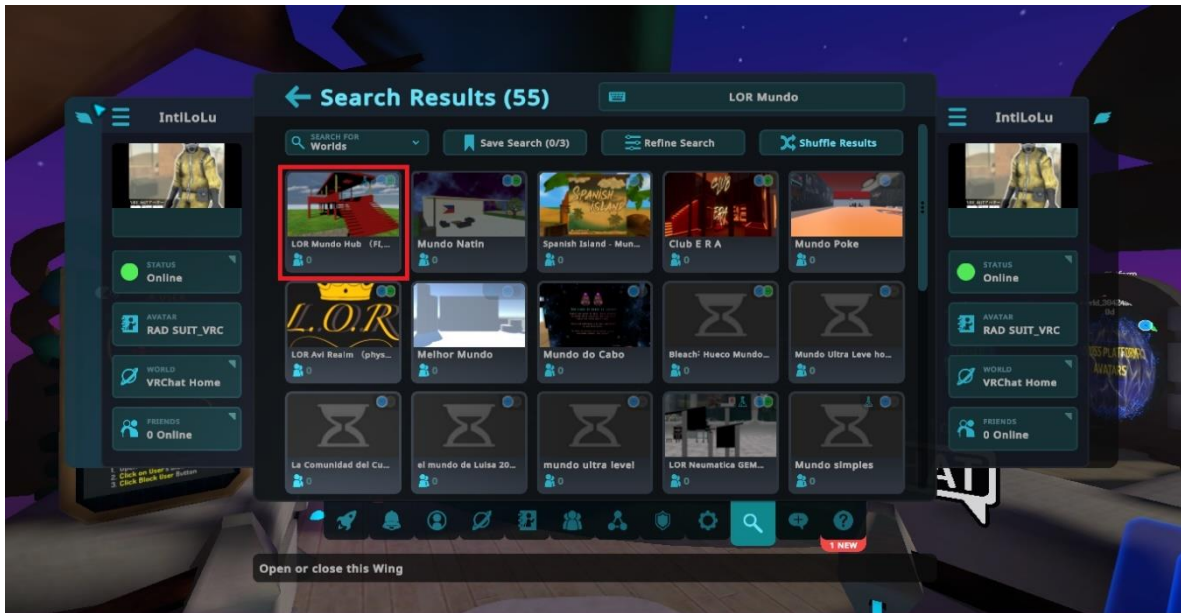


Figura 6.2. Ventana de búsqueda en VRChat donde se resalta el mundo de nombre “LOR Mundo”.

Este mundo (LOR Mundo) aloja varios mundos desarrollados por estudiantes y profesores de la facultad de ingeniería con una diversidad de temáticas. La figura 6.3 muestran algunos de los “portales”, encontrados en LOR Mundo, a los cuales se puede acceder. Uno de estos portales tiene el nombre LOR Neumática GemDig el cual es el mundo desarrollado en el presente trabajo. Para entrar al mundo se tiene que atravesar el portal con el avatar de los usuarios. La figura 6.4 muestra el mundo virtual LOR Neumática GemDig.

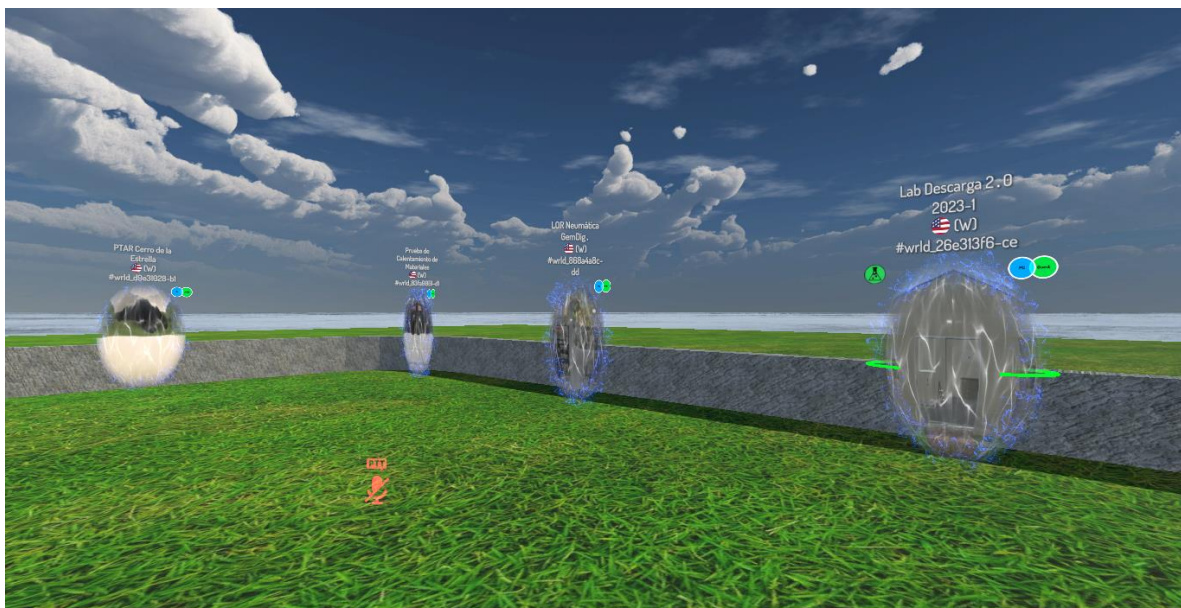


Figura 6.3. Dentro del mundo “LOR Mundo” se puede acceder a otros mundos desarrollados en la facultad de ingeniería a través de los portales.

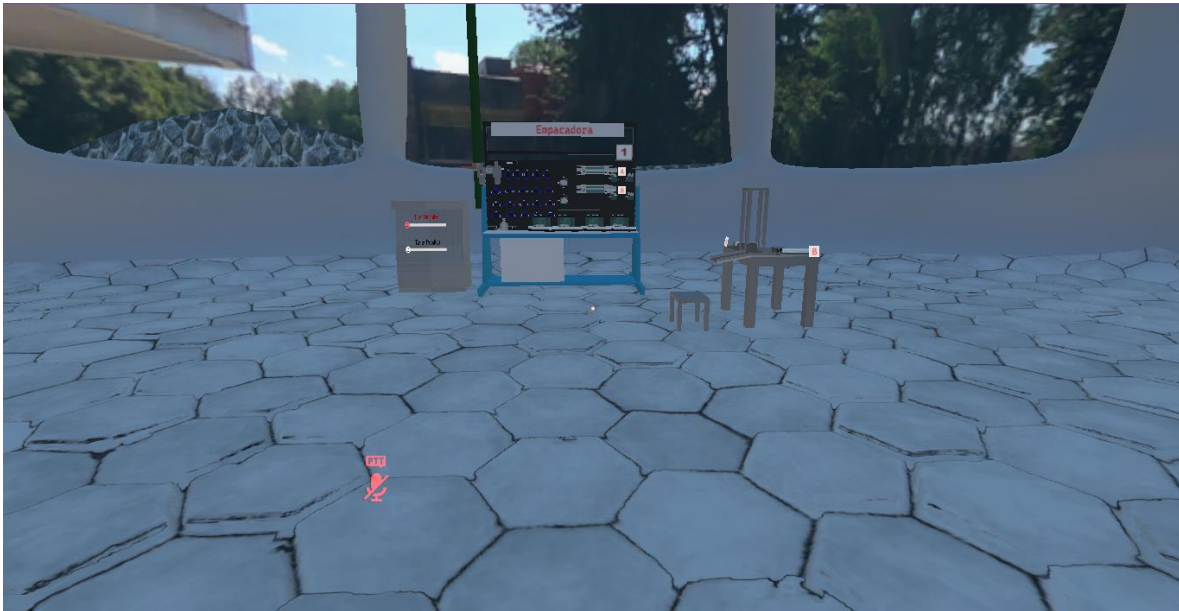


Figura 6.4. Imagen tomada desde VRChat para el mundo virtual LOR Neumática GemDig.

6.1. Pruebas de funcionamiento de la empacadora

Para las pruebas de la empacadora se construyeron los circuitos neumáticos que se mostraron en las figuras 3.3 y 3.4. También se simulan algunas situaciones que pudieran llegar a ocurrir durante la ejecución de la simulación.

Para el circuito neumático mostrado en la figura 3.3, el cual hace uso de una memoria neumática y un temporizador negativo para lograr el funcionamiento requerido, se construyó el circuito neumático en la mesa de trabajo virtual correspondiente al gemelo digital de la empacadora, el cual es mostrado en la figura 6.5. Las líneas de color azul claro representan las conexiones por medio de las mangueras. Al alimentar el circuito (abriendo la llave de paso y la válvula de la unidad de mantenimiento) las mangueras que tienen presión de aire cambian de color a un azul oscuro (fig. 6.6). Como se recordará de la programación de la empacadora, los paquetes se crearán en el momento en que se activen la válvula de la unidad de mantenimiento y el botón enclavado. Los paquetes no desaparecerán de escena hasta que se desactiven ambos botones.

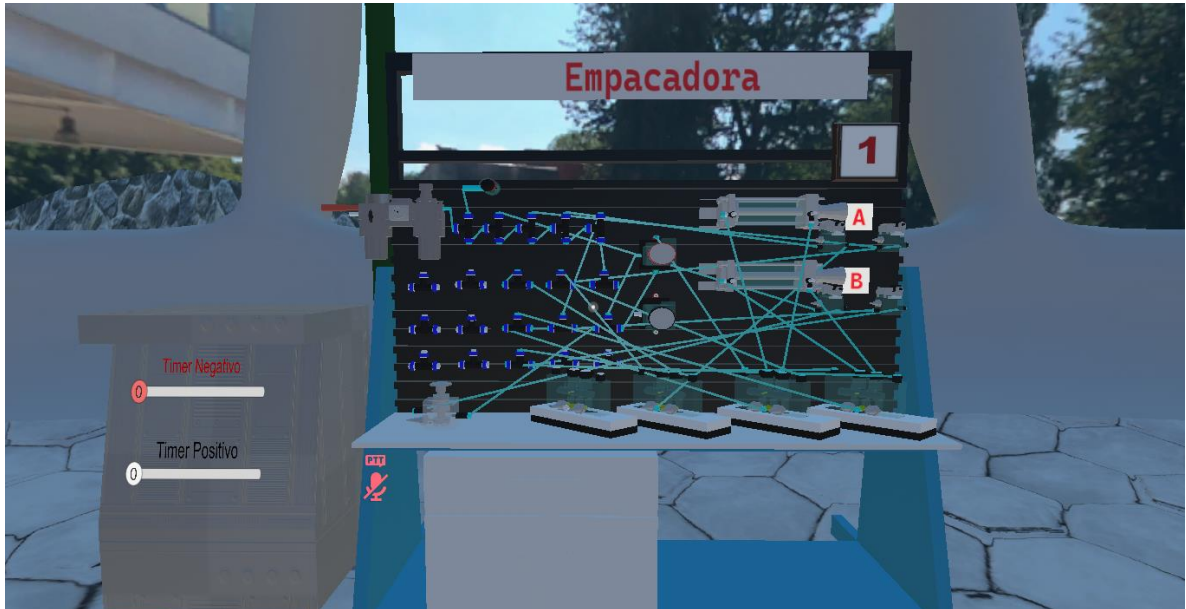


Figura 6.5. Construcción del circuito neumático para la empacadora mostrado en la figura 3.3, el cual hace uso de una memoria neumática y un temporizador negativo. Las líneas azul claro representan las conexiones mediante mangueras.

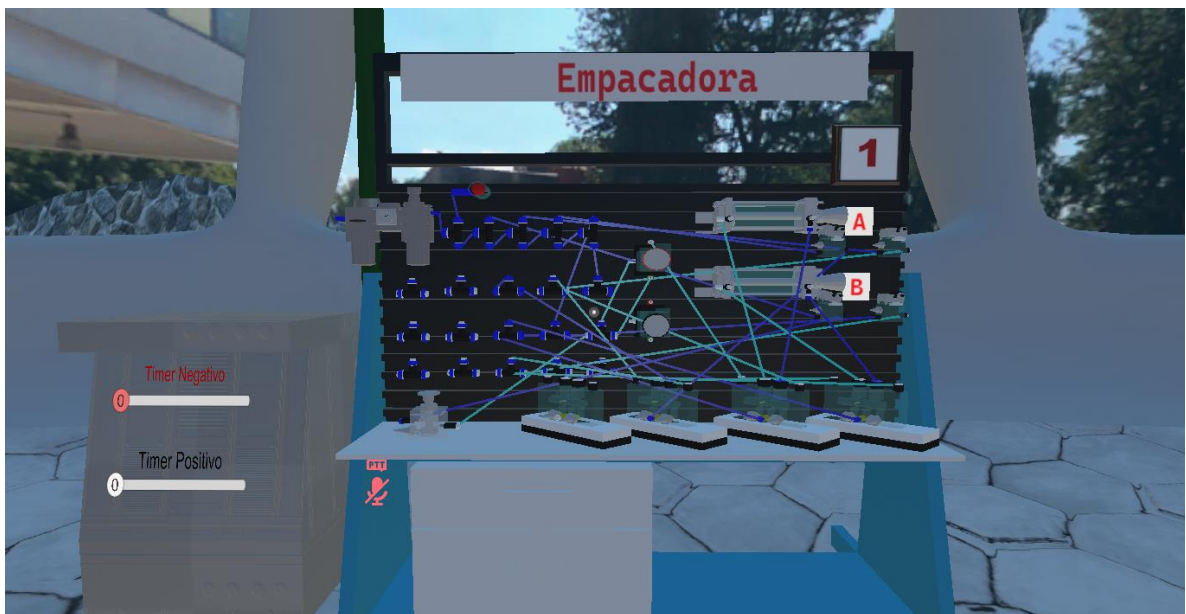


Figura 6.6. Construcción del circuito neumático para la empacadora mostrado en la figura 3.3, una vez que se ha alimentado el circuito. Las líneas azul oscuro representan mangueras con presión de aire comprimido. Las líneas azul claro no tienen presión de aire comprimido.

La imagen 6.7 muestra la creación en escena de los paquetes y de la caja contenedora. Si el circuito esta armado correctamente, justo en el momento en el que inicia el ciclo de movimiento con el cilindro A, se crean los paquetes desde la posición inferior del soporte de almacenamiento de paquetes. La creación de los paquetes es más rápida que el movimiento del vástago por lo que antes de empujarse el primer paquete los siete restantes situados por encima ya han sido creados. Cuando un paquete es empujado por el cabezal A, se crea un nuevo paquete en la parte superior del elemento almacenador de paquetes. La figura 6.8 muestra a los vástagos "A" de la mesa de trabajo y del gemelo digital expandidos (movimiento A+). Se observa, además que el paquete ha sido empujado fuera del soporte contenedor.

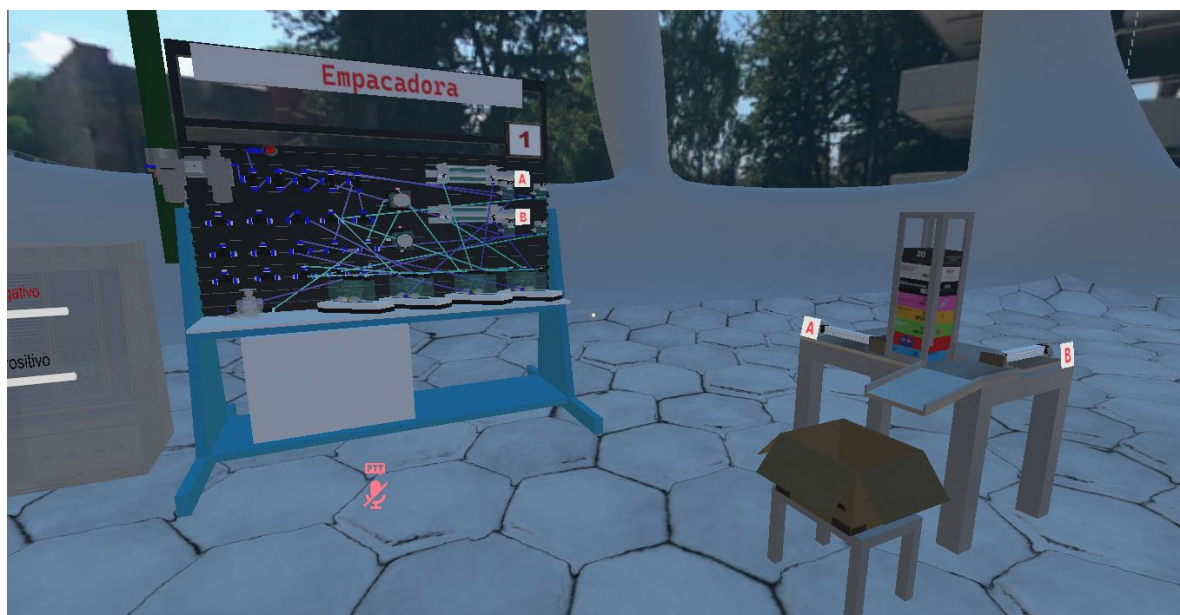


Figura 6.7. Cuando se activa la unidad de mantenimiento y el botón enclavado se crean los paquetes y la caja contenedora para la operación de la empacadora.



Figura 6.8. El paquete se empuja desde el soporte contenedor con la expansión del cilindro A. Se observa que los cilindros A de la mesa de trabajo y del gemelo digital se han expandido. También se observa que se creó un paquete nuevo en la parte superior del soporte contenedor de paquetes.

El movimiento A- (según plantea del diagrama de movimiento) se observa en la figura 6.9. Los paquetes dentro del soporte contenedor caen por gravedad y el inferior está listo para ser empujado de nueva cuenta por el cilindro A.

La figura 6.10 muestra el movimiento B+ del cilindro B de la empacadora, donde puede apreciarse que el paquete ha sido empujado a la caja contenedora.

Por último, la figura 6.11 muestra el movimiento B- del cilindro B con lo que se completa el ciclo de trabajo de la empacadora y se puede volver a reiniciar el ciclo. Si el botón se mantiene enclavado el ciclo continuará indefinidamente.

La figura 6.12 muestra que cuando tres paquetes han caído a la caja contenedora de paquetes esta se cierra. Posteriormente desaparecerá de la escena y se crea una nueva caja vacía y abierta para volver a ser llenada con paquetes. El tiempo de cerrado de la caja, destrucción y creación de la nueva caja es suficiente para poder apreciar el proceso y también para que las piezas posteriores caigan a la nueva caja.



Figura 6.9. El cilindro A regresa a su posición original (movimiento A-). Los paquetes restantes en el soporte almacenador caen por gravedad.

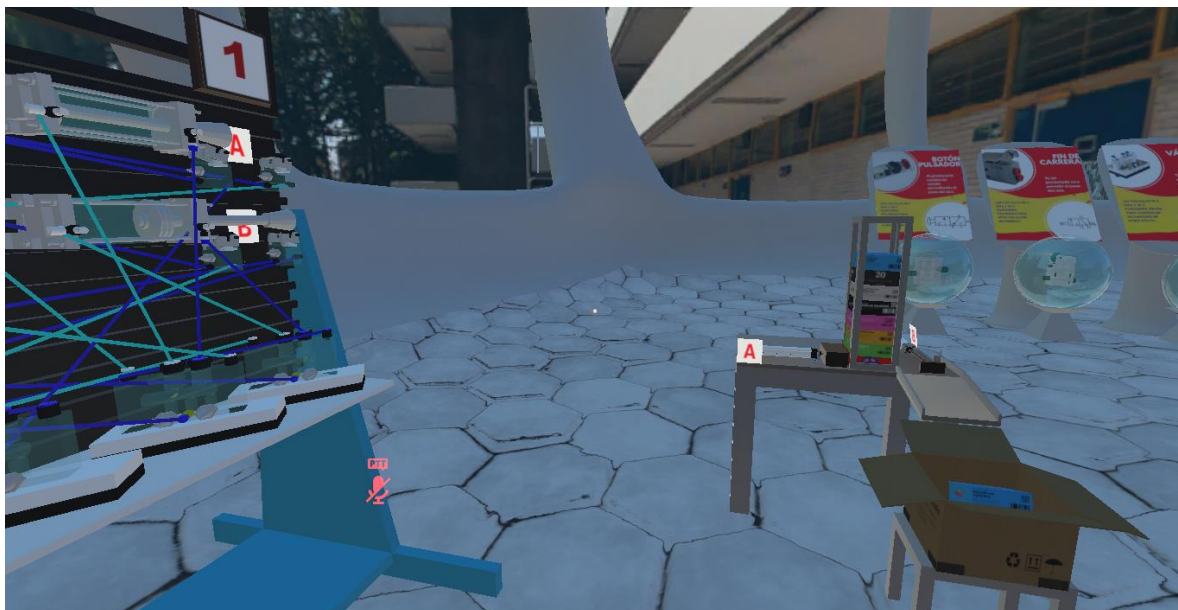


Figura 6.10. Movimiento B+ del cilindro B de la empacadora. Se observa el paquete azul cayendo a la caja contenedora.

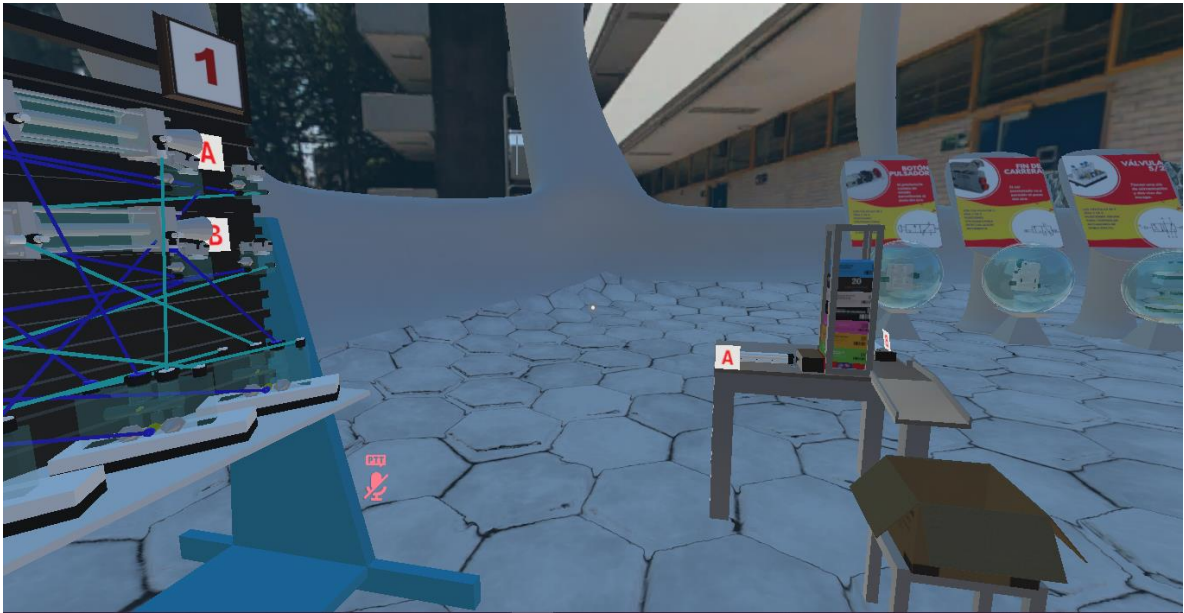


Figura 6.11. Movimiento B- del cilindro B de la empacadora que completa el ciclo de movimiento.



Figura 6.12. En el momento que caen tres paquetes en la caja contenedora esta se cierra y posteriormente aparece una nueva caja contenedora abierta.

Debido al tamaño de las imágenes y al desorden de las conexiones es difícil apreciar si el circuito es construido por método cascada o con memorias, por lo que se omite el circuito construido por el método cascada. Basta decir que el método cascada funciona de manera correcta y se obtienen resultados similares a los ya mostrados.

El usuario también podría establecer ciertas secuencias de movimiento incorrectas que no cumplirían el objetivo de posicionar paquetes dentro de la caja contenedora; por ejemplo, podría establecer la secuencia: B+A+B-A-. Este caso es mostrado en la figura 6.13 donde puede observarse que los paquetes han quedado atorados. Los vástagos de la empacadora permanecerán en la posición mostrada hasta que se borren los paquetes lo cual puede hacerse desactivando la unidad de mantenimiento y el botón enclavado. Los vástagos de la mesa de trabajo si se moverán con la secuencia implementada.

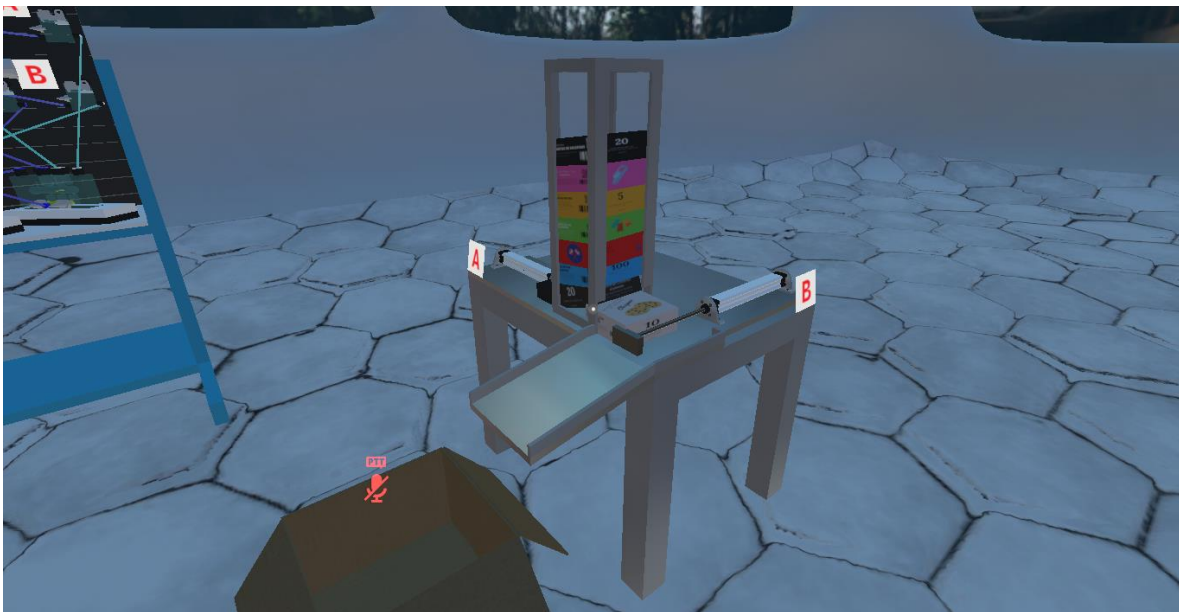


Figura 6.13. Secuencia de movimiento que da por resultado que las piezas se atoren.

6.2. Pruebas de funcionamiento de la indentadora

Las pruebas de funcionamiento de la indentadora consistieron en la elaboración de los circuitos neumáticos mostrados en las figuras 3.7 y 3.8, correspondientes a los circuitos elaborados con los métodos intuitivo y el método cascada respectivamente. La figura 6.14 muestra el circuito neumático de la figura 3.8 implementado en la mesa de trabajo virtual asociada al gemelo digital de la indentadora. La secuencia de movimiento realizada es A+B+B-A-. A diferencia de la empacadora donde las piezas de trabajo (paquetes) se crean si están activados simultáneamente el botón enclavado y la válvula de la unidad de mantenimiento, la pieza de trabajo de la indentadora (cubo) se crea con la extensión del vástago del cilindro A. La figura 6.15 muestra el instante en el que se activa la expansión del pistón del cilindro A de la indentadora y se crea el cubo en su posición inicial. La creación del cubo es suficientemente rápida para permitir que este se cree antes de ser empujado por el cabezal del vástago.

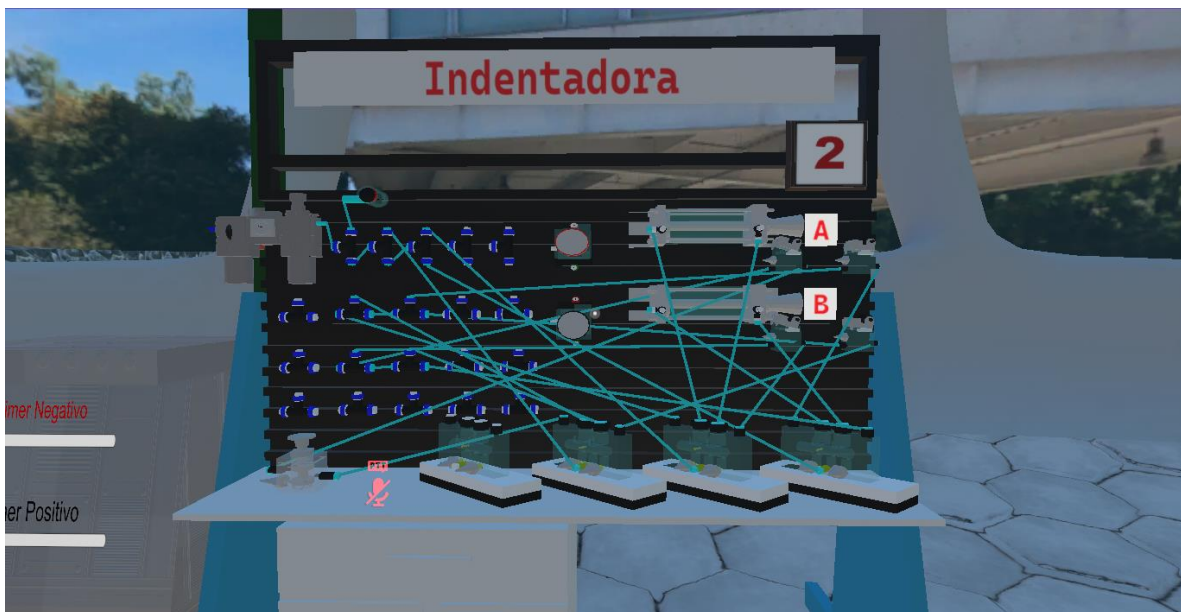


Figura 6.14. Circuito neumático utilizando el método cascada para la secuencia de movimientos de la indentadora.

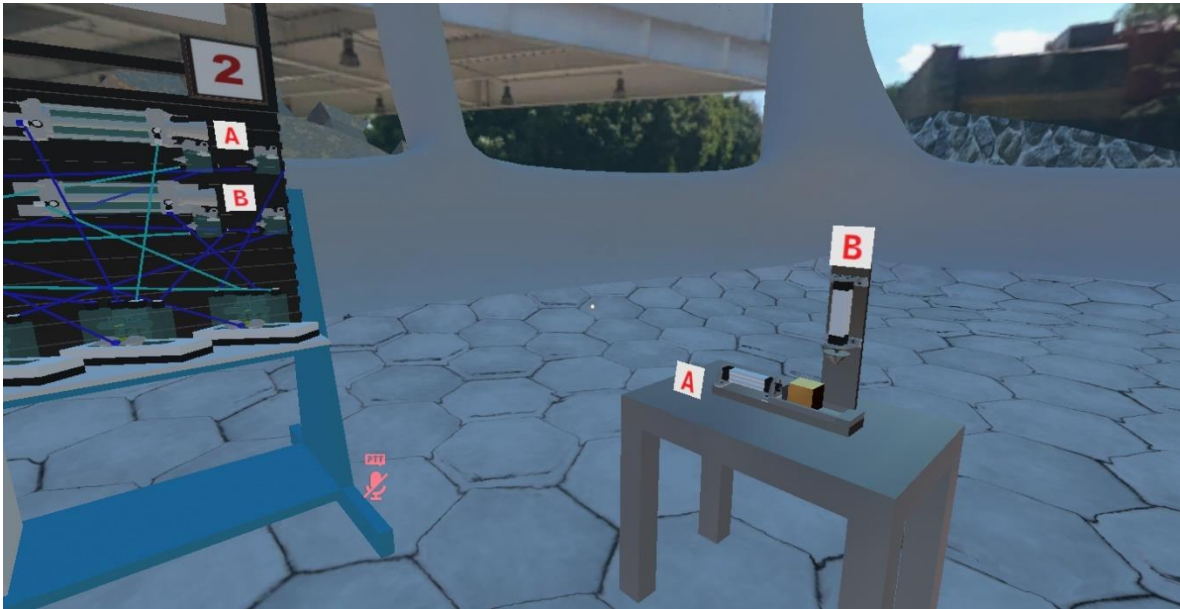


Figura 6.15. Se crea el cubo en el instante en el que el pistón del cilindro A de la indentadora se extiende.

La figura 6.16 muestra el cubo una vez que el pistón del cilindro A de la indentadora se ha extendido por completo y la pieza de trabajo se encuentra en la posición donde se realizará la indentación.

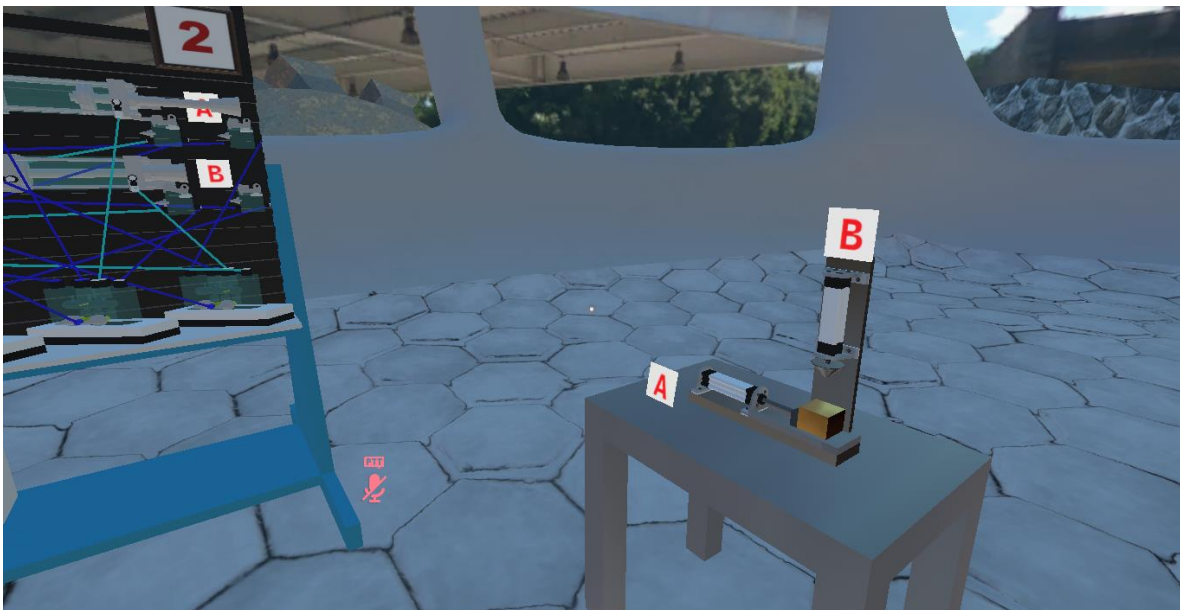


Figura 6.16. Movimiento A+ de los cilindros neumáticos de la mesa de trabajo y del gemelo digital.

El movimiento B+ cuando se realiza la indentación de la pieza de trabajo se observa en la figura 6.17. Se puede observar que el indentador perfora la pieza. En el momento en el que indentador hace “contacto” con la pieza de trabajo la estampa (Decal) del agujero (ver sección 4 de la programación de la indentadora) se coloca en la parte superior del cubo.

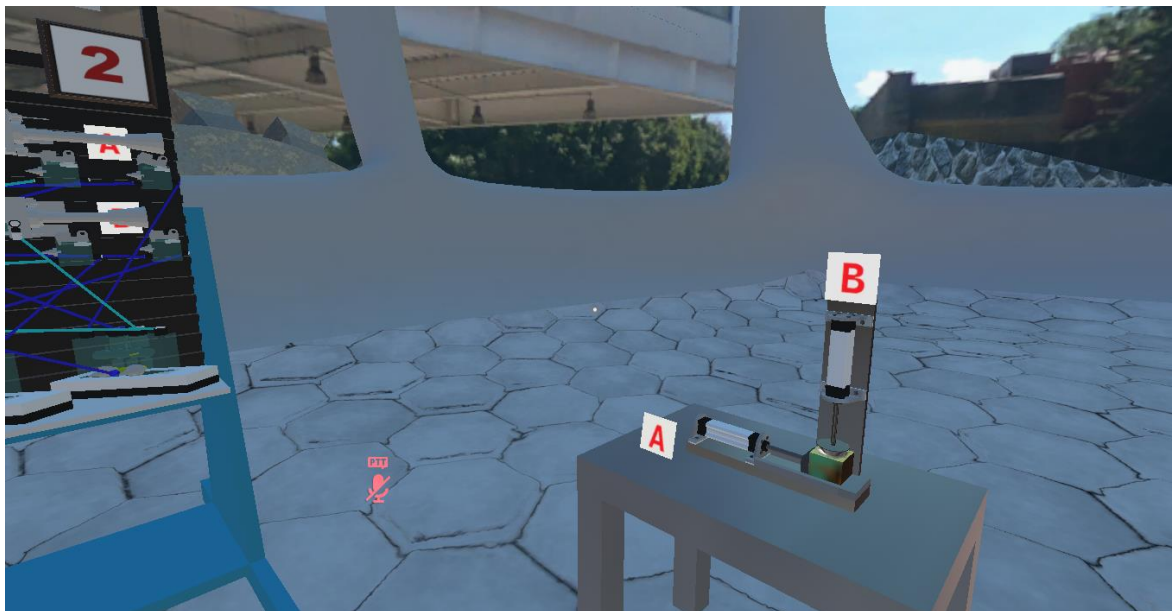


Figura 6.17. Movimiento B+ del pistón del cilindro B. Se observa que el cabezal del vástago realiza la indentación de la pieza de trabajo.

En la figura 6.18 el decal ya ha sido colocado por lo que al ocurrir el movimiento de retorno del pistón B (movimiento B-) se puede observar la “huella” del indentador.

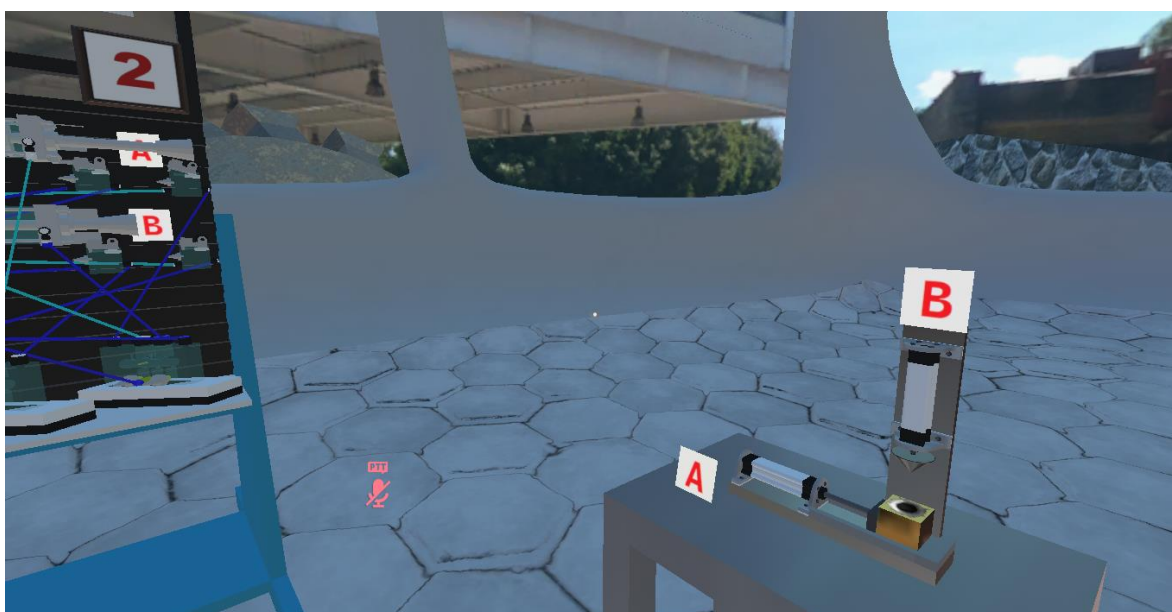


Figura 6.18. Movimiento B- donde el pistón del vástago B ha retrocedido. Se observa que la pieza tiene una marca de indentación.

El último movimiento para completar el ciclo es A-, el cual se observa en la figura 6.19. La pieza de trabajo, a la cual se realizó la indentación, permanecerá en escena solo un instante después de que el cilindro A se ha retraído, se destruirá e inmediatamente (si el botón sigue enclavado), se creará una nueva pieza de trabajo para comenzar de nuevo el ciclo de movimiento.

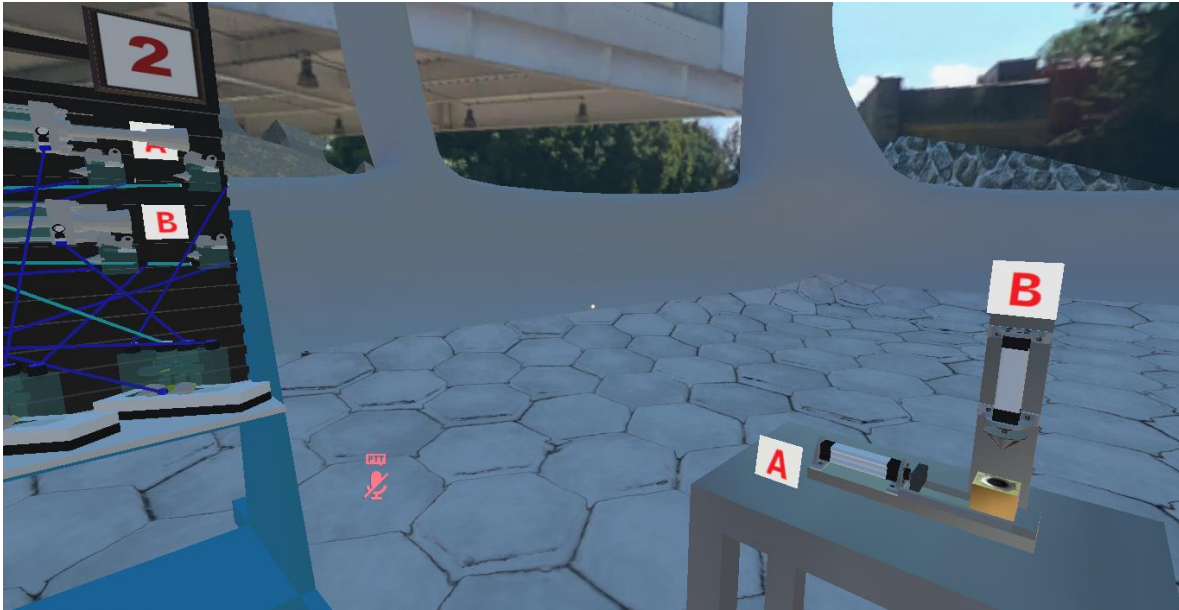


Figura 6.19. Movimiento A- del pistón del cilindro A de la indentadora.

Una secuencia de movimiento errónea que el usuario podría implementar para el gemelo digital de la indentadora podría ser: B+A+B-A-. En este ciclo de movimiento la indentación de la pieza de trabajo no ocurrirá debido a que primero se expande el pistón del cilindro B y, por lo tanto, al expandir el pistón del cilindro A, la pieza de trabajo se atorará con la parte lateral del indentador como se observa en la figura 6.20. En este caso, debido a la posición del colisionador que se colocó en el cabezal del cilindro B, la pieza no hace contacto con el cabezal por lo que la simulación puede no verse “realista”; sin embargo, se cumple el objetivo de impedir que la pieza se posicione en un lugar que no debería debido a la secuencia de movimiento incorrecta. La pieza de trabajo será destruida cuando ambos cilindros se contraigan y se volverá a atorar en la misma posición si el circuito sigue alimentado y el botón enclavado pulsado. Otras secuencias de movimiento, como: B+B-A+A-, colocarán la pieza de trabajo en la posición adecuada, pero no se logrará la indentación y al final la pieza será destruida cuando los cilindros regresen a sus posiciones retraídas.

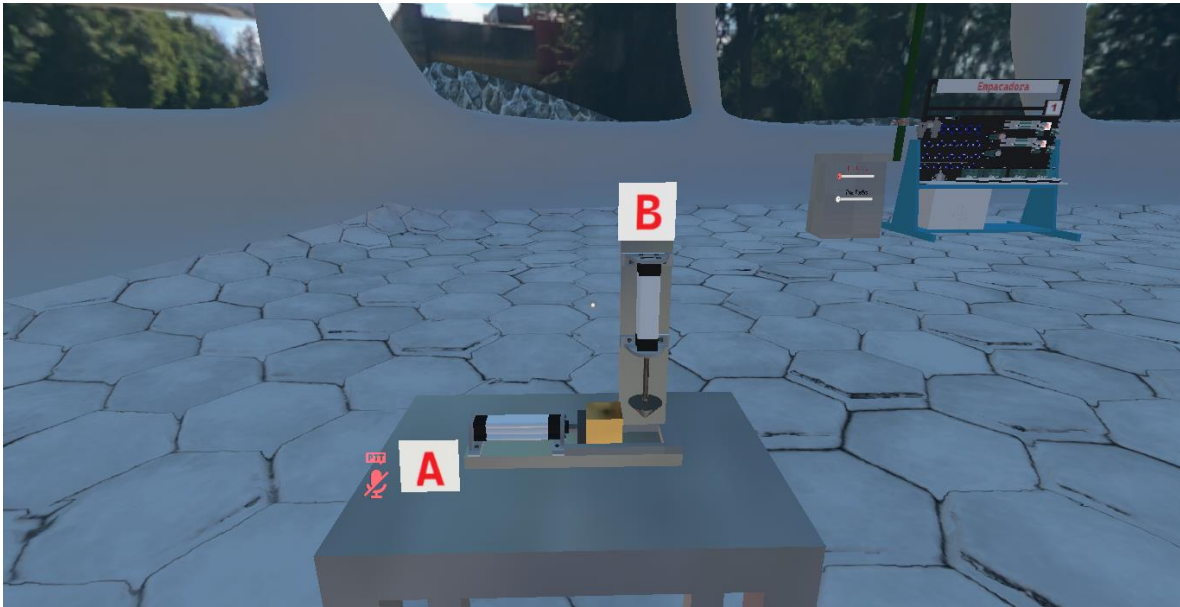


Figura 6.20. Secuencia de movimiento errónea donde el pistón del cilindro B se expande primero que el pistón del cilindro A y la pieza se atora entre los cabezales de los vástagos.

6.3. Pruebas de funcionamiento de la estampadora

Para las pruebas de la estampadora se construyó el circuito neumático mostrado en la figura 3.11, en la mesa de trabajo asociada al gemelo digital de la estampadora, el cual se muestra en la figura 6.21. Las piezas de trabajo y la caja contenedora de las piezas se crean cuando se presionan la válvula de la unidad de mantenimiento y el botón enclavado como se muestra en la figura 6.22.



Figura 6.21. Circuito neumático utilizado para el ciclo de trabajo de la estampadora. El circuito ya ha sido alimentado.

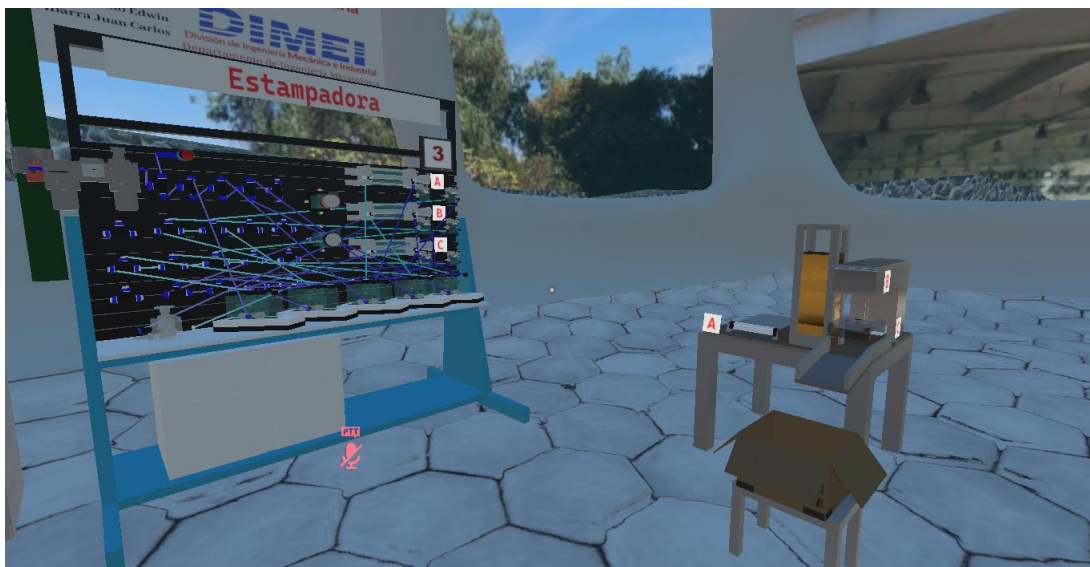


Figura 6.22. Al inicio del ciclo de movimiento se crean las 8 piezas de trabajo en el almacén y una caja de almacenamiento de piezas.

La figura 6.23 muestra el primer movimiento del ciclo de trabajo (A+), donde la pieza de trabajo se posiciona en el lugar donde ocurrirá el estampado.

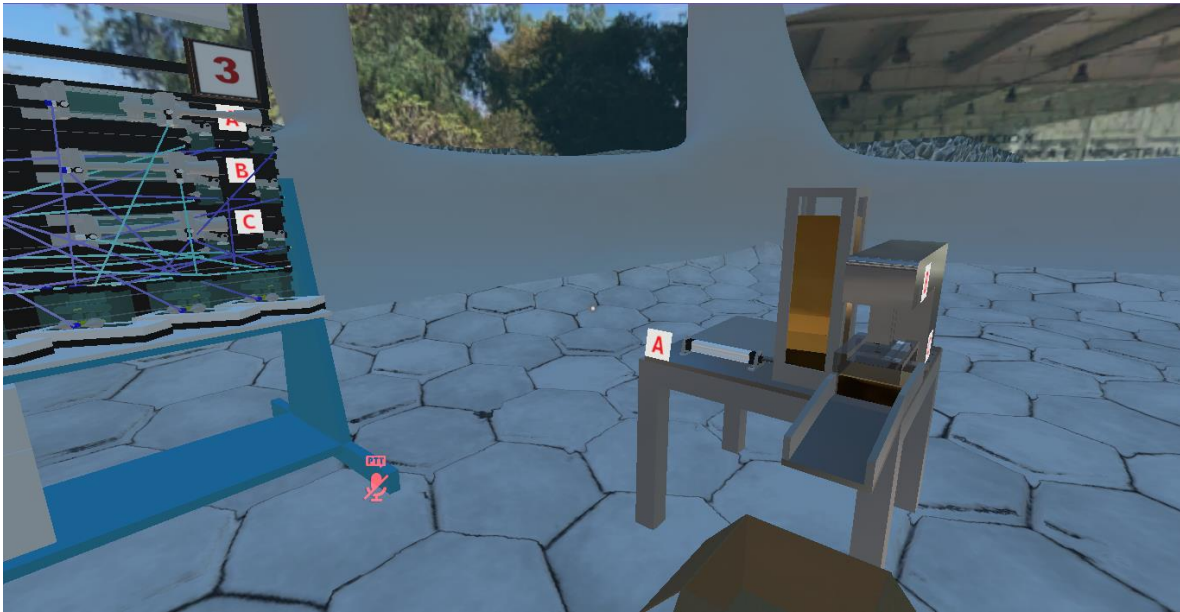


Figura 6.23. La pieza de trabajo se posiciona en el sitio donde será estampada. Movimiento A+.

La figura 6.24 muestra el momento en el que ocurre el movimiento B+, donde la pieza de trabajo es estampada. En esta figura se puede observar la transparencia del cabezal del cilindro B y la pieza “deformada”. Al regresar el vástago B a su posición retraída la pieza de trabajo ya está deformada y lista para empacarse (fig. 6.25).

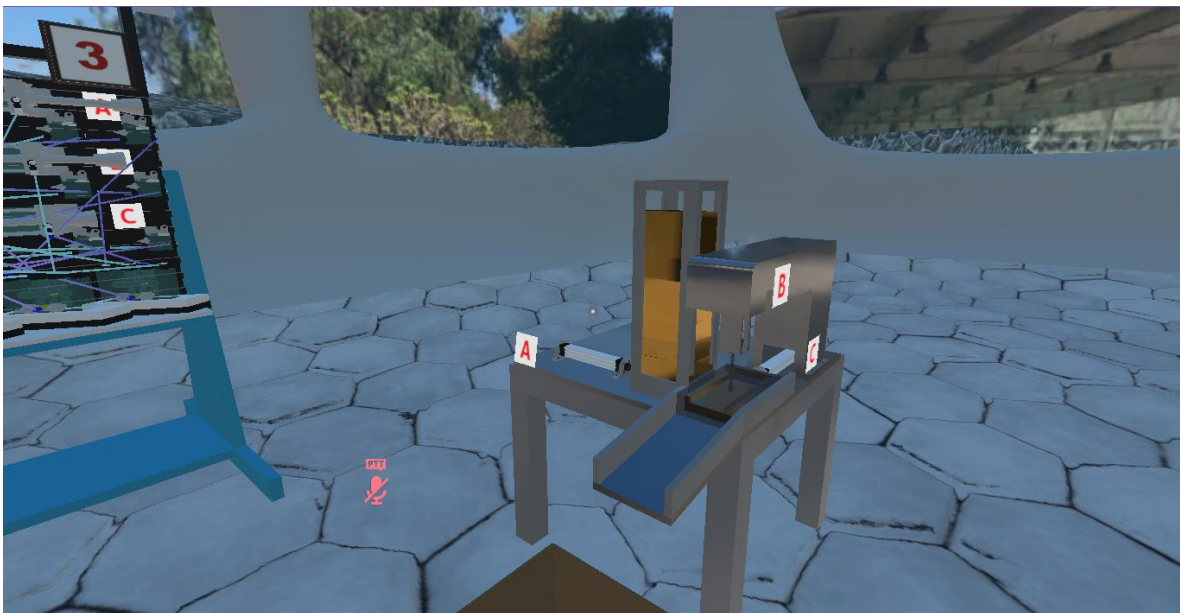


Figura 6.24. Movimiento B+ correspondiente al estampado de la pieza de trabajo.

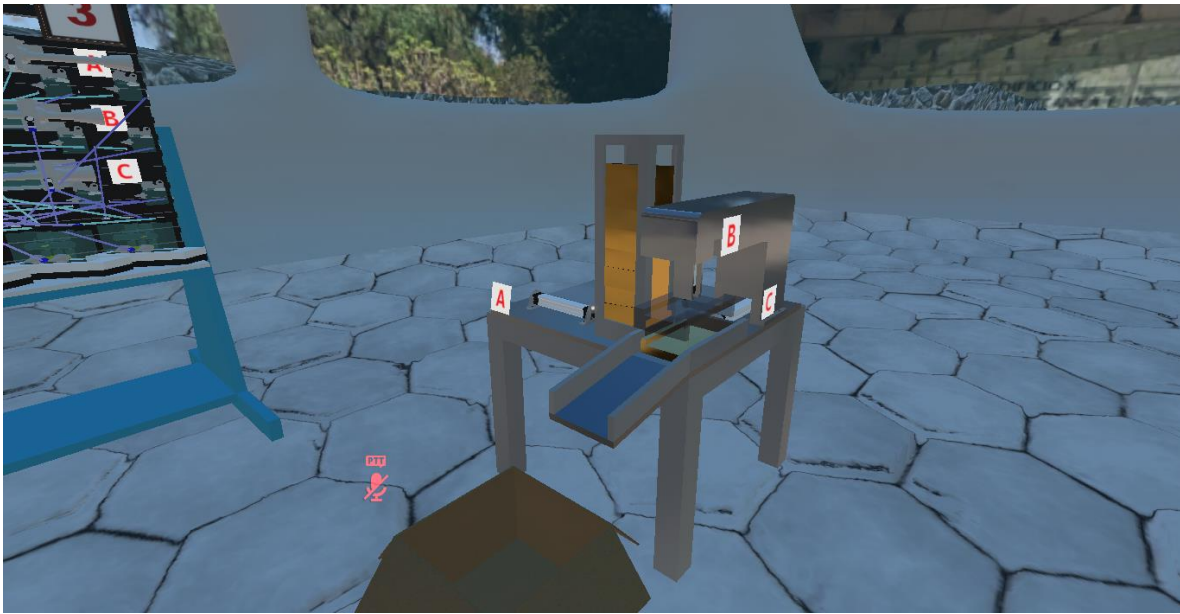


Figura 6.25. Movimiento B-, donde ya se realizó el estampado de la pieza.

En la figura 6.26 la pieza de trabajo es empujada hacia la caja contenedora. Las figura 6.27 y 6.28 completan el ciclo de movimiento retrayendo los cilindros C y A (movimiento C- y A- respectivamente).

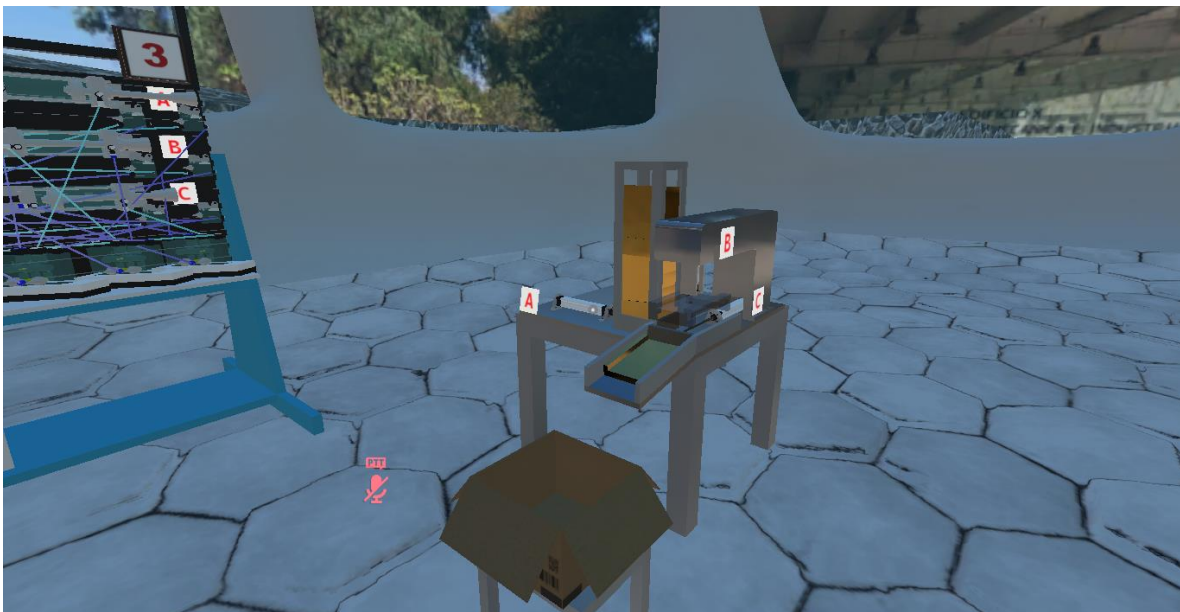


Figura 6.26. Movimiento C+ donde la pieza estampada es empujada hacia la caja contenedora.

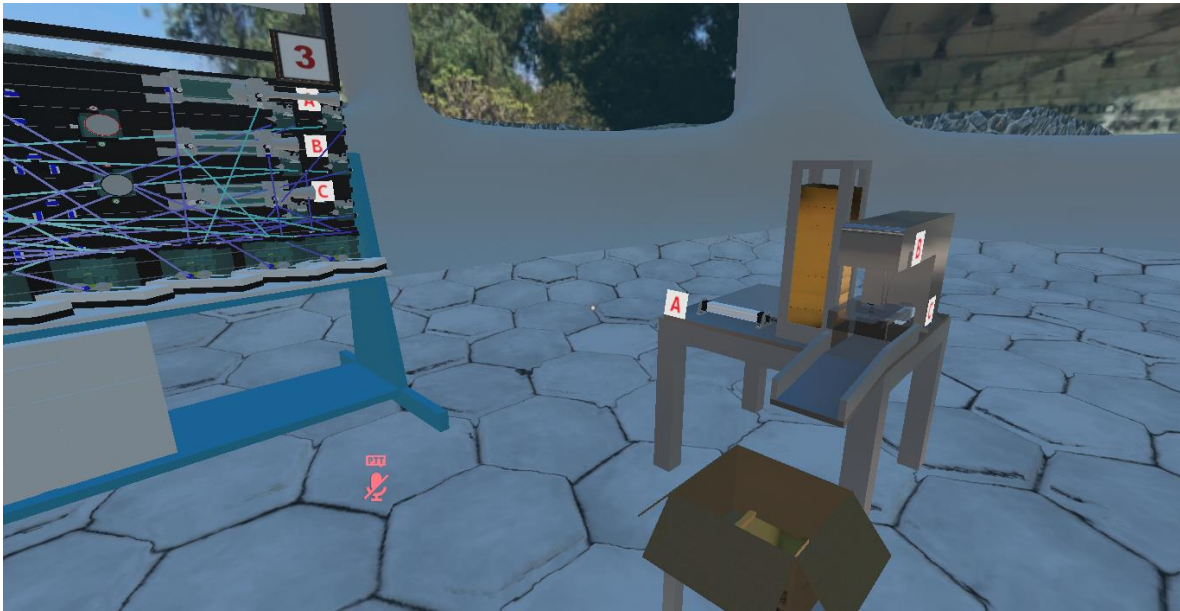


Figura 6.27. Movimiento C- La pieza de trabajo ya estampada es puesta en la caja contenedora.

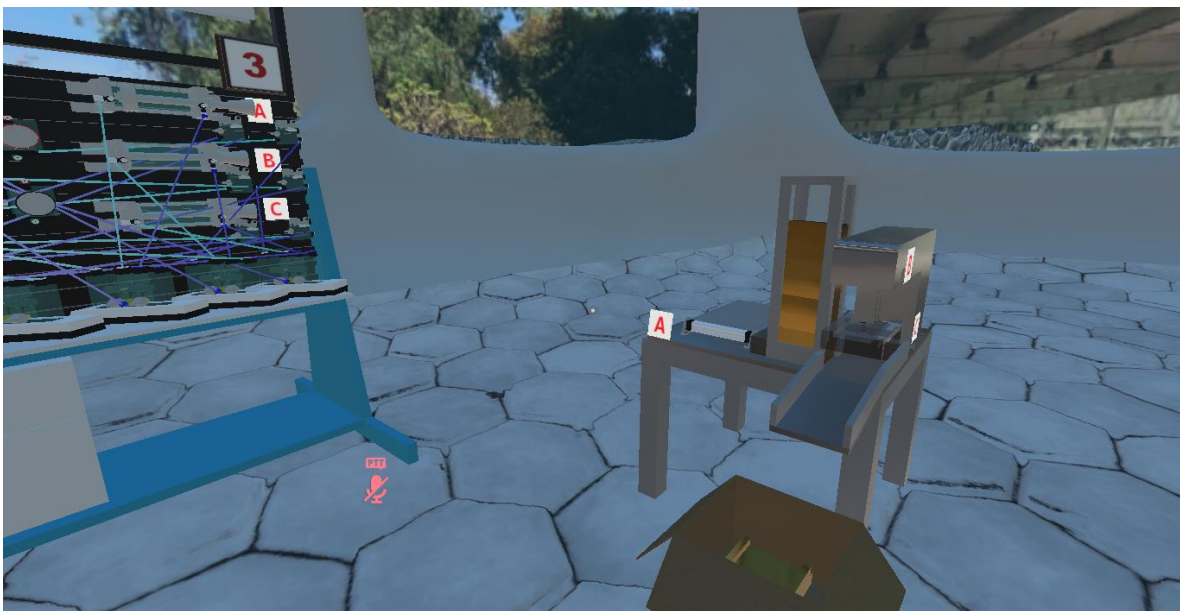


Figura 6.28. Movimiento A- donde se completa el ciclo de movimiento de la estampadora.

La caja contenedora de piezas estampadas también se cerrará, al igual que la caja contenedora de paquetes en la empacadora, cuando se llene con tres piezas estampadas. La figura 6.29. muestra el momento en el que la caja se llenó con tres piezas estampadas y se está cerrando.



Figura 6.29. Momento en el que la caja contenedora se llena con tres piezas estampadas y se está cerrando.

Además de la secuencia de movimiento que se espera que se construya, se probaron algunos movimientos erróneos que el usuario podría implementar. No se construyeron circuitos completos para hacer estas pruebas solo se movieron individualmente los vástagos para simular la situación. En la figura 6.30 se simula el caso donde el usuario primero hace el movimiento B+ y luego el A+. Como se puede observar la pieza de trabajo no atraviesa el cabezal del vástago B, por lo que el vástago del cilindro A no se expande por completo (se atora). En la misma figura se puede observar que los cilindros de la mesa de trabajo están extendidos.

Si el usuario primero acciona el movimiento C+ y luego el B+ se tiene la situación presentada en la figura 6.31, en donde el vástago B no baja hasta su posición final debido a que el cabezal del vástago C impide que se siga moviendo.

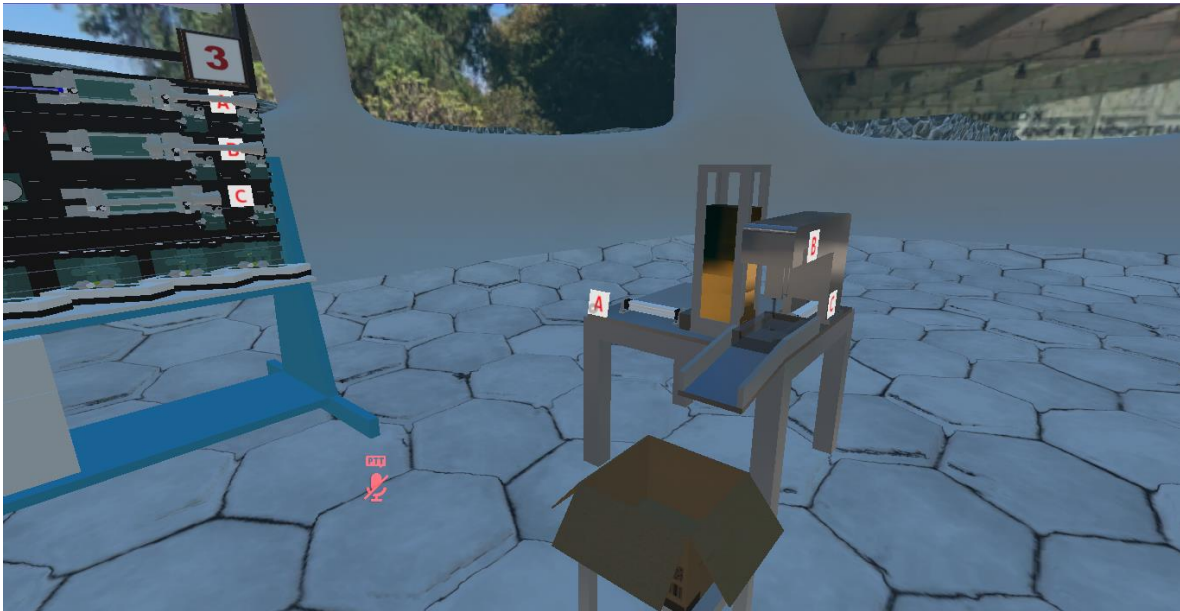


Figura 6.30. Secuencia incorrecta en la que el usuario primero hace el movimiento B+ y luego el A+. Se observa que la pieza no atraviesa el cabezal del vástago B.

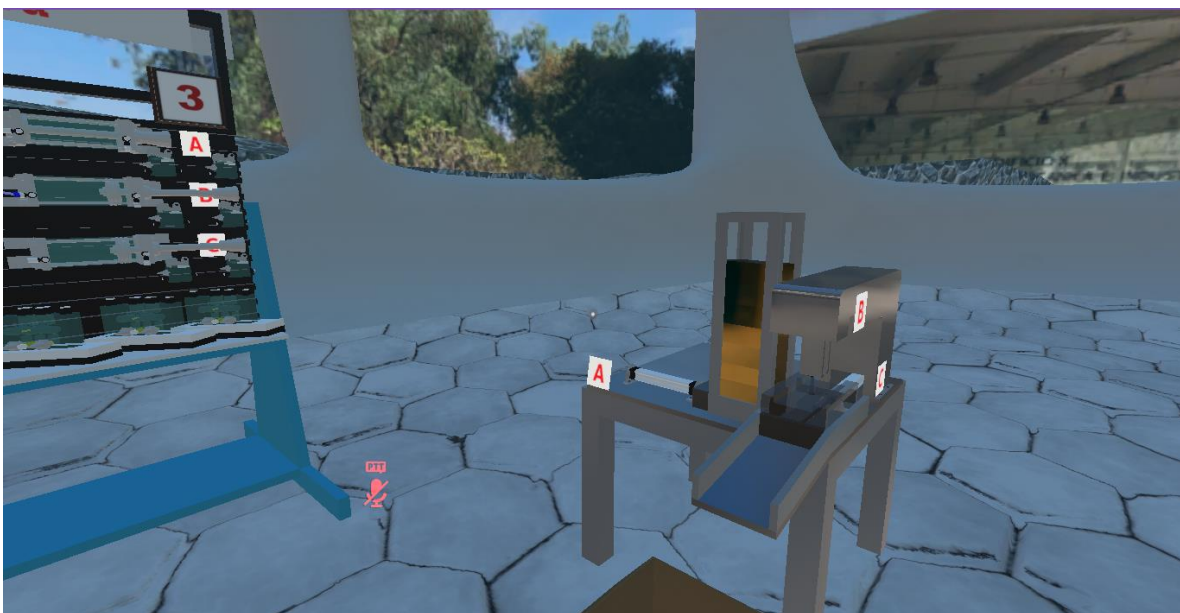


Figura 6.31. Secuencia de trabajo incorrecta donde primero se activó el movimiento C+ y luego el B+. El cabezal del vástago C impide el movimiento del cabezal B.

En la figura 6.32 se muestra la implementación de una secuencia incorrecta: C+A+B+C-A-B-. Esta secuencia de movimiento es simple y no involucra el uso de ninguna memoria neumática. Se puede observar que hay un paquete atorado debido a que el ciclo primero expande el vástago del cilindro C (C+) y luego se empuja la pieza con la expansión del vástago del cilindro A (A+). La pieza permanecerá atorada hasta que desaparezcan las piezas de la escena. Esto se consigue desactivando la unidad de mantenimiento y el botón enclavado en la mesa de trabajo asociada a la estampadora.

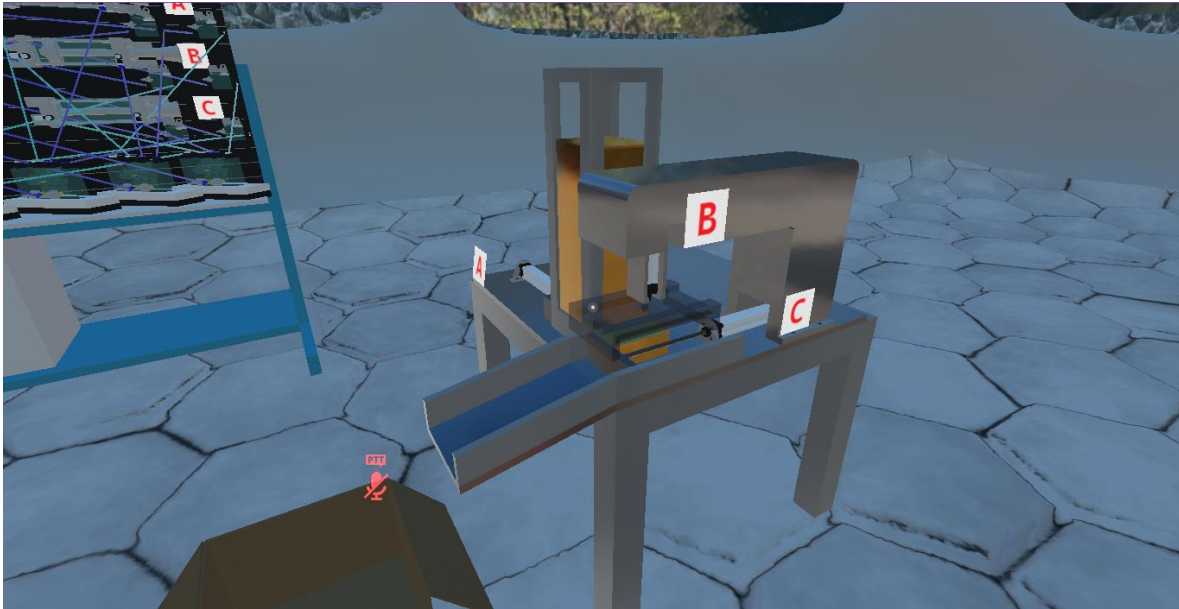


Figura 6.32. Secuencia de movimiento: C+A+B+C-A-B-, que provoca que la pieza de trabajo se atore.

7. Conclusiones

La integración de los gemelos digitales dentro del simulador NERV es una herramienta para que los estudiantes obtengan experiencia en la interpretación de los planos de situación neumáticos de una manera totalmente interactiva. Al ser una herramienta virtual, los estudiantes pueden experimentar situaciones reales sin temor a equivocarse, lo que complementa su formación académica al ayudarlos a descubrir aspectos que con los diagramas 2D pudieran pasar desapercibidos.

El uso de las mesas de trabajo neumáticas virtuales ayuda a los estudiantes a desarrollar habilidades en la construcción de circuitos neumáticos. El uso constante ayuda a familiarizarse con cada uno de los componentes disponibles en la mesa y capacita de manera adecuada para la posterior implementación de los circuitos reales.

Los gemelos digitales implementados dentro del simulador NERV funcionan de manera adecuada cuando se implementan los circuitos neumáticos requeridos para su funcionamiento. También se probaron algunas secuencias de movimiento erróneas para verificar el funcionamiento de los gemelos digitales en estos casos. Los resultados obtenidos fueron satisfactorios ya que los gemelos digitales se desempeñan como se piensa deberían funcionar en la realidad. Aún falta realizar pruebas de funcionamiento que involucren sobre todo circuitos erróneos para poder corregir desperfectos no detectados, ya que las pruebas fueron en su mayoría con los circuitos neumáticos que hacían funcionar de manera adecuada los modelos.

Se agregaron los elementos requeridos para lograr el funcionamiento de los gemelos digitales, pero las mesas de trabajo se ven llenas de algunos de los elementos; e incluso, como en el caso de los conectores en T, no se utilizan todos los elementos en la construcción de los circuitos. Se requiere que los elementos se vayan creando conforme el usuario los requiera, por ejemplo, que estuvieran disponibles en un modelo de maletín y que el usuario pudiera sacarlos cuando los requiera. También hace falta tener más variedad de conectores.

Por último, aunque se refirió a los modelos tridimensionales de los planos de situación como gemelos digitales, se debe tener presente que, según las definiciones de gemelos digitales vistas en la introducción, esta implementación no corresponde a unos “auténticos gemelos digitales”, ya que faltarían sus contrapartes físicas reales. Sin embargo, en vista de que construir estos gemelos digitales en la realidad puede llegar a ser costoso, se desarrolla esta herramienta para que los estudiantes tengan acercamiento a lo que sería el funcionamiento “real” de los planos de situación neumáticos.

8. Bibliografía

- [1] Singh M., Fuenmayor E., Hinchy E., Qiao Y., Murray N., & Devine D. (2021). Digital Twin: Origin to Future. *Applied System Innovation*, 4(2), 36. Recuperado el 3 de mayo de 2023 de <https://doi.org/10.3390/asi4020036>
- [2] Toala F., Maldonado K., Toala M., Álava J. (2022). Gemelos digitales en la industria. *Revista Científica Arbitrada Multidisciplinaria PENTACIENCIAS - ISSN 2806-5794.*, 4(1), 75–83. Recuperado el 2 de mayo de 2023 de <https://www.editorialalema.org/index.php/pentaciencias/article/view/29>
- [3] Factoy I/O. (2023). Recuperado el 2 de mayo de <https://docs.factoryio.com/>
- [4] Iglesias M., Mujica H. (2021). Implementación del gemelo digital de un proceso secuencial electroneumático. *Memorias del congreso nacional automático ISSN: 2594 -2402*. Recuperado el 5 de mayo de 2023 de: <https://revistadigital.amca.mx/wp-content/uploads/2022/06/0077-1.pdf>
- [5] Silva J., Southworth M., Raptis C., Silva J. (2018). Emerging Application of Virtual Reality in Cardiovascular Medicine. *J Am Coll Cardiol Basic Trans Science*. Recuperado el 23 de marzo de 2023. <https://doi.org/10.1016/j.jacbts.2017.11.009>
- [6] Winkler I., Murari T., Ferreira C., Freitas F. (2021). VR-Based Product Development Process: Opportunities and Challenges in the Automotive Industry. *Journal of interconnection Networks*, 22. Recuperado el 23 de marzo de 2023. DOI: https://doi.org/10.5753/svr_estendido.2022.226711
- [7] Capecchi I., Neri F., Borghini T., Bernetti J. (2023). Use of virtual reality technology in chainsaw operations, education and training, *Forestry: An International Journal of Forest Research*, cpad007. Recuperado en 24 de marzo de 2023. <https://doi.org/10.1093/forestry/cpad007>
- [8] Bernardo A. (2017). Virtual Reality and Simulation in Neurosurgical Training. *World Neurosurgery*. Volume 106. Recuperado el 24 de marzo de 2023. <https://doi.org/10.1016/j.wneu.2017.06.140>.
- [9] Alho P., Ribeiro M., Alves M. (2015). Virtual reality applied to the study of the integration of transformers in substations of power systems. *The International Journal of Electrical Engineering & Education*, 53. Recuperado el 25 de marzo de 2023. <https://doi.org/10.1177/0020720915583865>

- [10] Pisanty A., Lucet G. (2015). Observatorio de visualización: Ixtli, Instalación de realidad virtual de la UNAM. Revista digital Universitaria, 6. Recuperado del 25 de marzo de 2023. <https://ru.tic.unam.mx/handle/123456789/1005>
- [11] Ramírez D. (2013). Aplicaciones de la realidad virtual en la ingeniería automotriz. Revista Digital Universitaria, 14. Recuperado el 25 de marzo de 2023. <https://www.revista.unam.mx/vol.14/num5/art06/>
- [12] Buendía M., Ambrosio J. (2012). Reconstrucción y visualización 3D del interior de las células. Revista Digital Universitaria, 13. Recuperado el 25 de marzo de 2023. <https://www.ru.tic.unam.mx/handle/123456789/2070>
- [13] Boyles, B. (2017). Virtual reality and augmented reality in education. Center For Teaching Excellence, United States Military Academy, West Point, Ny, 67.
- [14] Hurtado G., Martínez M., Cruz S., Bautista L. (2021). Simulador en realidad virtual para el desarrollo de prácticas de neumática. Congreso internacional anual de la SODIM. Recuperado el 1 de abril de 2023. https://somim.org.mx/memorias/memorias2021/articulos/A5_95.pdf
- [15] Hurtado G., Martínez M., Bautista L. (2022). Evaluación del Uso de un Simulador Neumático en Realidad Virtual en Ingeniería. XXVIII Congreso Internacional Anual de la SOMIM. Recuperado el 3 de abril de 2023 https://somim.org.mx/memorias/memorias2022/articulos/A5_71.pdf
- [16] Hurtado G., Bautista L., Martínez M., Reyes F. (2022). Laboratorio virtual de neumática en el metaverso para la enseñanza en ingeniería. XV Congreso Iberoamericano de Ingeniería Mecánica. Recuperado el 3 de abril de 2023 de http://e-spacio.uned.es/fez/eserv/bibliuned:congresoCIBIM-2022UPMEspana-Ghurtado/Abs_119_183013.pdf
- [17] Haas, J. K. (2014). A History of the Unity Game Engine. Recuperado el 6 de abril de 2023 de <https://digitalcommons.wpi.edu/igp-all/3207>
- [18] Lidon M. (2019). Unity 3D. AlfaOmega.

[19] Unity. (2016). Flujo de trabajo de los Assets. Recuperado el 6 de abril de 2023 de <https://docs.unity3d.com/es/530/Manual/AssetWorkflow.html>

[20] Unity. (2022). Asset workflow. Recuperado el 6 de abril de 2023 <https://docs.unity3d.com/2019.4/Documentation/Manual/AssetWorkflow.html>

[21] Unity (2022). GameObject. Recuperado el 23 de abril de 2023 de <https://docs.unity3d.com/2019.4/Documentation/Manual/GameObjects.html>

[22] Unity (2022). Transform. Recuperado el 8 de abril de 2023 de <https://docs.unity3d.com/2019.4/Documentation/Manual/class-Transform.html>

[23] Unity (2022). Rigidbody. Recuperado el 8 de abril de 2023 de <https://docs.unity3d.com/2019.4/Documentation/Manual/class-Rigidbody.html>

[24] Unity. (2022). Colliders. Recuperado el 9 de abril de 2023 de <https://docs.unity3d.com/es/2019.4/Manual/CollidersOverview.html>

[25] Unity. (2022). BoxCollider. Recuperado el 9 de abril de 2023 de <https://docs.unity3d.com/es/2019.4/Manual/class-BoxCollider.html>

[26] Unity. (2022). Physic Material component reference. Recuperado el 9 de abril de 2023 de <https://docs.unity3d.com/2019.4/Documentation/Manual/class-PhysicMaterial.html>

[27] Unity (2022). Animation. Recuperado el 23 de abril de 2023 de <https://docs.unity3d.com/Manual/AnimationOverview.html>

[28] Unity (2022). Material parameters. Recuperado el 2 de mayo de 2023 de <https://docs.unity3d.com/2019.4/Documentation/Manual/StandardShaderMaterialParameterHeightMap.html>

[29] Unity (2022). Albedo Color and Transparency. Recuperado el 15 de mayo de 2023 de <https://docs.unity3d.com/2019.4/Documentation/Manual/StandardShaderMaterialParameterAlbedoColor.html>

[30] Texturas 3D (2023). ¿Qué son las texturas PBR? Recuperado el 2 de mayo de 2023 de

<https://www.texturas3d.com/fotogrametria/texturas-pbr/>

[31] Baechler O., Greer X. (2020). Blender 3D by Example. Second Edition.

https://books.google.com.mx/books?id=4LoDwAAQBAJ&printsec=frontcover&hl=es&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=true

[32] Blender (2023). Materials. Introduction. Recuperado el 24 de abril de 2023 de

https://docs.blender.org/manual/en/dev/getting_started/about/introduction.html

[33] Blender (2023). Rendering. Introduction. Recuperado el 30 de abril de 2023 de

<https://docs.blender.org/manual/en/3.5/render/introduction.html>

[34] Blender (2023). Eevee. Introduction. Recuperado el 30 de abril de 2023 de

<https://docs.blender.org/manual/en/3.5/render/eevee/introduction.html>

[35] Saffo D., Yildirim C., Bartolomeo S., & Dunne C. (2020). Crowdsourcing Virtual Reality Experiments using VRChat. In Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems (CHI EA '20). Association for Computing Machinery, New York, NY, USA, 1–8. Recuperado el 17 de mayo de 2023 de:

<https://doi.org/10.1145/3334480.3382829>

[36] Festo. (2023). Software de configuración CAD Festo Design Tool 3D. Recuperado el 6 de mayo de 2023 de:

https://www.festo.com/mx/es/e/soluciones/digitalizacion/software-de-ingenieria/festo-design-tool-3d-id_330026/

[37] Traceparts. (2023). Biblioteca de archivos 3D. Recuperado el 6 de mayo de:

<https://www.traceparts.com/es>

[38] Espíndola F., Yeber P. (2021). The Unity shaders Bible.

[39] Unity. (2021). Creating models for optimal performance. Recuperado el 10 de mayo de 2023 de:

<https://docs.unity3d.com/2019.4/Documentation/Manual/ModelingOptimizedCharacters.html>

[40] Freepik. (2023). Imágenes De Textura Metal. Recuperado el 11 de mayo de 2023 de

<https://www.freepik.es/fotos-vectores-gratis/textura-metal>

[41] TextureCan (2023). Textures. Recuperado el 15 de mayo de 2023 de

<https://www.texturecan.com/>

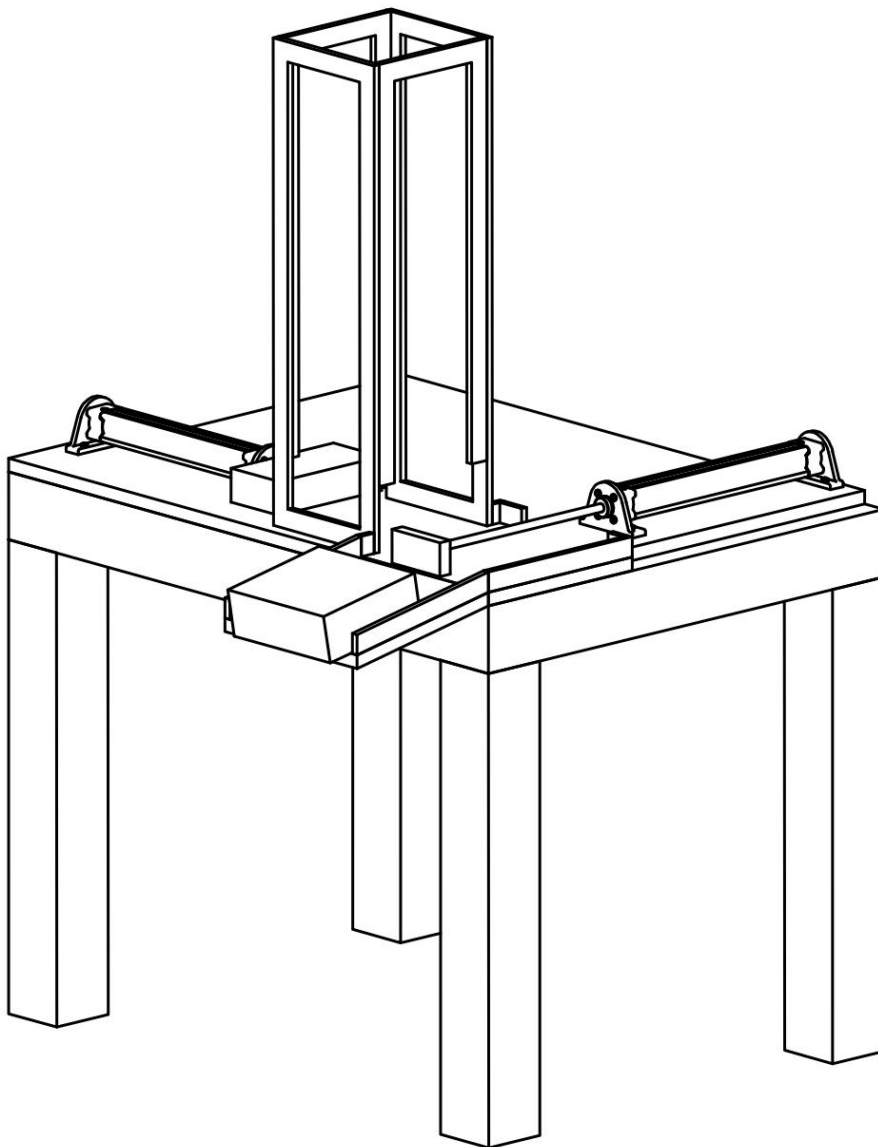
[42] PNGitem (2019). Recuperado el 25 de mayo de 2023 de


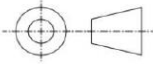
https://www.pngitem.com/middle/iihJxhx_art-hole-bullethole-stickers-bullet-hole-sticker-hd/

[43] Canva (2023). <https://www.canva.com/>

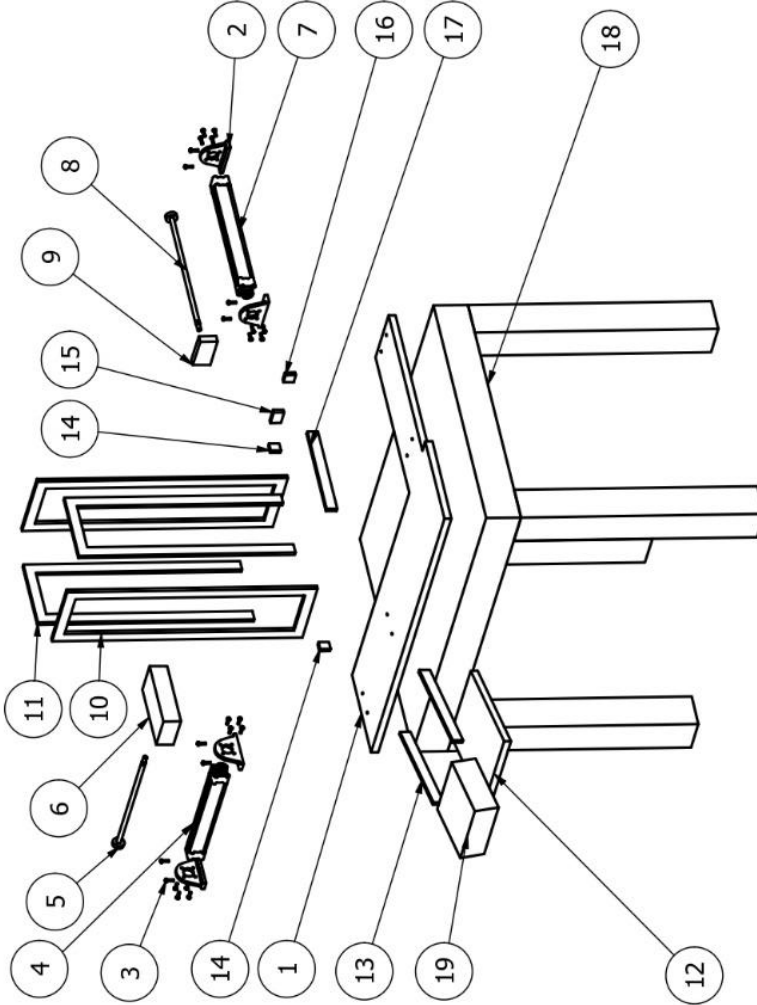
9. Apéndice A. Planos del Gemelo Digital de la Empacadora



1. Isométrico Ensamble Empacadora
2. Vista Explosionada de la Empacadora
3. Paquete
4. Base
5. Soporte Cilindro
6. Cabezal Vástago 260 mm
7. Cabezal Vástago 300 mm
8. Soporte Paquete 1
9. Soporte Paquete 2
10. Rampa
11. Guía Rampa
12. Guía 1
13. Guía 2
14. Guía 3
15. Guía 4
16. Mesa
17. Cilindro de Carrera de 260 mm Marca Festo
18. Cilindro de Carrera de 300 mm Marca Festo
19. Vástago del Cilindro de Carrera de 260 mm Marca Festo
20. Vástago del Cilindro de Carrera de 300 mm Marca Festo

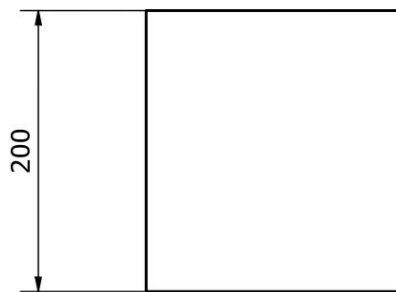
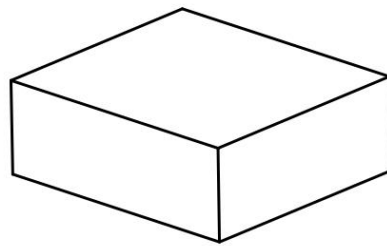
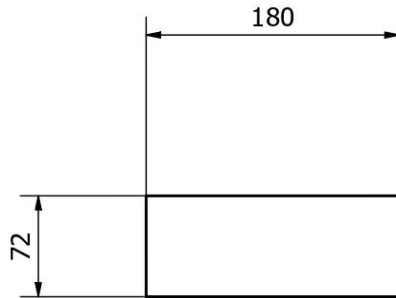


	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Isométrico Ensamble Empacadora				
	Fecha: 06/06/2023	Escala: 1:21	Unidades: mm	No. de hoja: 1/20	Tamaño: A4

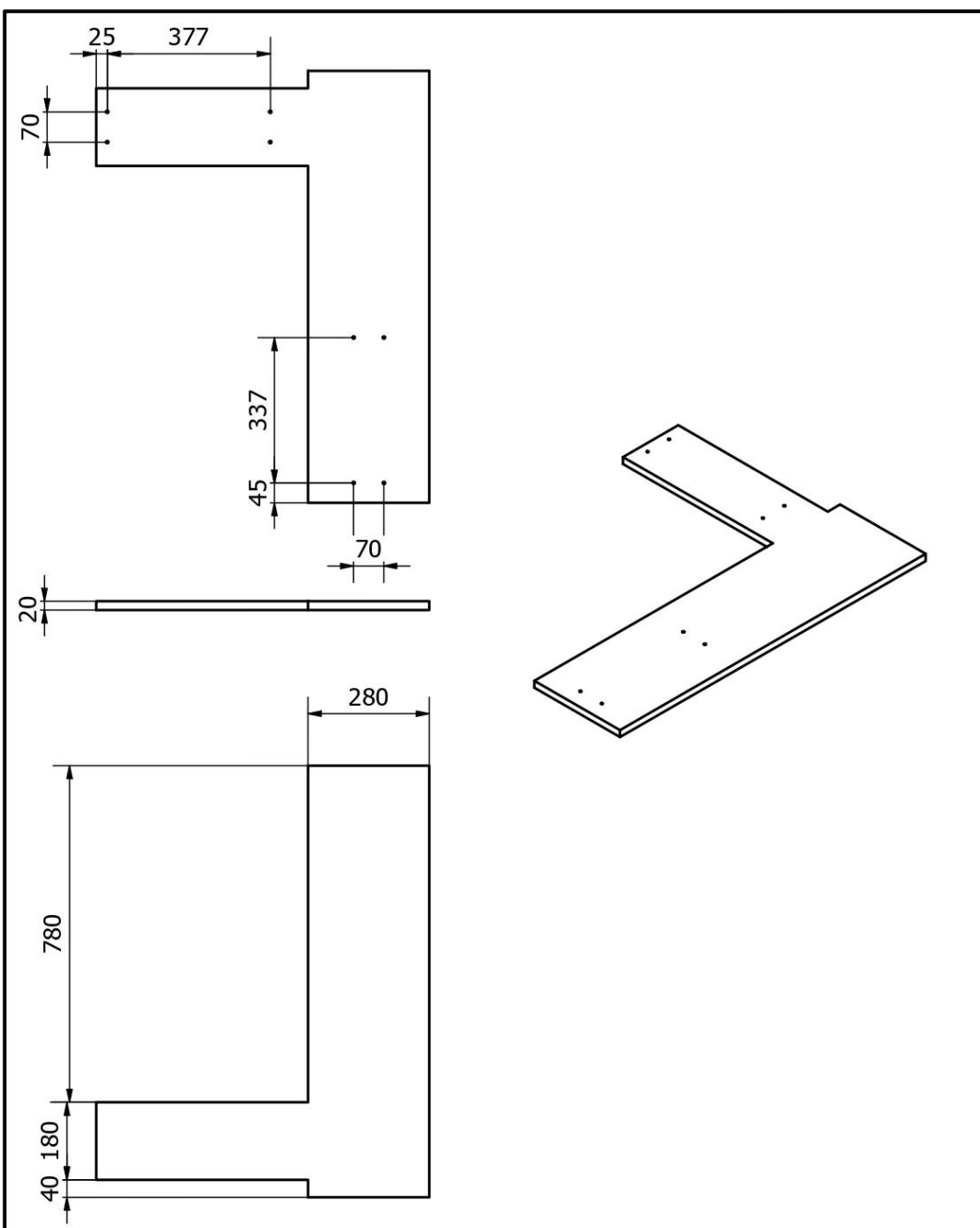
PARTS LIST	
ITEM	QTY PART NUMBER
1	1 Base
2	4 Soporte Cilindro
3	24 Perno M5
4	1 Cilindro 260 mm
5	1 Vástago 260
6	1 Cabezal 260
7	1 Cilindro 300 mm
8	1 Vástago 300
9	1 Cabezal 300
10	2 Soporte Paquete 1
11	2 Soporte Paquete 2
12	1 Rampa
13	2 GuiaRampa
14	2 Guía 1
15	1 Guía 2
16	1 Guía 3
17	1 Guía 4
18	1 Mesa
19	1 Paquete


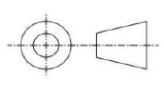


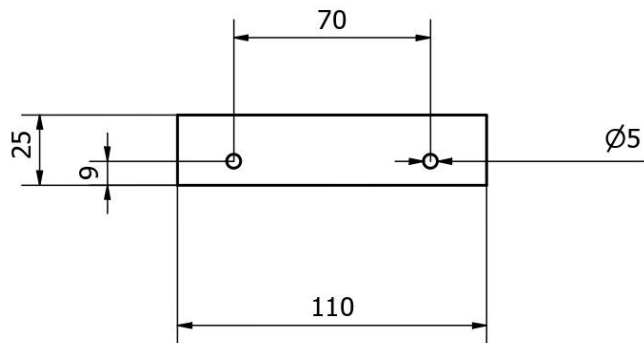
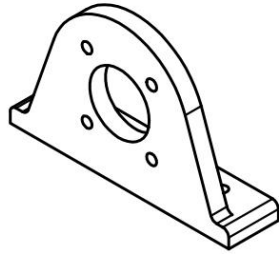
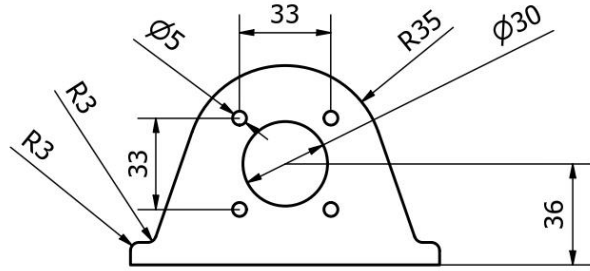
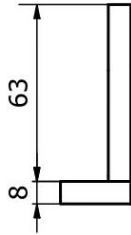
	Autor:	Inti López Lucero	Revisado por:	M.F. Gabriel Hurtado Chong
	Título:	Vista Explosionada de la Empacadora		
	Fecha:	06/06/2023	Escala:	1:15
			Unidades:	mm
			No. de hoja:	2/20
			Tamaño:	A4



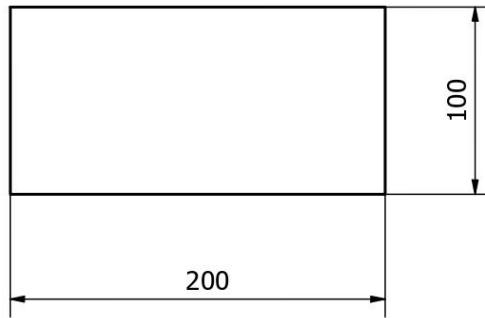
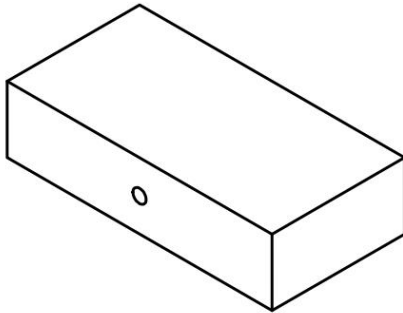
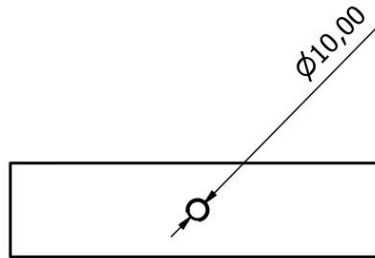
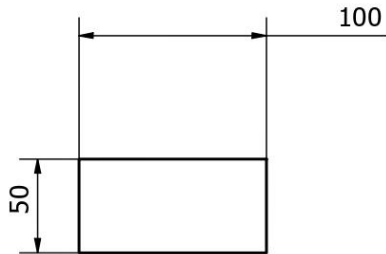
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Paquete				
	Fecha: 06/06/2023	Escala: 1:15	Unidades: mm	No. de hoja: 3/20	Tamaño: A4



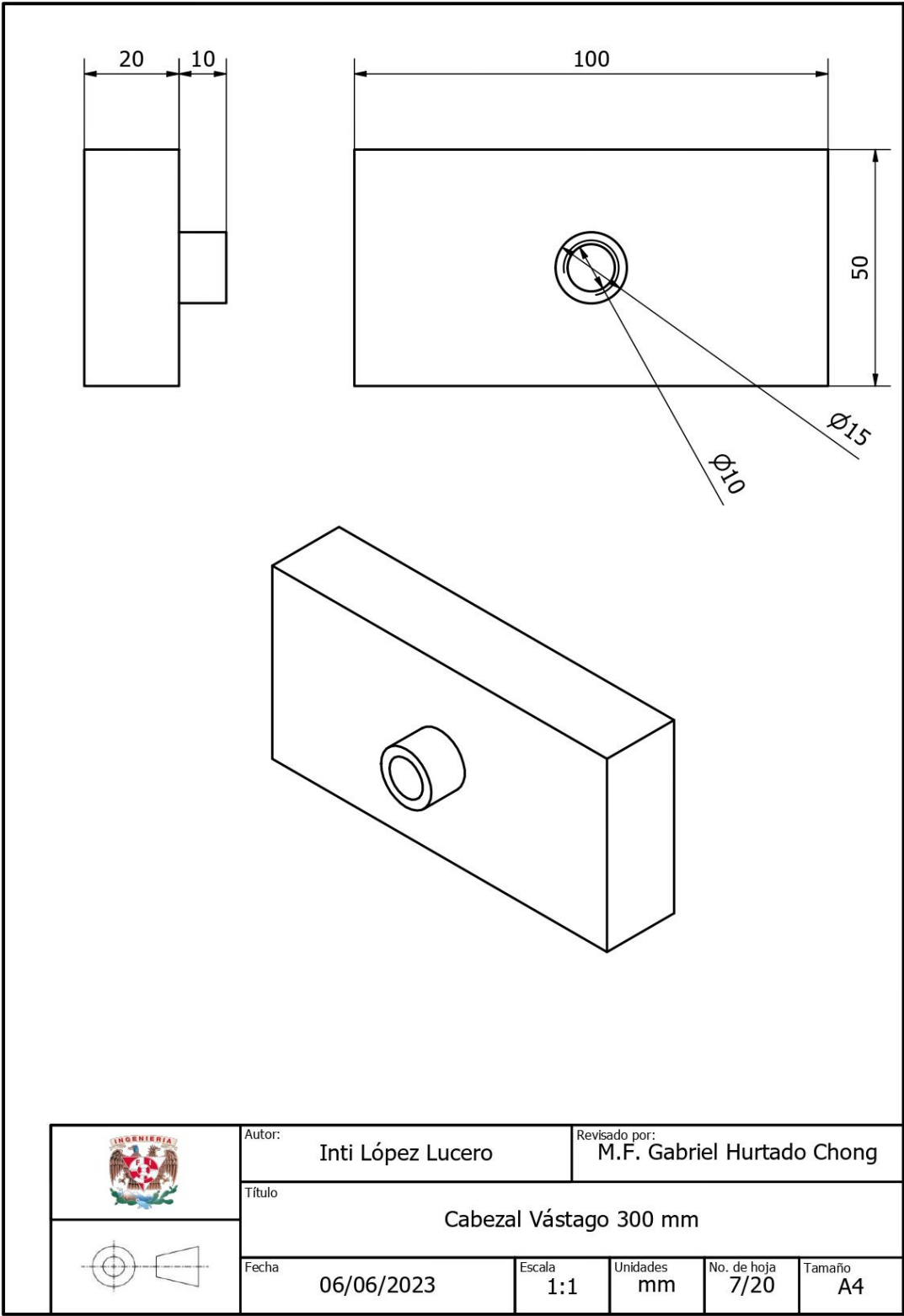
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Base				
	Fecha: 06/06/2023	Escala: 1:12	Unidades: mm	No. de hoja: 4/20	Tamaño: A4


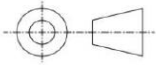


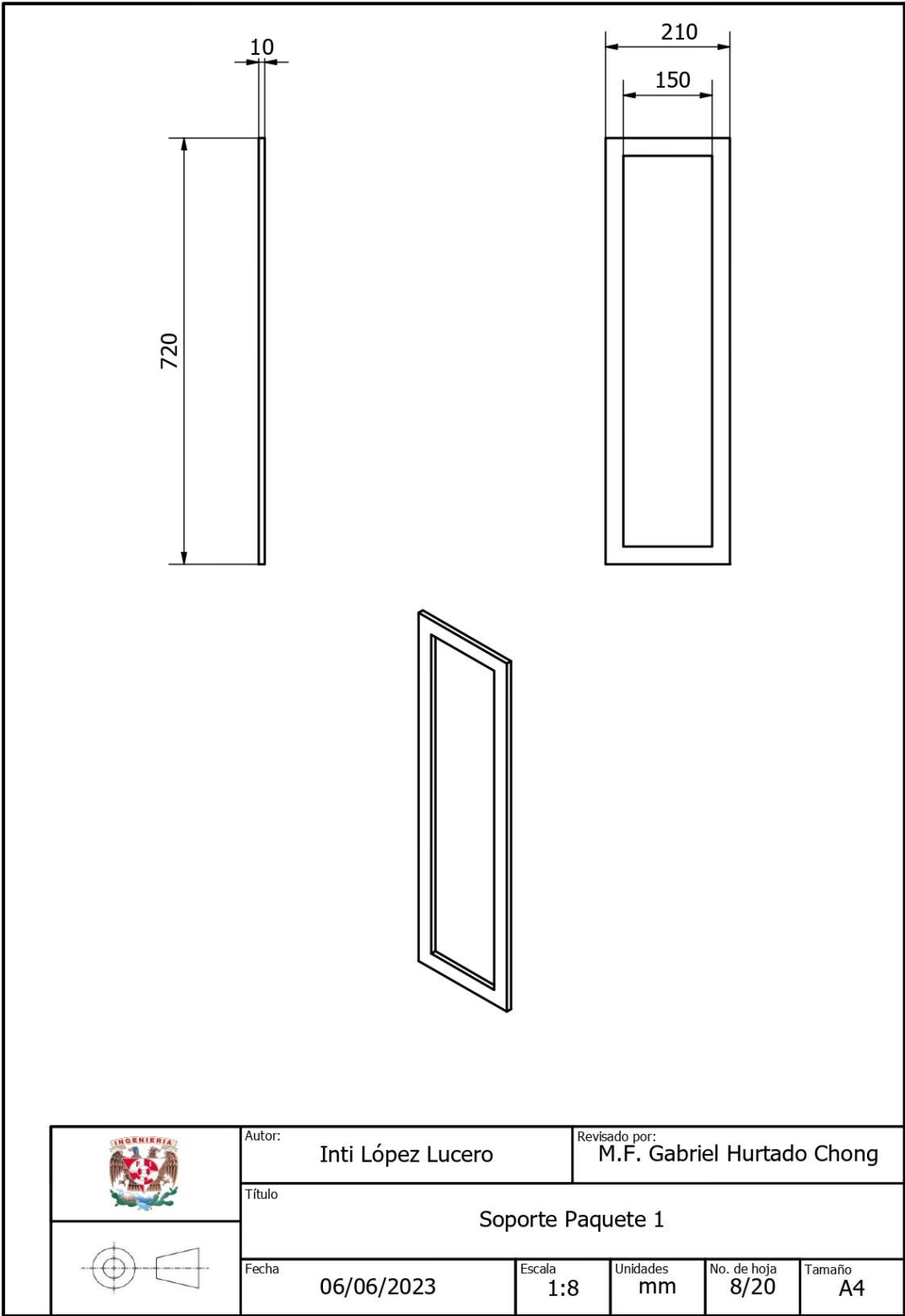
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Soporte Cilindro				
	Fecha: 06/06/2023	Escala: 1:2	Unidades: mm	No. de hoja: 5/20	Tamaño: A4



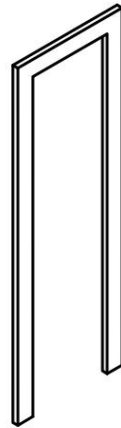
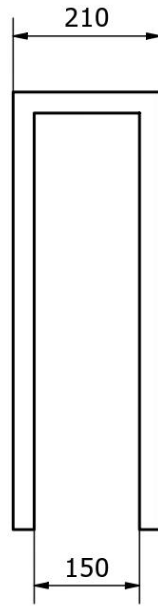
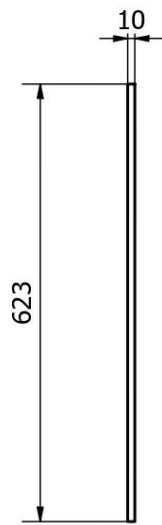
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Cabezal Vástago 260 mm				
	Fecha: 06/06/2023	Escala: 1:3	Unidades: mm	No. de hoja: 6/20	Tamaño: A4


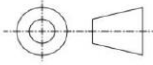


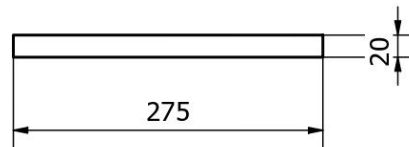
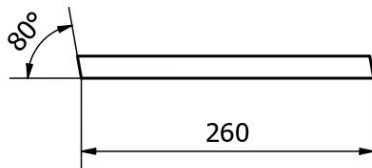
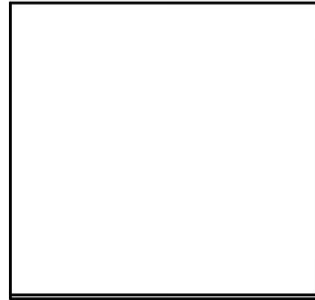
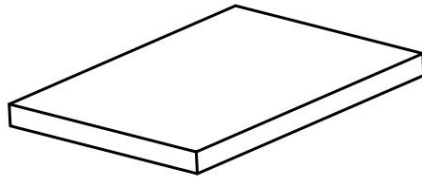
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Cabezal Vástago 300 mm				
	Fecha: 06/06/2023	Escala: 1:1	Unidades: mm	No. de hoja: 7/20	Tamaño: A4



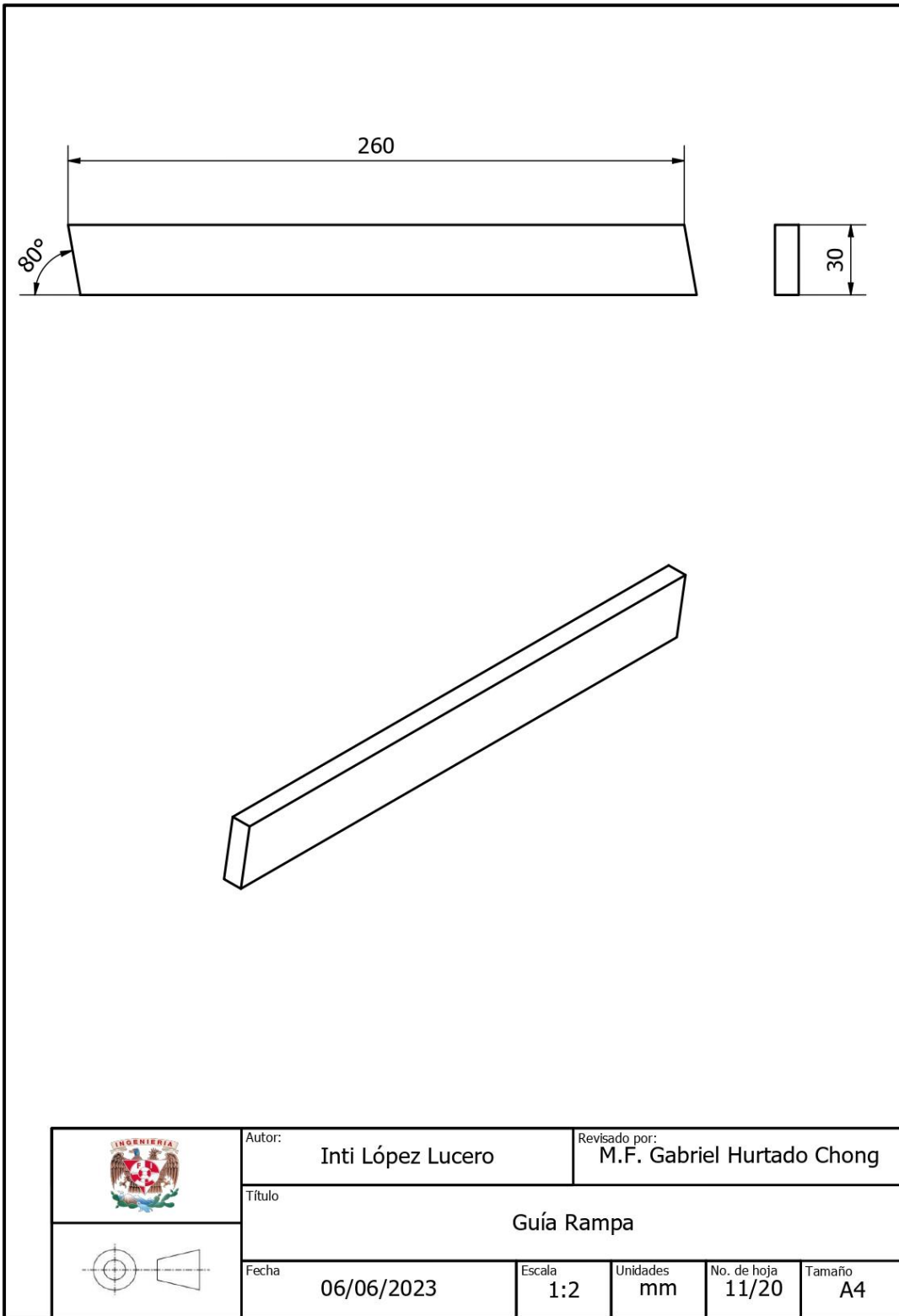
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Soporte Paquete 1				
	Fecha: 06/06/2023	Escala: 1:8	Unidades: mm	No. de hoja: 8/20	Tamaño: A4

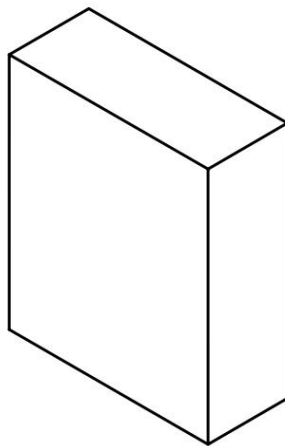
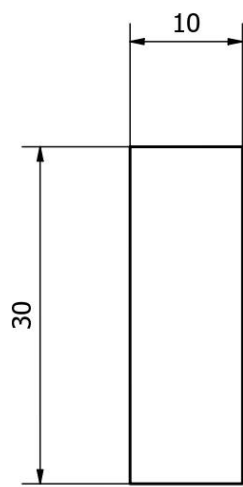


	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Soporte Paquete 2				
	Fecha: 06/06/2023	Escala: 1:8	Unidades: mm	No. de hoja: 9/20	Tamaño: A4

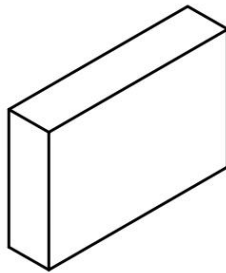
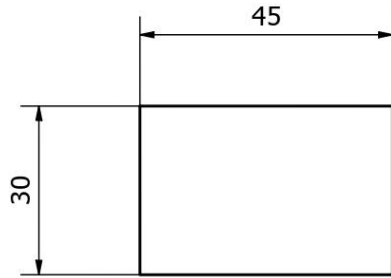



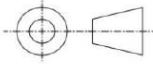
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Rampa				
	Fecha: 06/06/2023	Escala: 1:5	Unidades: mm	No. de hoja: 10/20	Tamaño: A4

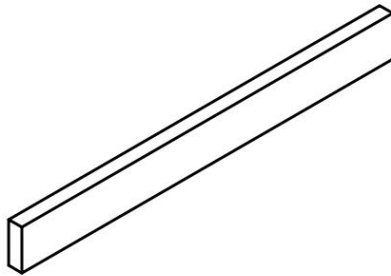
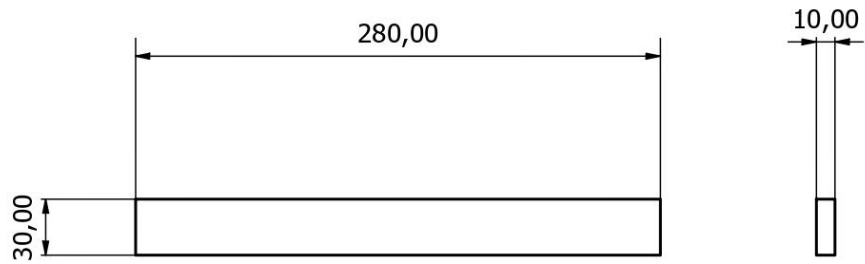



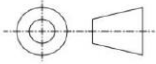


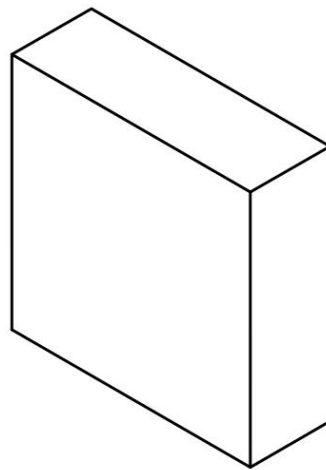
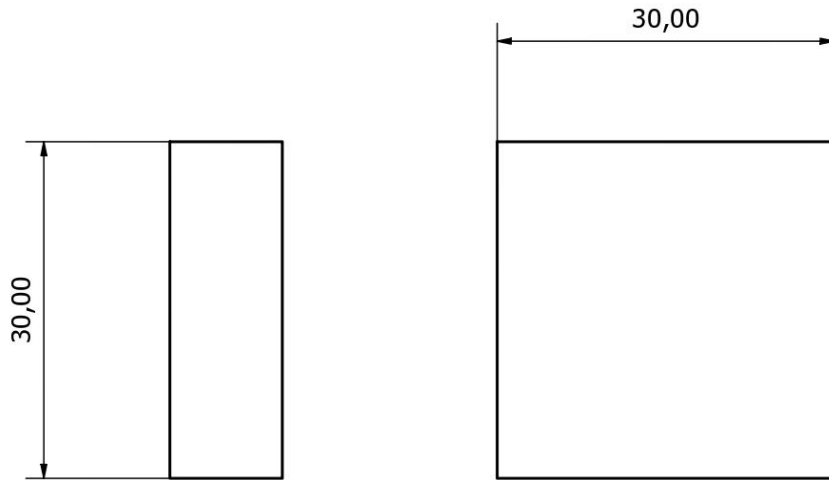
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Guía 1				
	Fecha: 06/06/2023	Escala: 2:1	Unidades: mm	No. de hoja: 12/20	Tamaño: A4


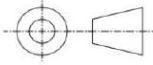


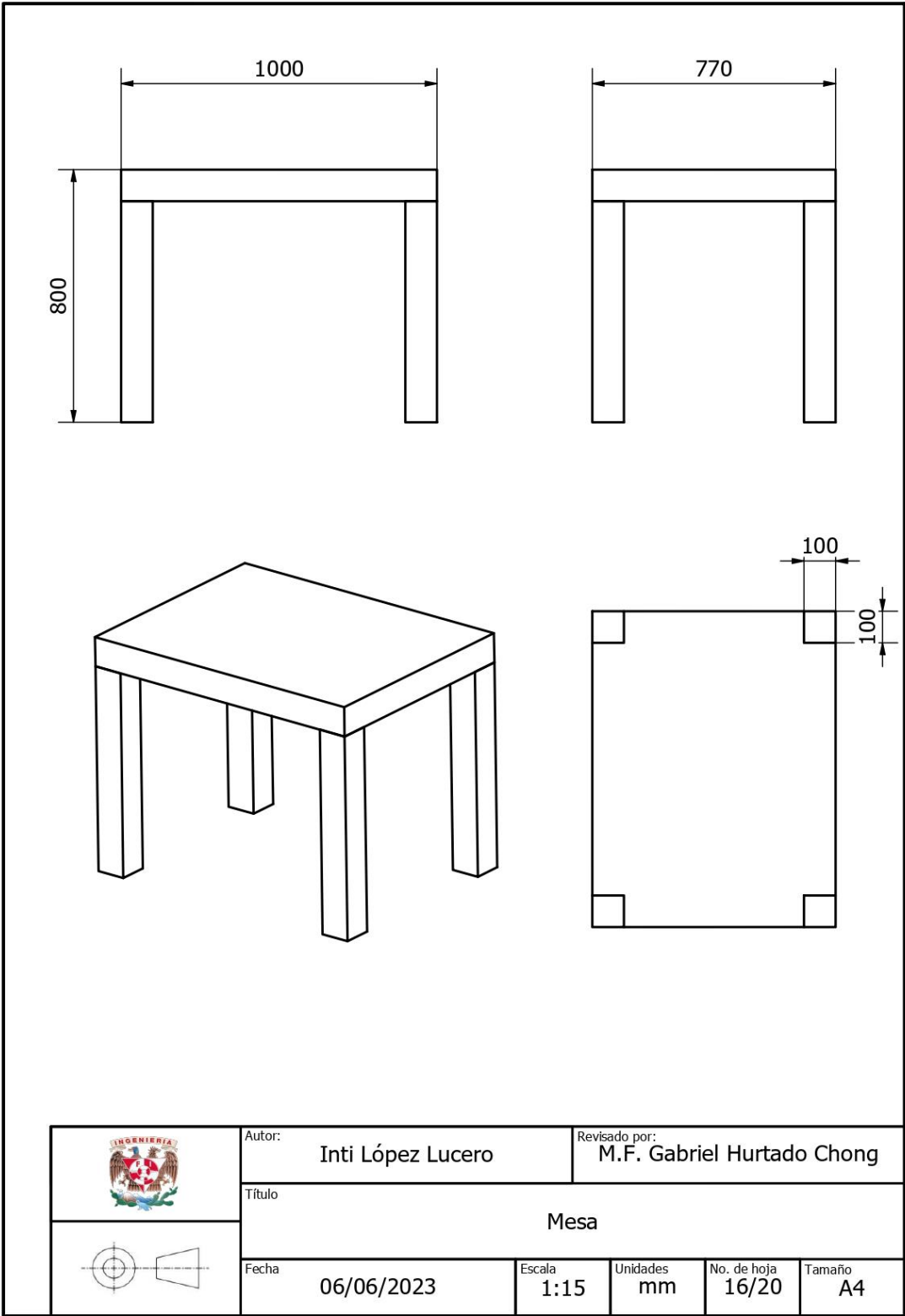
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Guía 2				
	Fecha: 06/06/2023	Escala: 1:1	Unidades: mm	No. de hoja: 13/20	Tamaño: A4



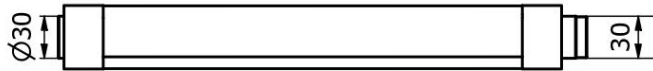
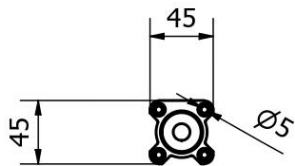
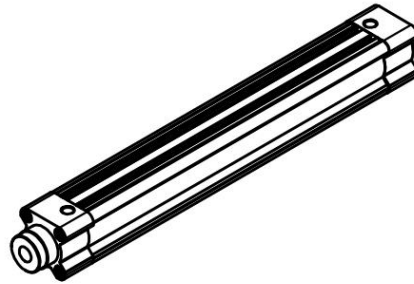
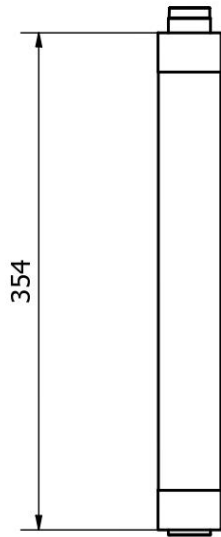
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Guía 3				
	Fecha: 06/06/2023	Escala: 1:2	Unidades: mm	No. de hoja: 14/20	Tamaño: A4



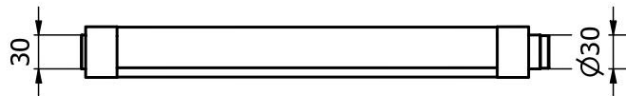
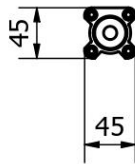
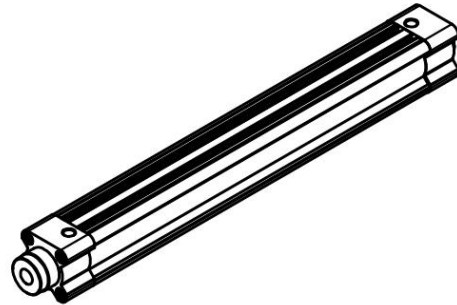
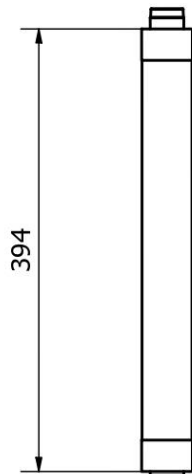
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Guía 4				
	Fecha: 06/06/2023	Escala: 2:1	Unidades: mm	No. de hoja: 15/20	Tamaño: A4


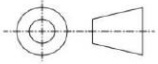


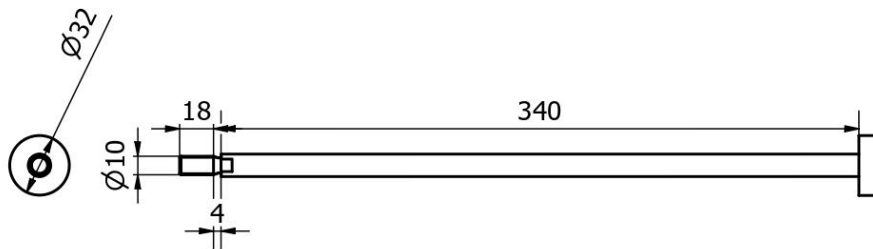
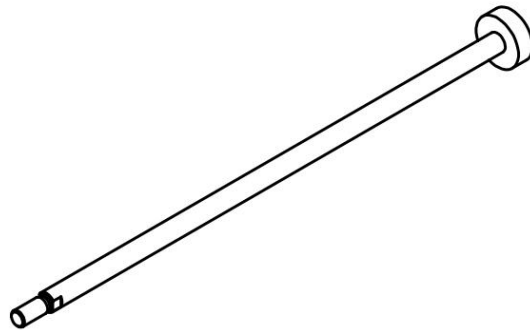
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Mesa				
	Fecha: 06/06/2023	Escala: 1:15	Unidades: mm	No. de hoja: 16/20	Tamaño: A4



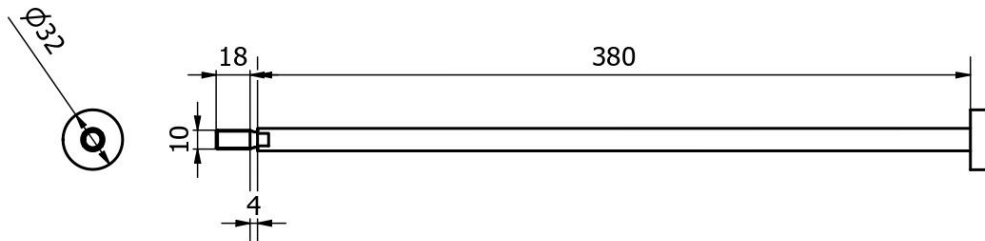
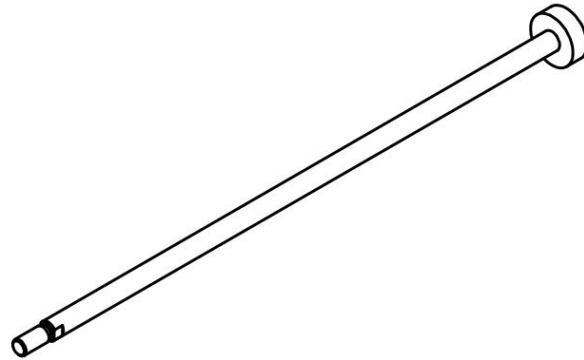
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Cilindro de Carrera de 260 mm Marca Festo				
	Fecha: 07/07/2023	Escala: 1:4	Unidades: mm	No. de hoja: 17/20	Tamaño: A4



	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Cilindro de Carrera 300 mm Marca Festo				
	Fecha: 07/06/2023	Escala: 1:5	Unidades: mm	No. de hoja: 18/20	Tamaño: A4



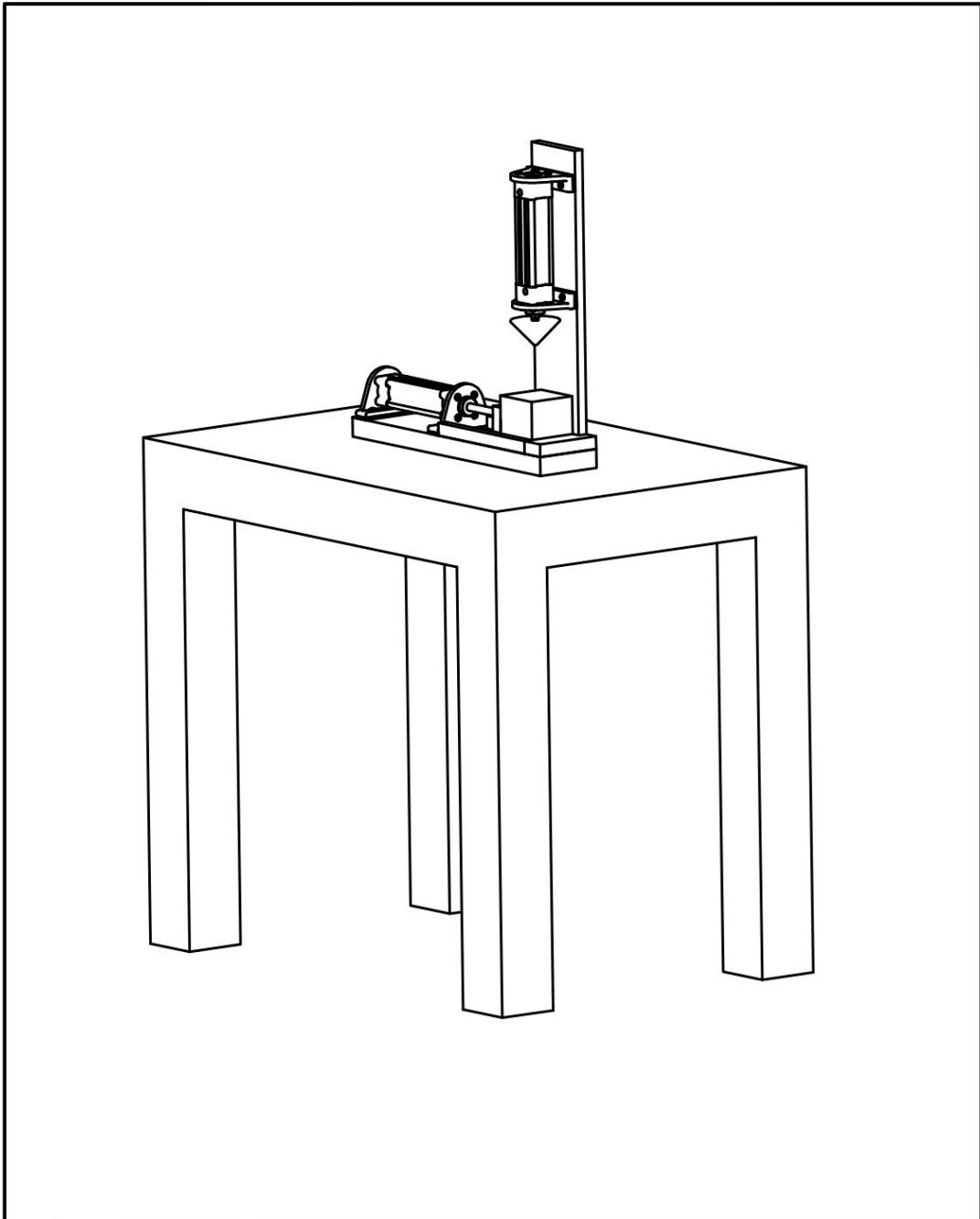
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Vástago del Cilindro de Carrera de 260 mm Marca Festo				
	Fecha: 07/06/2023	Escala: 1:3	Unidades: mm	No. de hoja: 19/20	Tamaño: A4



	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Vástago del Cilindro de Carrera de 300 mm Marca Festo				
	Fecha: 07/06/2023	Escala: 1:3	Unidades: mm	No. de hoja: 20/20	Tamaño: A4

10. Apéndice B. Planos del Gemelo Digital de la Indentadora

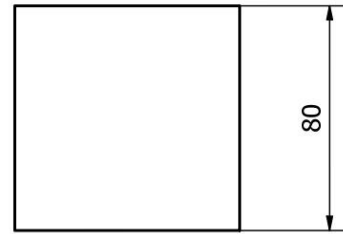
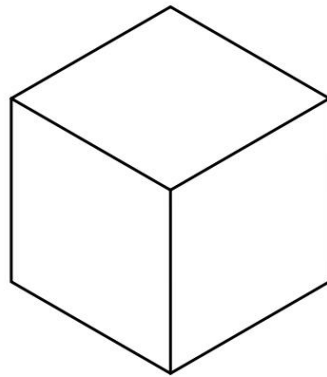
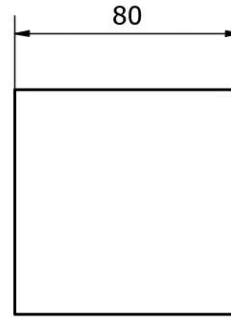
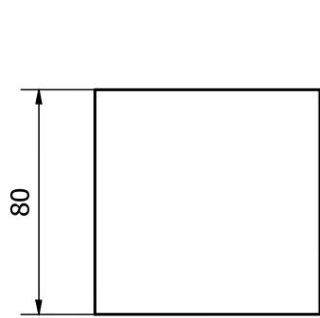
1. Isométrico Ensamble Indentadora
2. Vista Explosionada de la Indentadora
3. Pieza Cubo
4. Cabezal A
5. Cabezal Indentador
6. Base Horizontal
7. Base Vertical
8. Guía 1
9. Guía 2
10. Soporte Elevador
11. Soporte Cilindro
12. Mesa
13. Cilindro de Carrera de 100 mm Marca Festo
14. Vástago del Cilindro de Carrera de 100 mm Marca Festo



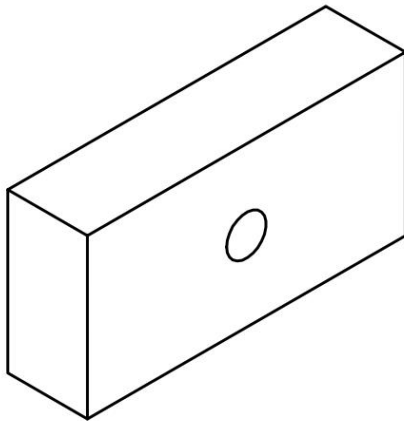
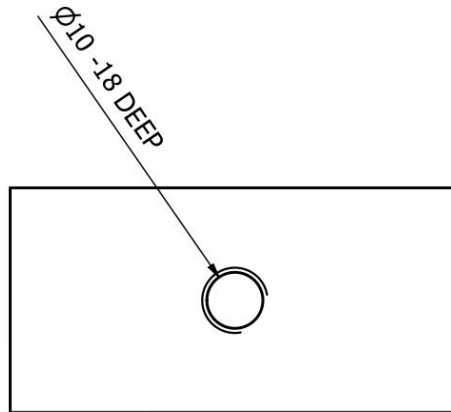
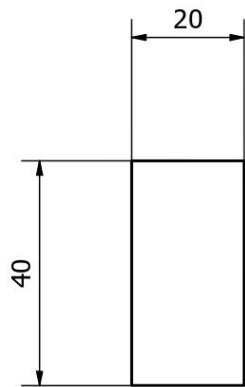
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Isométrico Ensamble Indentadora				
	Fecha: 07/06/2023	Escala: 1:8	Unidades: mm	No. de hoja: 1/14	Tamaño: A4

ITEM	QTY	PART NUMBER
1	1	Pieza Cubo
2	2	Cilindro 100 mm
3	2	Vástago 100
4	1	Cabezal A
5	1	Cabezal Indentador
6	1	Base Horizontal
7	1	Base Vertical
8	2	Guía 1
9	1	Guía 2
10	4	Soporte Elevador
11	4	Soporte Cilindro
12	24	Perno M5
13	1	Mesa

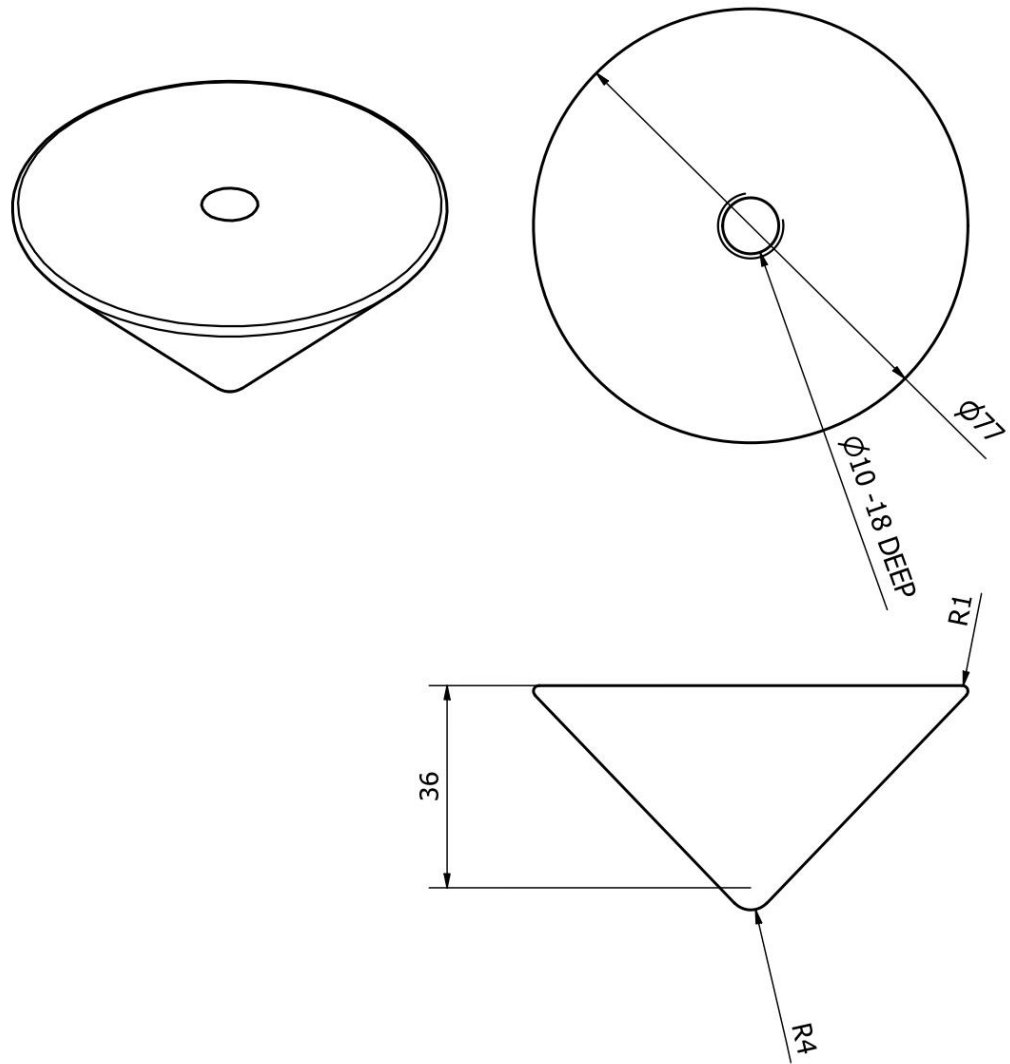
	Autor: Inti López Lucero	Revisado por: M.F. Gabriel Hurtado Chong
	Vista Explorcionada de la Indentadora	
Fecha: 06/06/2023	Escala: 1:14	Unidades: mm
	No. de hoja: 2/14	Tamaño: A4


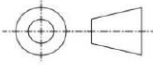


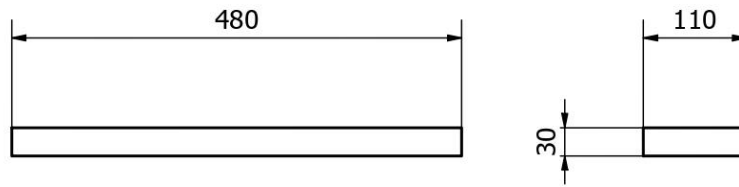
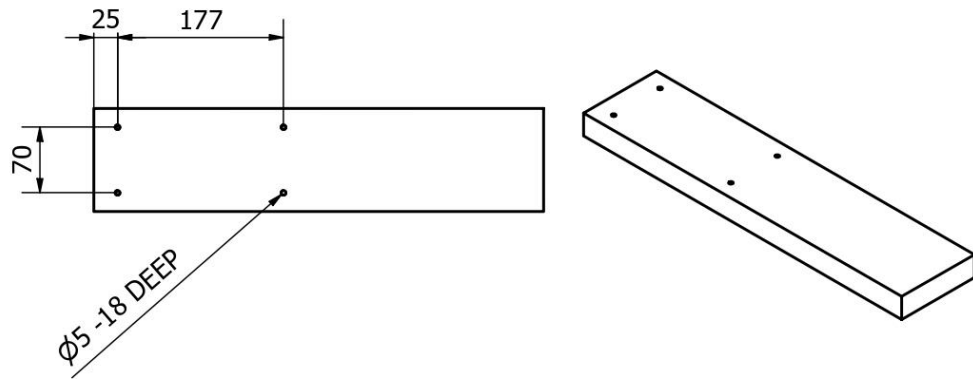
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Pieza Cubo				
	Fecha: 06/06/2023	Escala: 1:2	Unidades: mm	No. de hoja: 3/14	Tamaño: A4



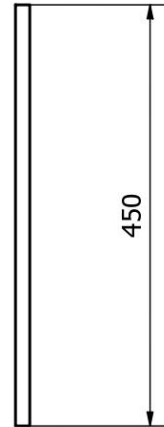
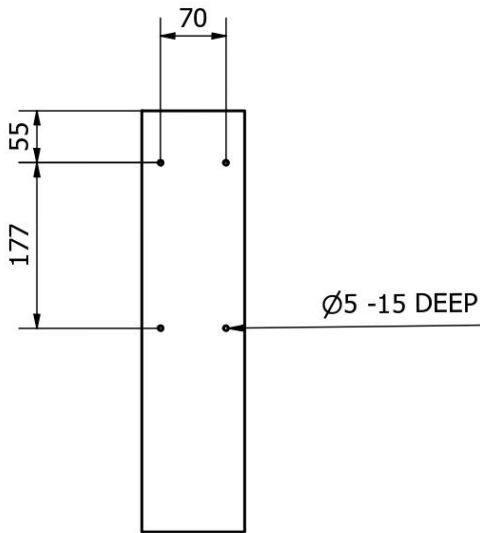
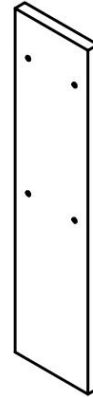
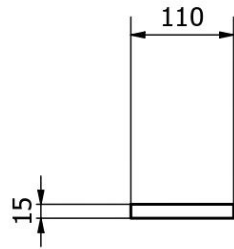
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Cabezal A				
	Fecha: 06/06/2023	Escala: 1:1	Unidades: mm	No. de hoja: 4/14	Tamaño: A4



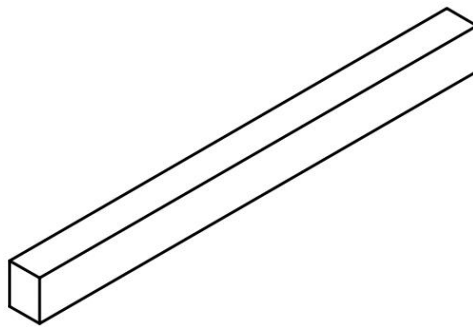
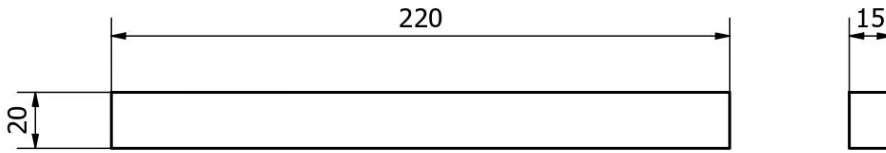
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Cabezal Indentador				
	Fecha: 06/06/2023	Escala: 1:1	Unidades: mm	No. de hoja: 5/14	Tamaño: A4


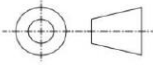


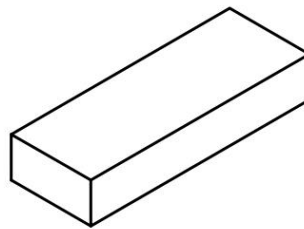
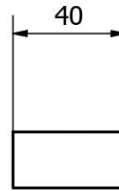
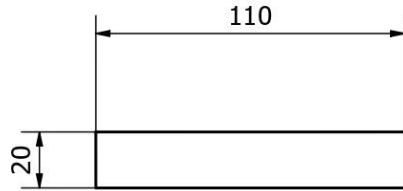
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Base Horizontal				
	Fecha: 07/06/2023	Escala: 1:6	Unidades: mm	No. de hoja: 6/14	Tamaño: A4



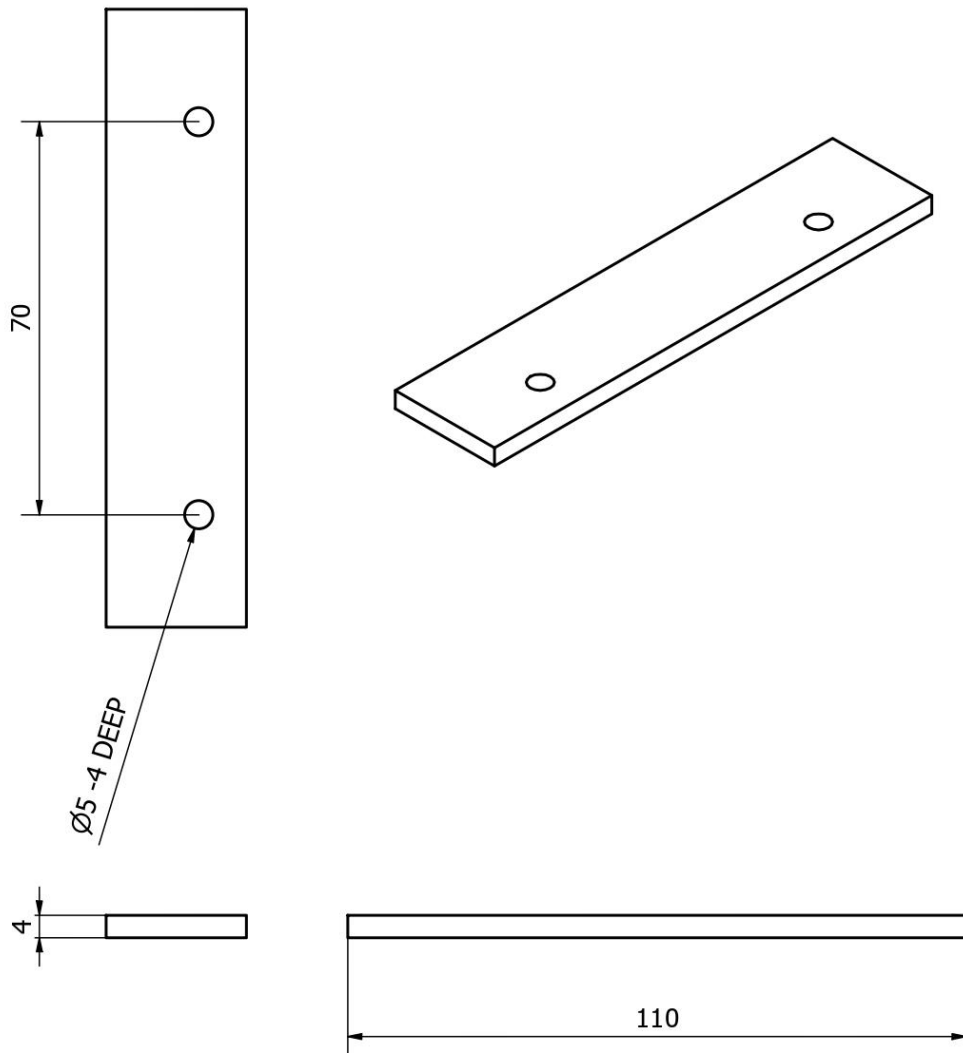
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Base Vertical				
	Fecha: 07/06/2023	Escala: 1:6	Unidades: mm	No. de hoja: 7/14	Tamaño: A4



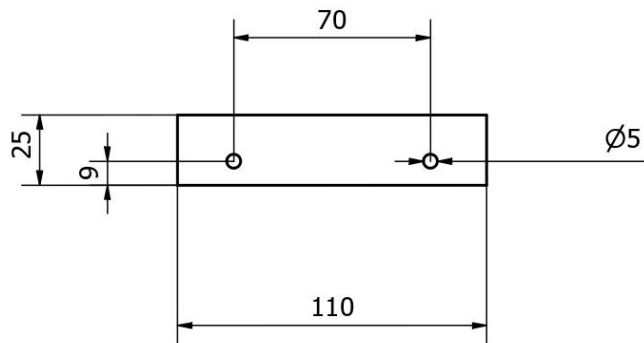
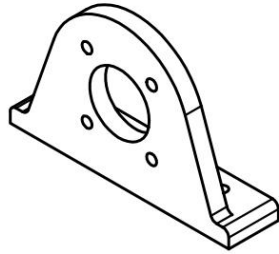
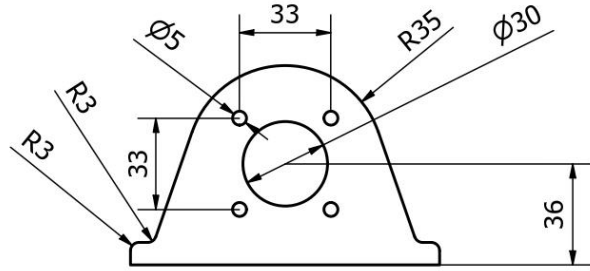
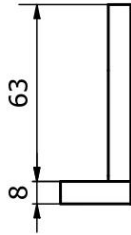
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Guía 1				
	Fecha: 06/06/2023	Escala: 1:2	Unidades: mm	No. de hoja: 8/14	Tamaño: A4


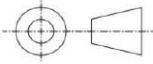


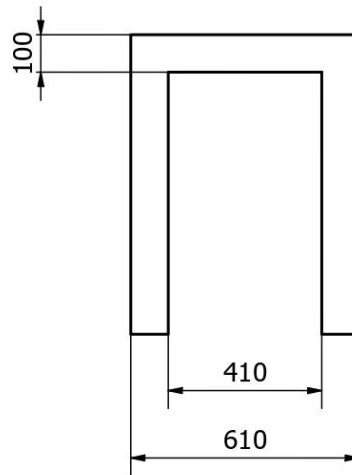
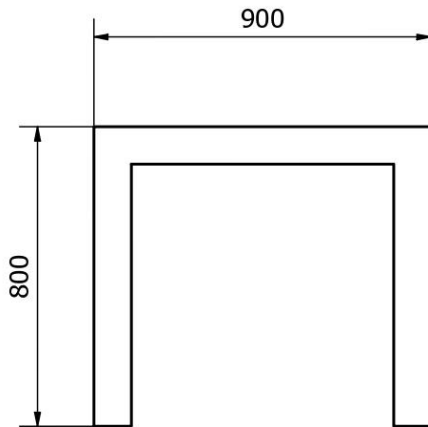
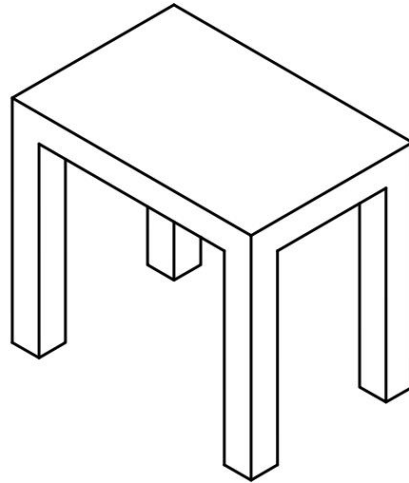
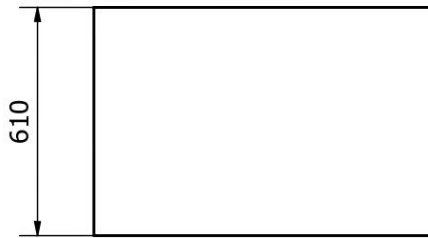
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Guía 2				
	Fecha: 06/06/2023	Escala: 1:2	Unidades: mm	No. de hoja: 9/14	Tamaño: A4


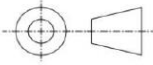


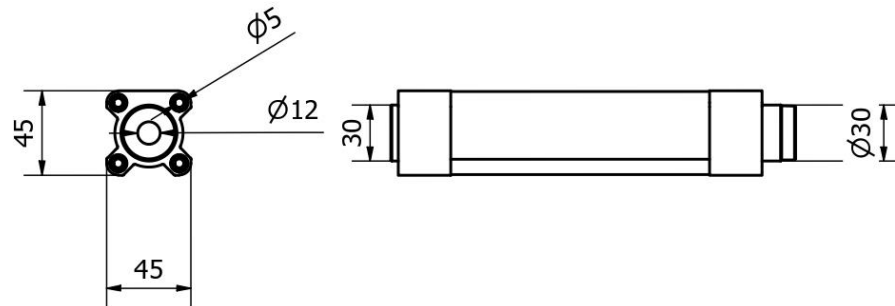
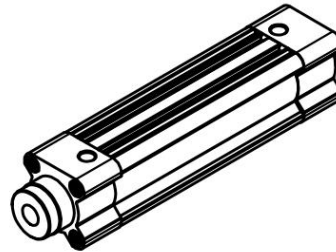
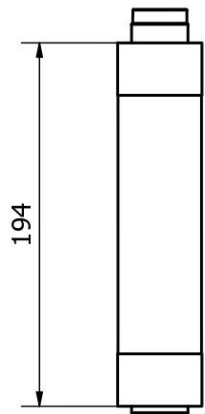
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Soporte Elevador				
	Fecha: 07/07/2023	Escala: 1:1	Unidades: mm	No. de hoja: 10/14	Tamaño: A4



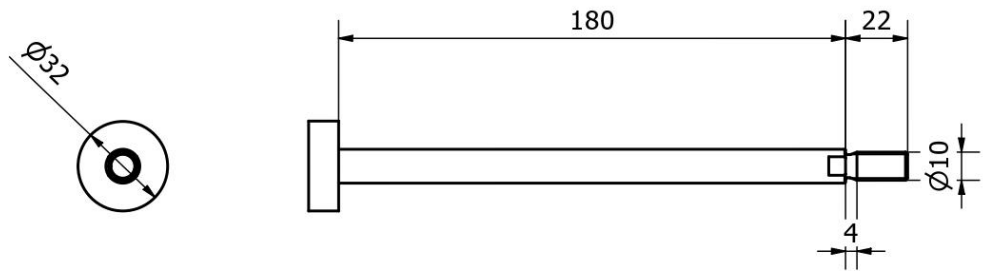
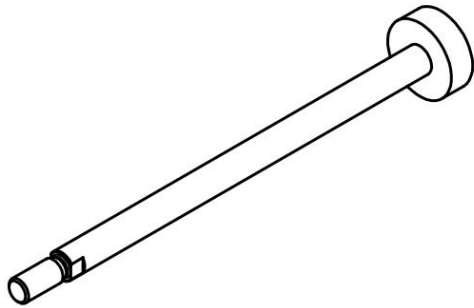
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Soporte Cilindro				
	Fecha: 06/06/2023	Escala: 1:2	Unidades: mm	No. de hoja: 11/14	Tamaño: A4


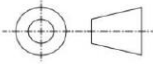


	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Mesa				
	Fecha: 07/07/2023	Escala: 1:15	Unidades: mm	No. de hoja: 12/14	Tamaño: A4



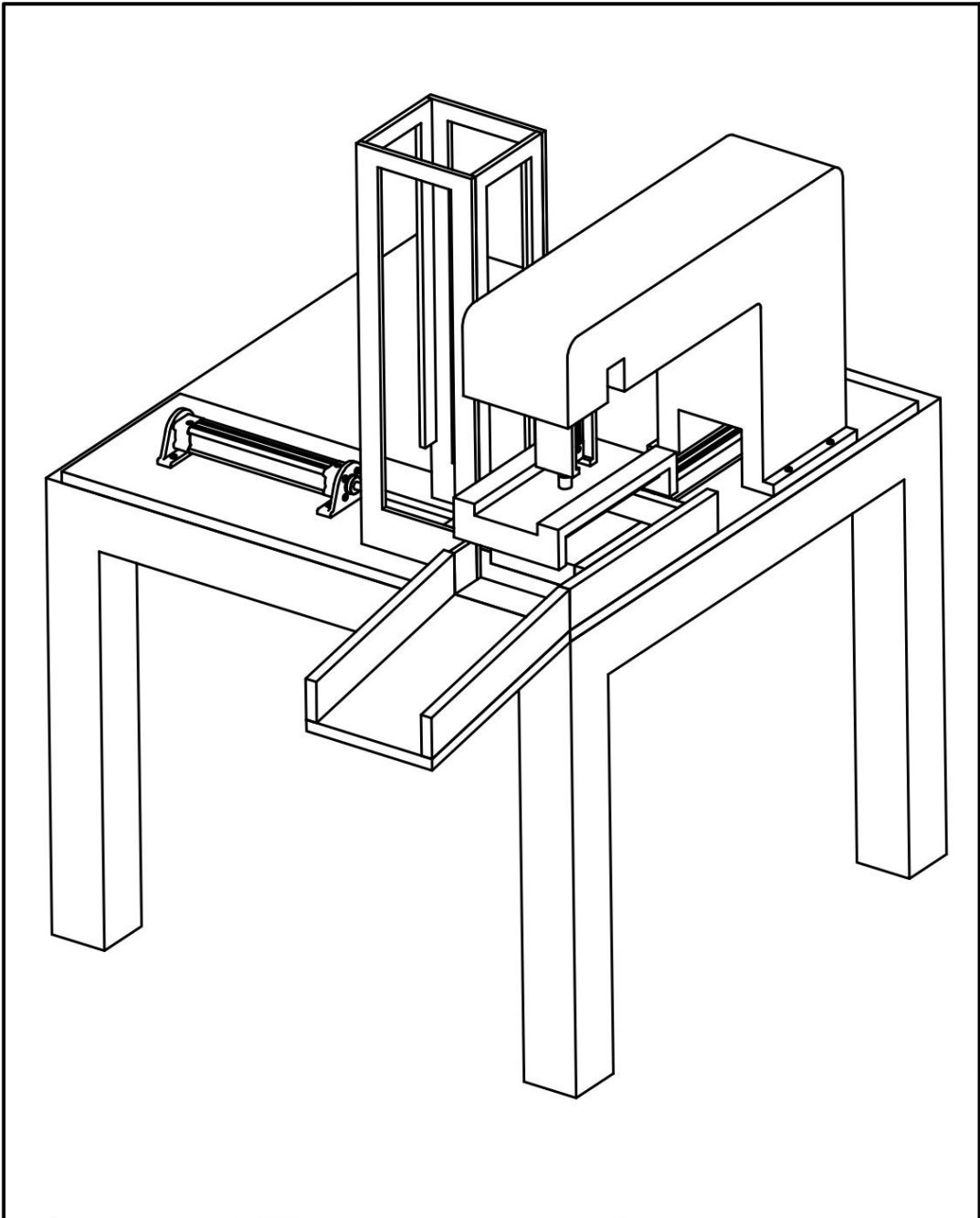
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Cilindro de Carrera de 100 mm Marca Festo				
	Fecha: 07/07/2023	Escala: 1:3	Unidades: mm	No. de hoja: 13/14	Tamaño: A4



	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Vástago del Cilindro de Carrera de 100mm Marca Festo				
	Fecha: 07/07/2023	Escala: 1:2	Unidades: mm	No. de hoja: 14/14	Tamaño: A4


11. Apéndice C. Planos del Gemelo Digital de la Estampadora

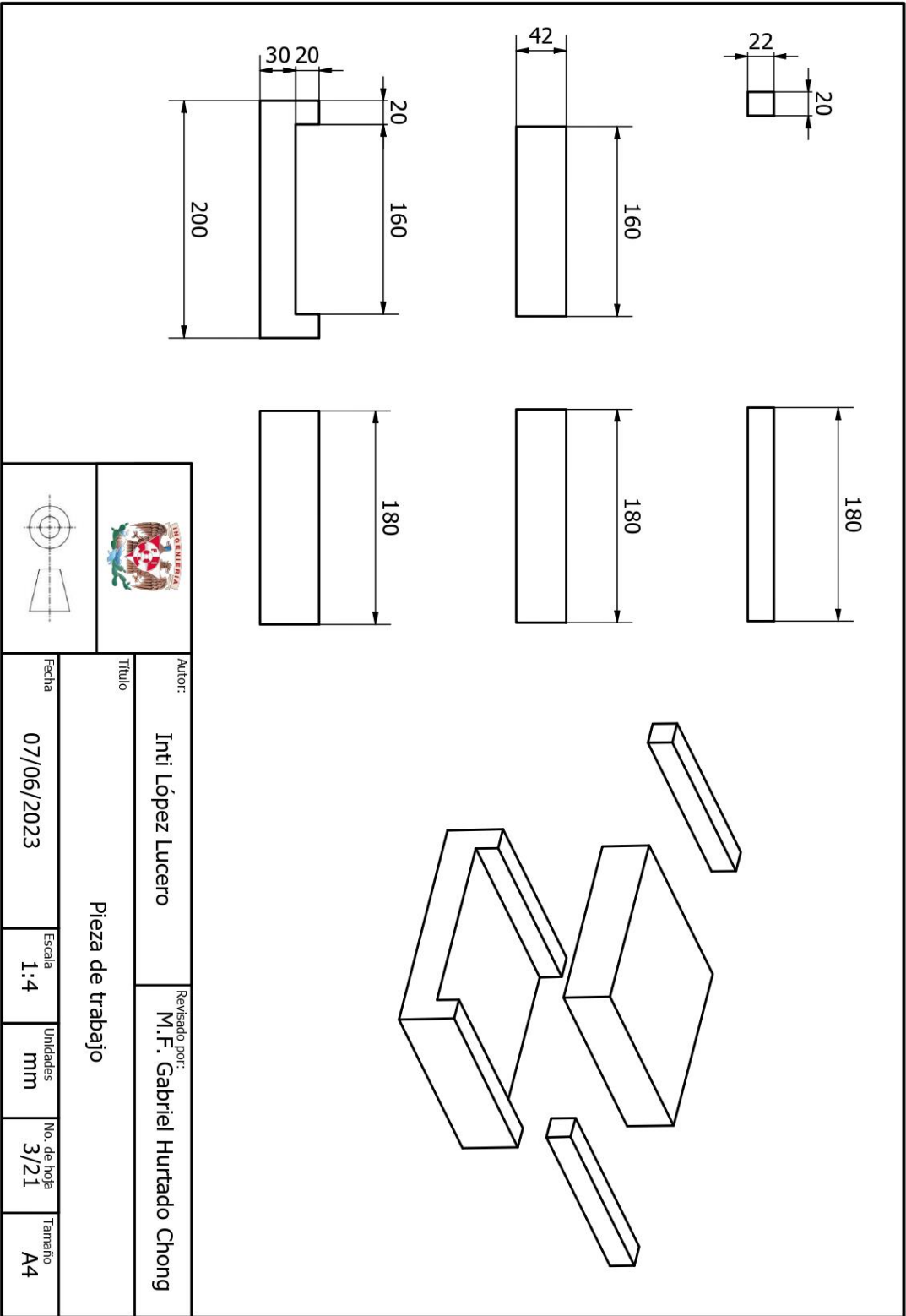
1. Isométrico Ensamble Estampadora
2. Vista Explosionada de la Estampadora
3. Pieza de trabajo
4. Base
5. Soporte Pieza 1
6. Soporte Pieza 2
7. Guía 1
8. Guía 2
9. Cabezal Cilindro 220
10. Cabezal Cilindro 320
11. Cabezal Cilindro 100
12. Soporte Cilindro 100
13. Soporte 2 Cilindro 100
14. Rampa
15. Guía Rampa
16. Soporte Cilindro
17. Mesa
18. Cilindro de Carrera de 220 mm Marca Festo
19. Cilindro de Carrera 320 mm de la Marca Festo
20. Vástago del Cilindro de Carrera de 220 mm Marca Festo
21. Vástago del Cilindro de Carrera 320 mm Marca Festo

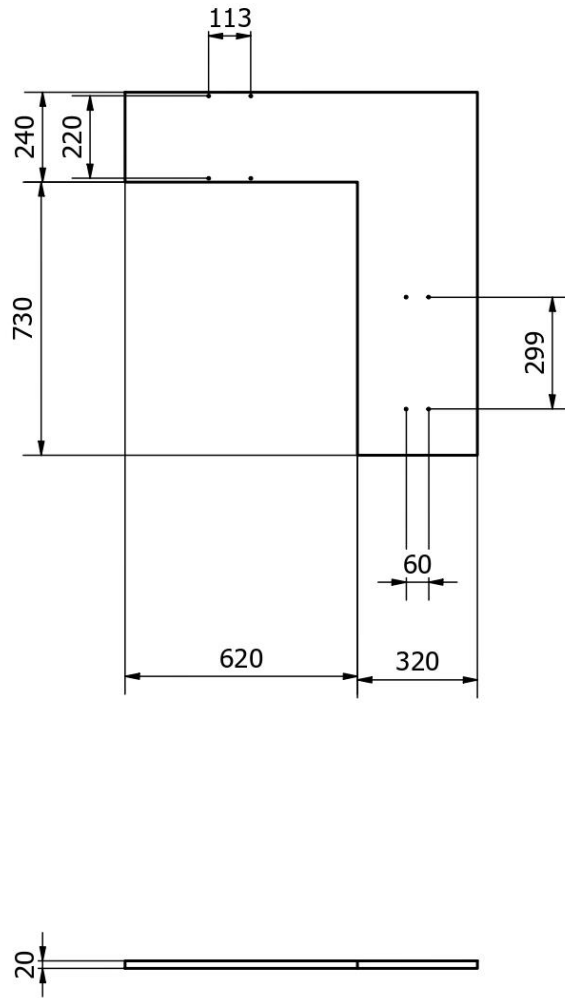
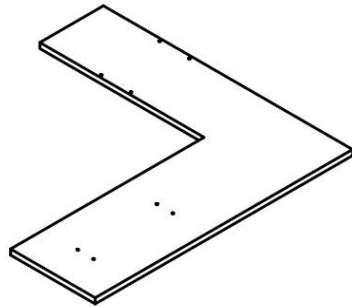



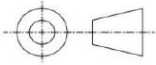
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Isométrico Ensamble Estampadora				
	Fecha: 07/06/2023	Escala: 1:8	Unidades: mm	No. de hoja: 1/21	Tamaño: A4

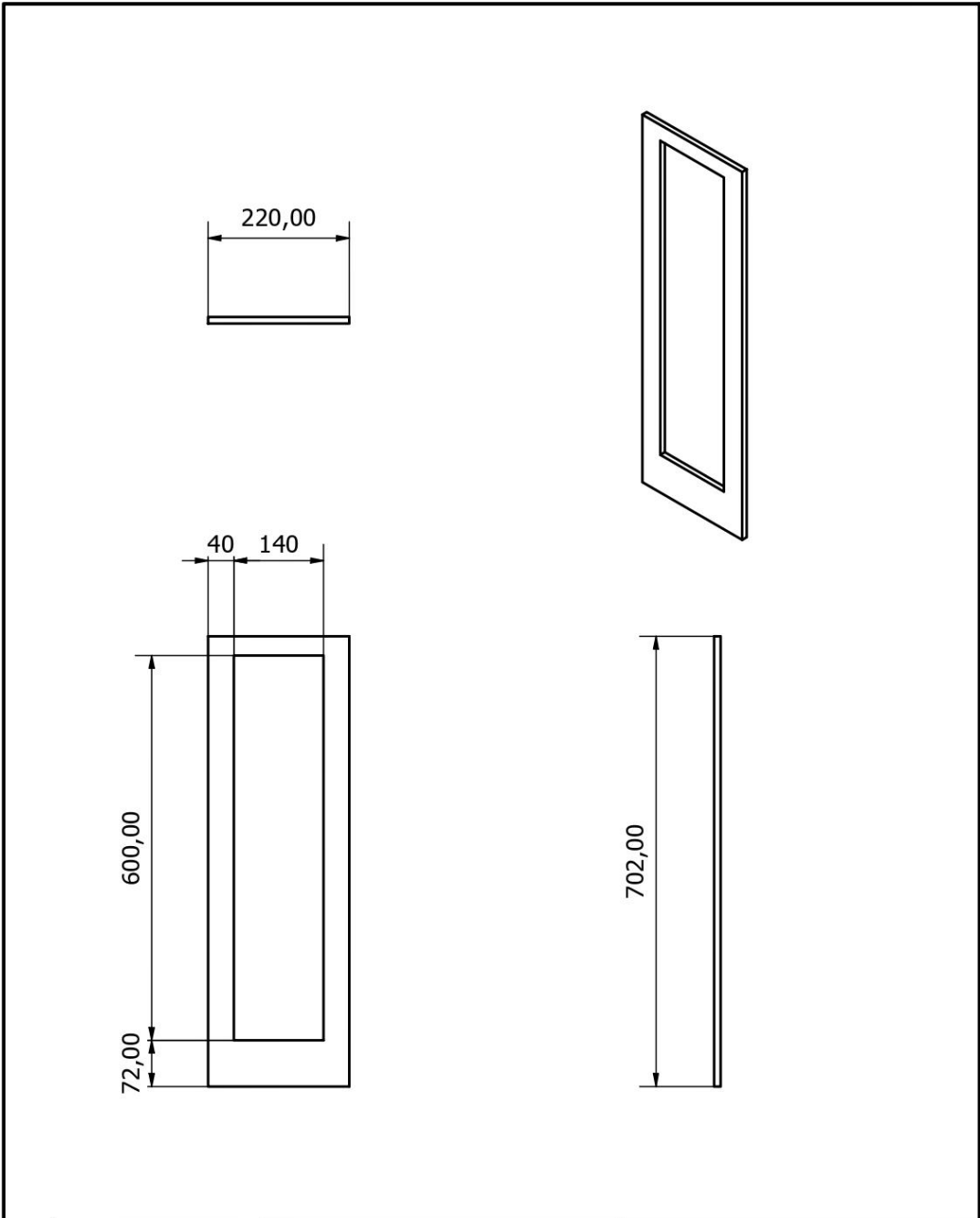
PARTS LIST	
ITEM QTY	PART NUMBER
1	Base
2	Cilindro 320 mm
3	Vástago Cilindro 320
4	Cabezal Cilindro 320
5	Cilindro 220 mm
6	Vástago Cilindro 220
7	Cabezal Cilindro 220
8	Cilindro 100 mm
9	Vástago Cilindro 100
10	Cabezal Cilindro 100
11	Guía 1
12	Guía 2
13	Soporte Pieza 1
14	Soporte Pieza 2
15	Mesa
16	Soporte Cilindro 100
17	Soporte 2 Cilindro 100
18	Rampa
19	Guía Rampa
20	Soporte Cilindro
21	Perno M5
22	Pieza de Trabajo

	Autor: Inti López Lucero	Revisado por: M.F. Gabriel Hurtado Chong
Título: Vista Exploracionada de la Estampadora		
Fecha: 07/06/2023	Escala: 1:20	Unidades: mm
		No. de hoja: 2/21
		Tamaño: A4

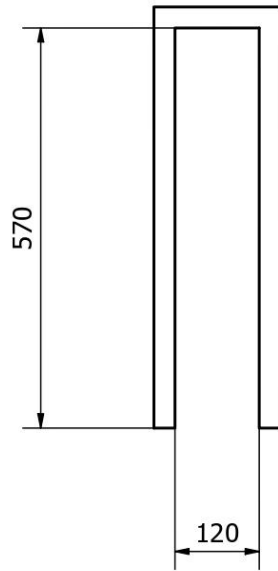
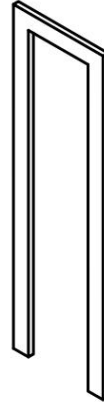
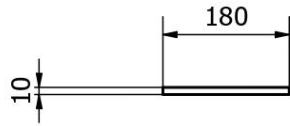




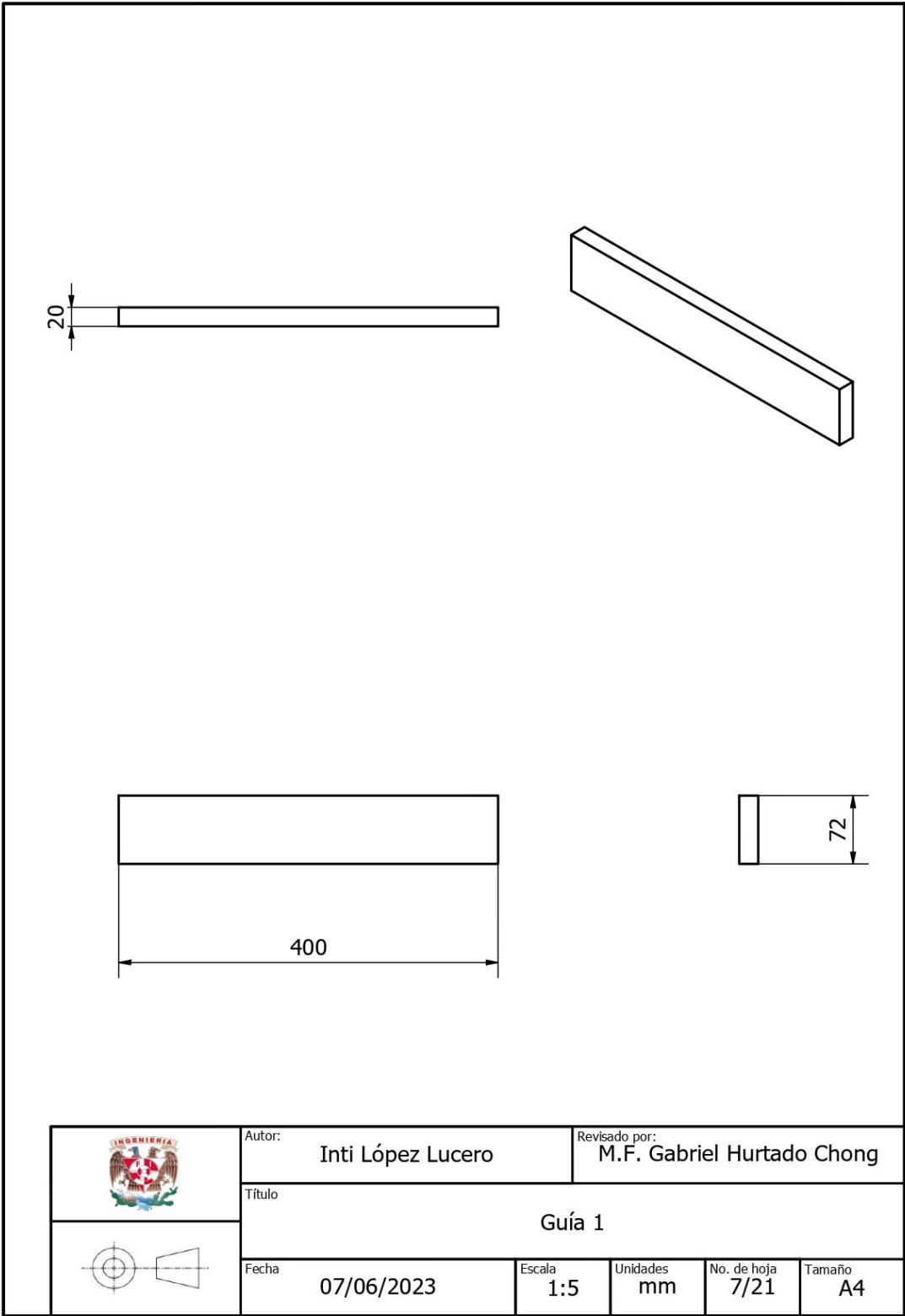
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Base				
	Fecha: 07/06/2023	Escala: 1:15	Unidades: mm	No. de hoja: 4/21	Tamaño: A4



	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Soporte Pieza 1				
	Fecha: 07/06/2023	Escala: 1:8	Unidades: mm	No. de hoja: 5/21	Tamaño: A4



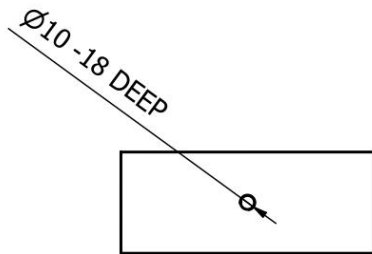
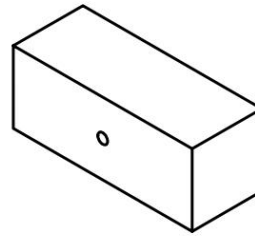
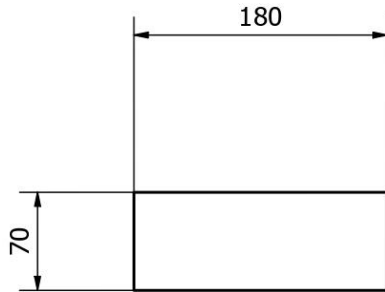
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Soporte Pieza 2				
	Fecha: 07/06/2023	Escala: 1:8	Unidades: mm	No. de hoja: 6/21	Tamaño: A4



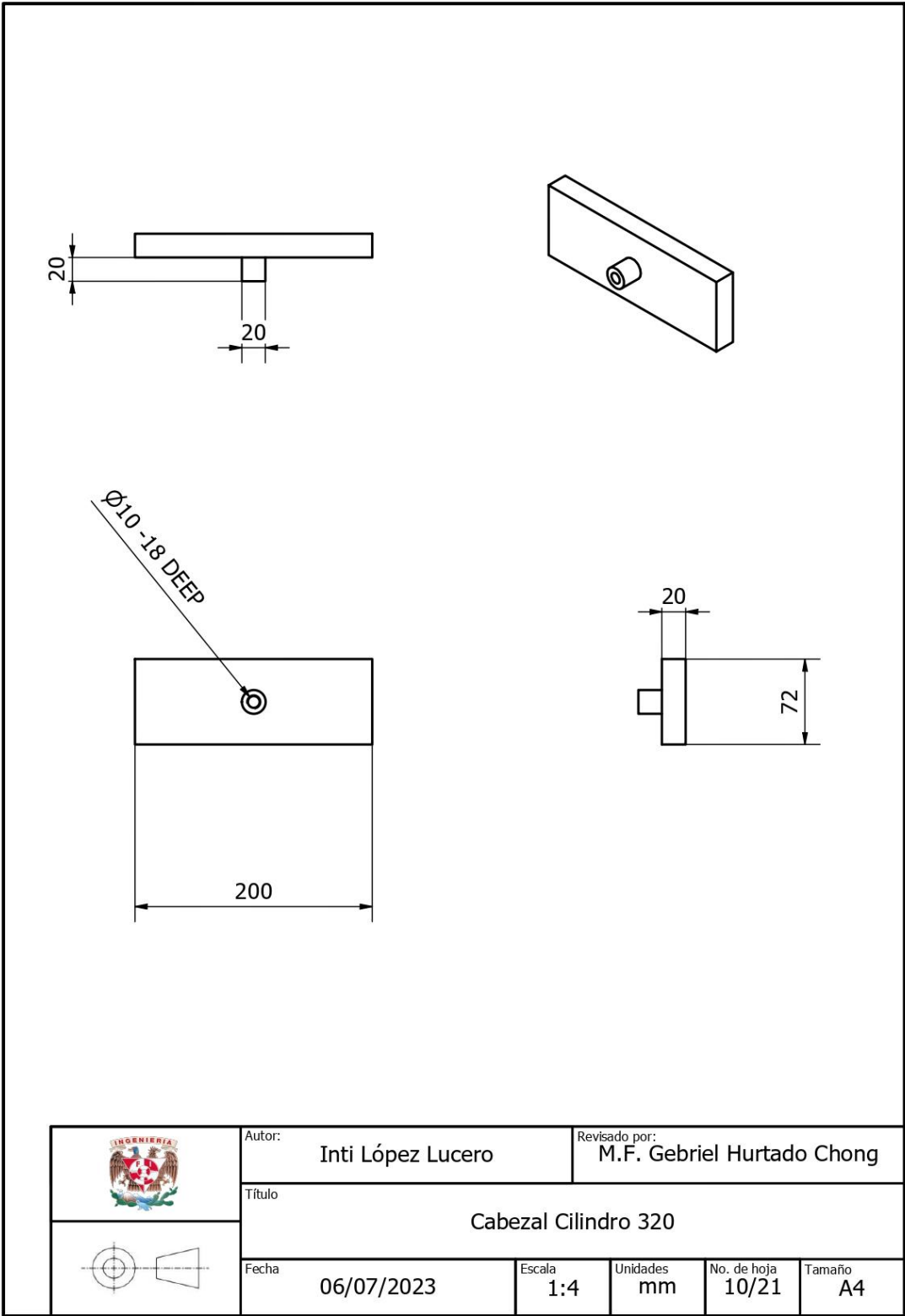
Technical drawing of a rectangular plate with the following dimensions:

- Top view: width 60
- Front view: height 72
- Side view: thickness 10

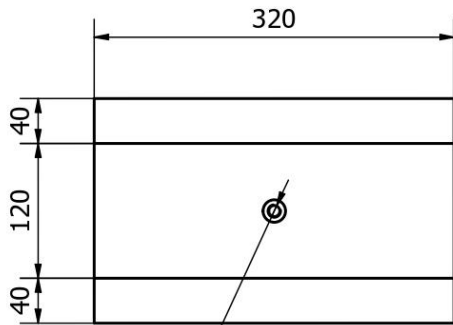
	Autor:	Inti López Lucero	Revisado por:	M.F. Gabriel Hurtado Chong
	Título	Guía 2		
	Fecha	07/06/2023	Escala	1:1
			Unidades	mm
			No. de hoja	8/21
			Tamaño	A4



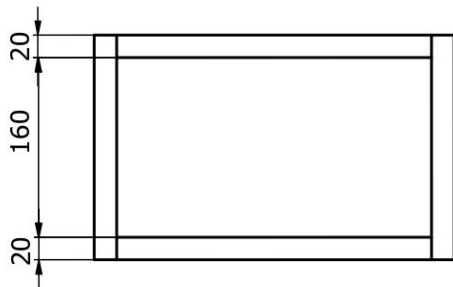
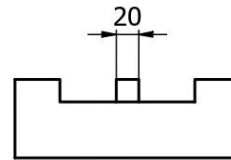
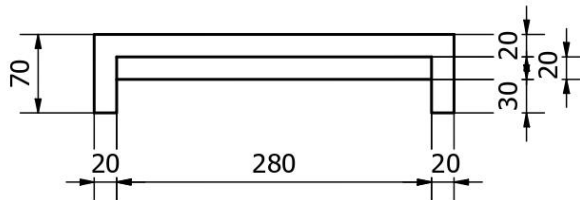
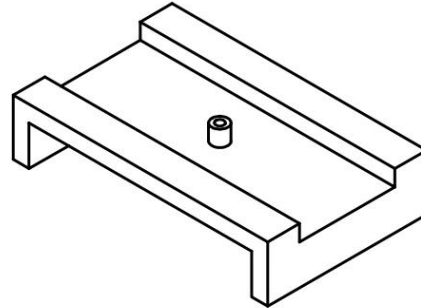
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Cabezal Cilindro 220				
	Fecha: 07/06/2023	Escala: 1:4	Unidades: mm	No. de hoja: 9/21	Tamaño: A4



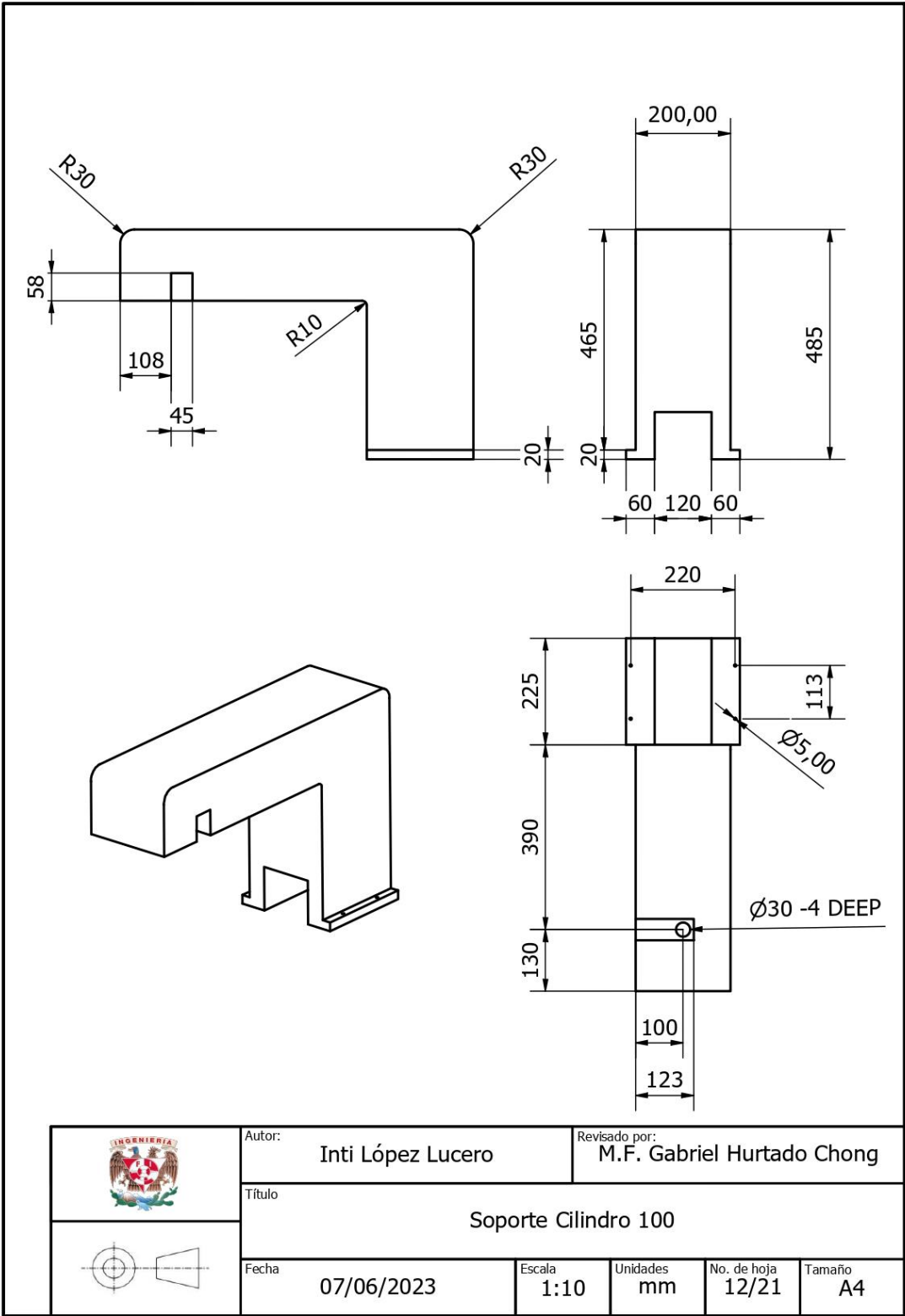
	Autor: Inti López Lucero		Revisado por: M.F. Gebriel Hurtado Chong		
	Título: Cabezal Cilindro 320				
	Fecha: 06/07/2023	Escala: 1:4	Unidades: mm	No. de hoja: 10/21	Tamaño: A4

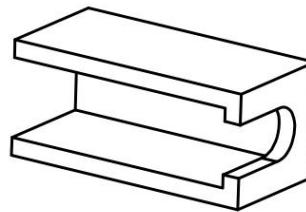
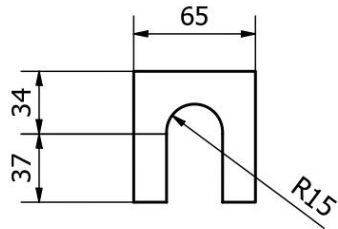
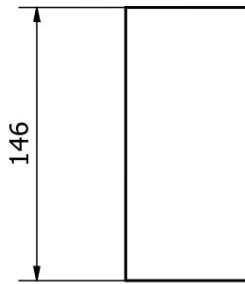
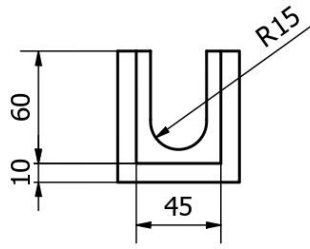



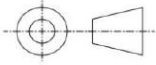
$\varnothing 10 - 18 \text{ DEEP}$

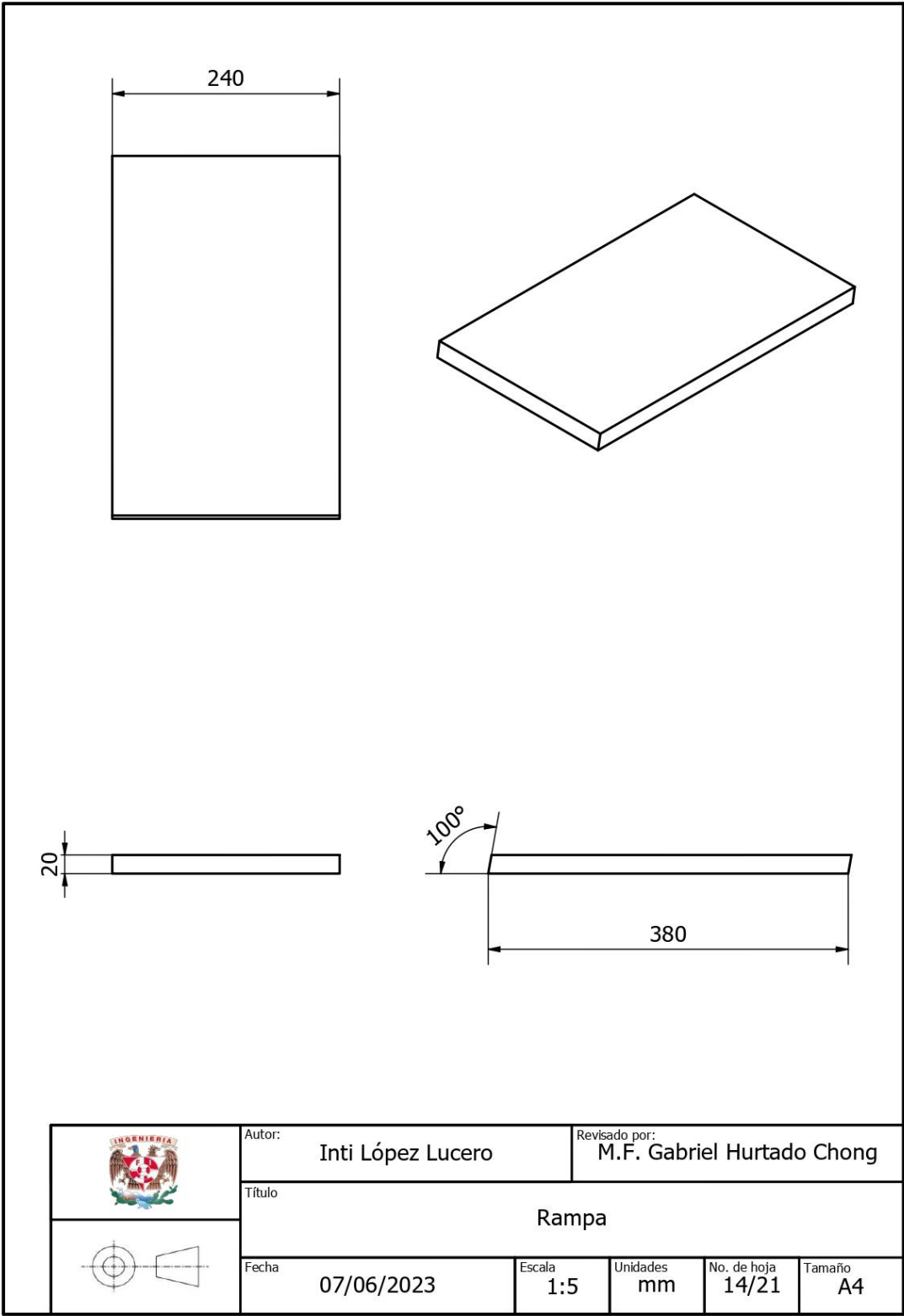


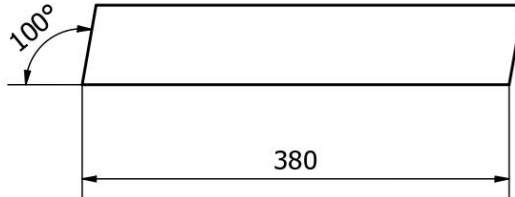
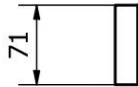
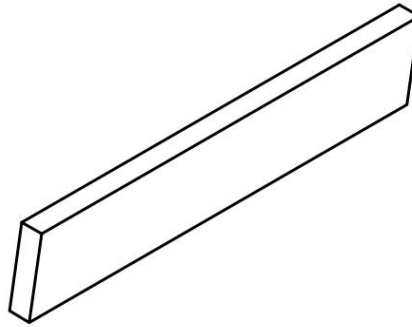
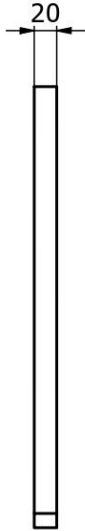
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Cabezal Cilindro 100				
	Fecha: 06/07/2023	Escala: 1:5	Unidades: mm	No. de hoja: 11/21	Tamaño: A4



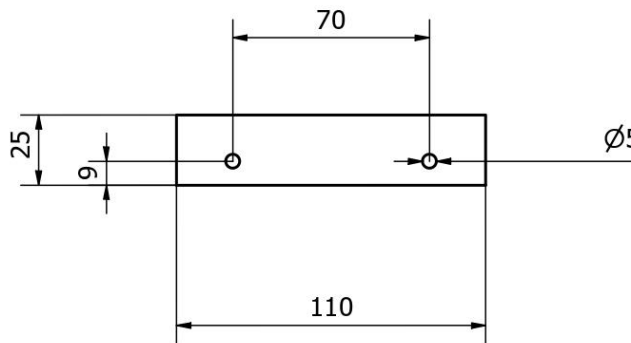
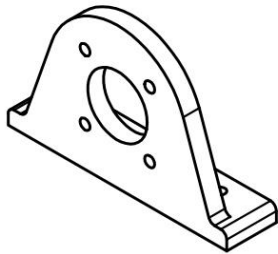
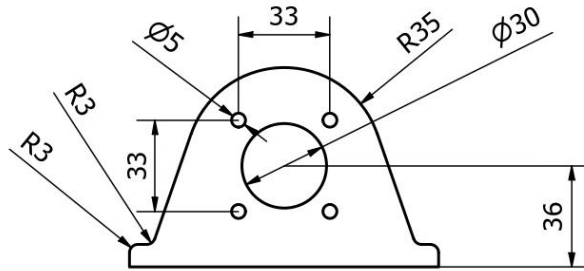
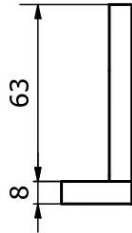


	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Soporte 2 Cilindro 100				
	Fecha: 06/07/2023	Escala: 1:3	Unidades: mm	No. de hoja: 13/21	Tamaño: A4

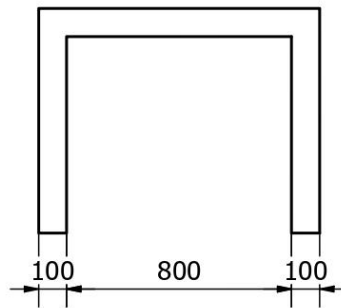
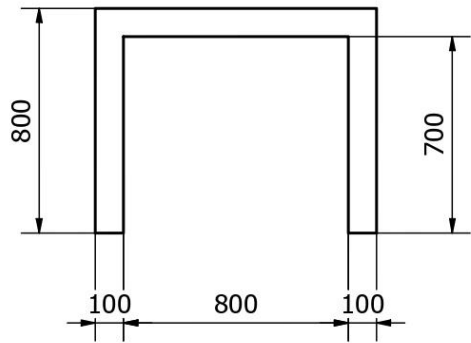
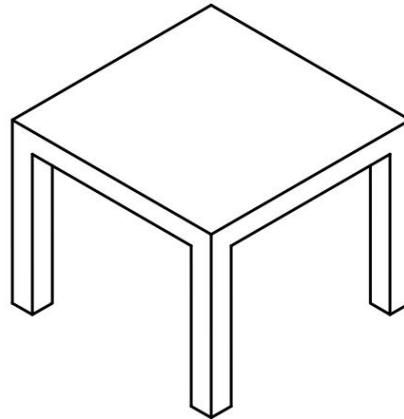
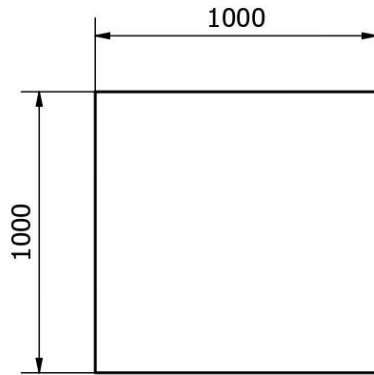




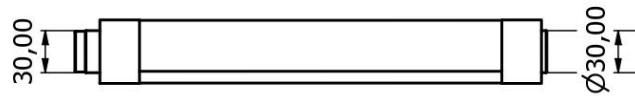
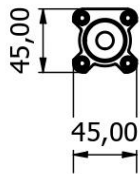
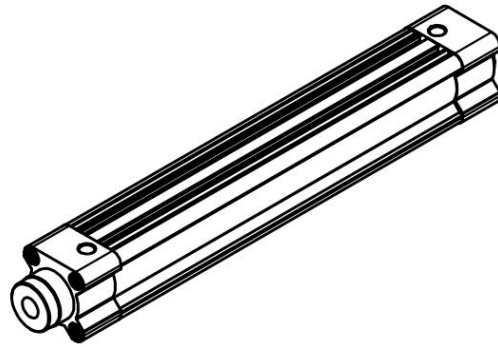
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Guía Rampa				
	Fecha: 07/06/2023	Escala: 1:5	Unidades: mm	No. de hoja: 15/21	Tamaño: A4



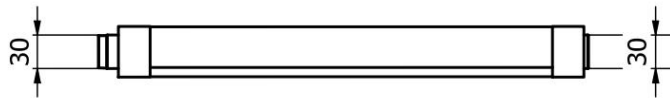
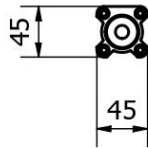
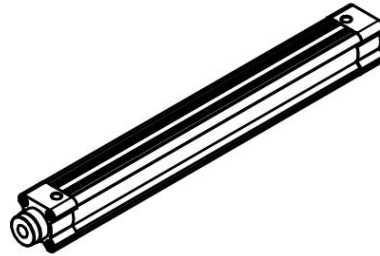
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Soporte Cilindro				
	Fecha: 07/06/2023	Escala: 1:2	Unidades: mm	No. de hoja: 16/21	Tamaño: A4


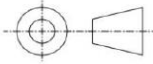


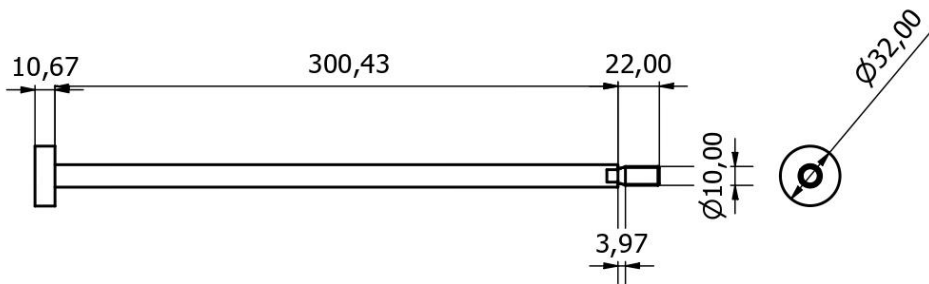
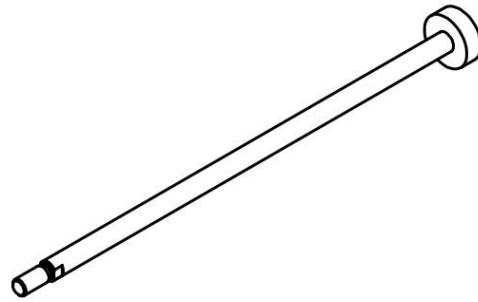
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Mesa				
	Fecha: 07/06/2023	Escala: 1:20	Unidades: mm	No. de hoja: 17/21	Tamaño: A4



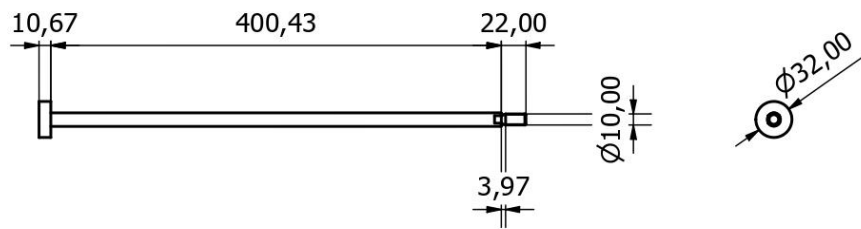
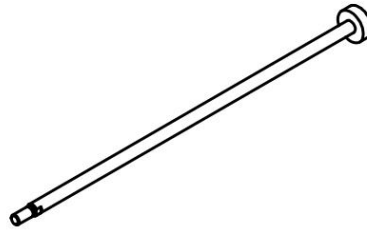
	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Cilindro de Carrera de 220 mm Marca Festo				
	Fecha: 07/06/2023	Escala: 1:3	Unidades: mm	No. de hoja: 18/21	Tamaño: A4



	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Cilindro de Carrera 320 mm de la Marca Festo				
	Fecha: 07/06/2023	Escala: 1:5	Unidades: mm	No. de hoja: 19/21	Tamaño: A4



	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Vástago del Cilindro de Carrera de 220 mm Marca Festo				
	Fecha: 07/06/2023	Escala: 1:3	Unidades: mm	No. de hoja: 20/21	Tamaño: A4



	Autor: Inti López Lucero		Revisado por: M.F. Gabriel Hurtado Chong		
	Título: Vástago del Cilindro de Carrera de 320 mm Marca Festo				
	Fecha: 07/06/2023	Escala: 1:5	Unidades: mm	No. de hoja: 21/21	Tamaño: A4

12. Apéndice D. Imágenes de las Texturas Utilizadas para los Paquetes del Gemelo Digital de la empaadora

1. Textura Paquete Candy Bar
2. Textura Paquete Chispas
3. Textura Paquete Dulzura
4. Textura Paquete FresPop
5. Textura Paquete Pansitas
6. Textura Paquete Papitas
7. Textura Paquete PinkGum
8. Textura Paquete Surtidito
9. Textura Caja Contenedora de Paquetes

BARRAS DE CHOCOLATE
Candy Bar



Candy Bar
BARRAS DE CHOCOLATE

EMPRESAS CHOCOBAR

BARRAS DE CHOCOLATE

Producto obtenido a partir del cacao y la mezcla de otros ingredientes. Las barras de chocolate son elaboradas con granos de cacao selecto con un agradable sabor y aroma. Envasado herméticamente.

dudas y sugerencias comunicarse a los 01 800 Candy Bar



**Industrias
Chocobar**

Proyecto
Este es un proyecto escolar

20

BARRAS DE CHOCOLATE

CHOCOLATEBAR.COM



Reciclando Cuidamos
el planeta

CANDY BAR CUIDA EL PLANETA

MANTENER EN LUGAR FRESCO • CONSUMIR ANTES DE
2020

GALLETAS CON CHISPAS
DE CHOCOLATE

Chispas



EMPRESAS CHISPAS

GALLETAS CON CHISPAS DE CHOCOLATE

Ingredientes: Harina de TRIGO 43,2 %, azúcar, grasa de palma, pasta de cacao, manteca de cacao, gasificantes (carbonatos de amonio, carbonatos de sodio, fosfatos de calcio), dextrosa, jarabe de glucosa y fructosa, sal, suero de LECHE en polvo, emulgentes (lecitinas de SOJA, lecitinas de girasol), aroma.

dudas y sugerencias comunicarse a los 01 869 Candy Bar



**Industrias
Chispas**

Proyecto
Este es un proyecto escolar

10

PAQUETES DE
GALLETAS CON
CHISPAS DE
CHOCOLATE

CHISPAS.COM



Reciclando Cuidamos
el planeta

CHISPAS CUIDA EL PLANETA

MANTENER EN LUGAR FRESCO • CONSUMIR ANTES DE
2025

DULCES DE SABORES

Dulzura



EMPRESAS DULZURA

DULCES DE SABORES

Ingredientes: sacarosa (azúcar de mesa), edulcorante de maíz, jarabe de maíz alto en fructosa, concentrados de jugos de frutas, néctares, azúcar en bruto, jarabe de malta, jarabe de arce, edulcorantes de fructosa, fructosa líquida, miel, melaza.

dulces y sugerencias comunicarse a los 01 863 Candy Bar



**Industrias
Dulzura**

Proyecto
Este es un proyecto escolar

100

DULCES DE SABORES

DULZURA.COM



Reciclando Cuidamos el planeta

DULZURA CUIDA EL PLANETA

MANTENER EN LUGAR FRESCO • CONSUMIR ANTES DE 2023

FresPop



FresPop

Paletas de caramelo sabor fresa. Ingredientes: Azúcares añadidos (azúcar, jarabe de maíz), goma base (resina sintética, carbonato de calcio, cera microcristalina, monogrol, esterina) ácido cítrico, saborizante artificial, colorante artificial ROJO 40).

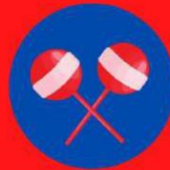


¡Para más información, visita nuestra página web!
www.industriasfrespop.com



50 PALETAS DE CARAMELO
SABOR FRESA

FRESPOP



**Industrias
FresPOP**

Proyecto
Este es un proyecto escolar

INDUSTRIAS FRESPOP

Reciclando Cuidamos el Planeta

Junta Protectora Escolar de Córdoba, Cuidamos el Planeta



GOMITAS DE SABORES

Pansitas



EMPRESAS PANSITAS

GOMITAS DE SABORES

Ingredientes: Jarabe de maíz, azúcar, agua, gellanina, ácido cítrico (regulador de la acidez), saborizantes artificiales (fresa, limón, piña y naranja), cera de carnaúba (agente de glaseado), rojo 40 (rojo añil), amarillo 5 (tartrazina), azul 1 (azul brillante), amarillo 6 (amarillo ocaso).

Dudas y sugerencias comuníquese a los 01 800 Candy Bar



Industrias Pansitas

Proyecto
Este es un proyecto escolar

20

PAQUETES DE GOMITAS DE SABORES

PANSITAS.COM



Reciclando Cuidamos el planeta

PANSITAS CUIDA EL PLANETA

MANTENER EN LUGAR FRESCO • CONSUMIR ANTES DE 2025

PAPAS FRITAS
Papitas



EMPRESAS PAPIAS

PAPAS FRITAS

Ingredientes: Papas, aceite vegetal, sal yodada, azúcar, guindilla (11%), ácido cítrico (E330), ácido acético (E260), glutamato monosódico, proteínas vegetales, aceite vegetal hidrogenado, colorantes artificiales (rojo allura E129, amarillo ocaso FCF E110 trazas de tartrazina).

dudas y sugerencias comunicarse a los 01 860 Candy Bar



**Industrias
Papitas**

Proyecto
Este es un proyecto escolar

5

PAQUETES DE PAPAS
FRITAS

PANITAS.COM



Reciclando Cuidamos
el planeta

PAPITAS CUIDA EL PLANETA

MANTENER EN LUGAR FRESCO • CONSUMIR ANTES DE
2023

INDUSTRIAS GUM

PinkGum



Goma de Mascar Sabor
FRAMBUESA

Ingredientes: Edulcorantes (xilitol *, glucósidos de esteviol), base de goma de chicle, aroma natural de frambuesa y otros aromas naturales, humectante (glicerol), ácido (ácido málico, ácido cítrico), estabilizante (goma arábiga), extracto de boniato, agente de glaseado (cera de carnauba).



**Industrias
PinkGum**

Propósito
Cada es un proyecto exoelr



**Caja con 30 paquetes de
chicles sabor frambuesa**

INDUSTRIAS PINKGUM

**Reciclando
Cuidamos el
Planeta**



Juntos Podemos Hacer un Cambio. Cuidemos el Planeta

SURTIDO DE GOLOSINAS
Surtidito



EMPRESAS PANSITAS

SURTIDO DE GOLOSINAS



Visite nuestra página web para conocer más acerca de nuestros productos

Dudas y sugerencias comunicarse a los 01 800 Candy Bar



0 24563 84925 54 2

Industrias Surtidito

Proyecto
Este es un proyecto escolar

20

PAQUETES DE UN
SURTIDO DE GOLOSINAS

SURTIDITO.COM



Reciclando Cuidamos el planeta

SURTIDITO CUIDA EL PLANETA

MANTENER EN LUGAR FRESCO • CONSUMIR ANTES DE 2023

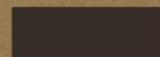
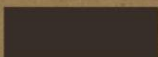
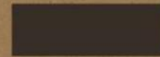
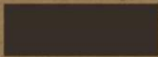
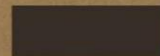
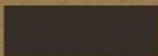
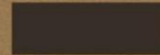
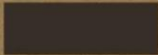
Industrias Caramelo

3 CAJAS CON PAQUETES DE CAMELO

caramelos@industriascaramelo.com • 55-78675955

• Ciudad de México

www.Caramelo.com.mx



13. Apéndice E. Código Original de la Mesas de Trabajo Neumáticas Proporcionado para la implementación de los Gemelos Digitales.

```

1 using UdonSharp;
2 using UnityEngine;
3 using VRC.SDKBase;
4 using VRC.Udon;
5 using TMPPro;
6 using UnityEngine.UI;
7 using System;
8 using VRC.Udon.Common.Interfaces;
9
10
11 public class Man_Valv : UdonSharpBehaviour
12 {
13     //Un arreglo de lineas que actúan como mangueras
14     public LineRenderer[] lr = new LineRenderer[40];
15     //Un arreglo de Objetos que son las conexiones de los elementos
16     public GameObject[] botones = new GameObject[40];
17     // Variables Sincronizadas
18     [UdonSynced]
19     public bool[] isPresed = new bool[40];
20     [UdonSynced]
21     public bool[] isOrigin = new bool[40];
22     [UdonSynced]
23     public bool[] isTarget = new bool[40];
24     [UdonSynced]
25     public bool[] isActive = new bool[40];
26     [UdonSynced]
27     public bool desactivados = false;
28     [UdonSynced]
29     public bool IsActiveGeneral = false;
30     [UdonSynced]
31     public bool fuga = false;
32     [UdonSynced]
33     public bool fuga2 = false;
34     [UdonSynced]
35     public bool fuga3 = false;
36     [UdonSynced]
37     public bool fuga4 = false;
38     [UdonSynced]
39     private float TimerCount;
40     private float Count;
41
42     public int[] Num = new int[40];
43     [UdonSynced]
44     public int indexOriginActive = 80;
45     [UdonSynced]
46     public int Indice;
47     [UdonSynced]
48     public int numarreglo;
49     public Animator animMan, animBoton, animBoton2, animResorte,

```

```

...Single - copia\Assets\Códigos\Tradicional\Man_Valv.cs 2
animVastago, animVastago2, animValv52B, animValv52M, animFinal,
animFinal2, animPiloto;
50 public AudioSource sonidoPiston, sonidoFuga;
51 public UdonBehaviour slider_N;
52 public UdonBehaviour slider_P;
53
54
55 public void Start()
56 {
57     //Inicializamos animaciones para colocarlas en su estado inicial
58     animValv52B.SetBool("anim52Bool", true);
59     animValv52M.SetBool("anim52Bool", true);
60     animPiloto.SetBool("pilotoV2Bool", false);
61
62     //Inicializamos en falso lo arreglos de booleanos para conectar
63     for (int i = 0; i < isPresed.Length; i++)
64     {
65         isPresed[i] = false;
66         isOrigin[i] = false;
67         isActive[i] = false;
68     }
69     //Inicializamos los num en 100 para que no esten conectados con
nada
70     for (int i = 0; i < Num.Length; i++)
71     {
72         Num[i] = 100;
73     }
74     //Inicializamos el tamaño de los linerenderer
75     for (int i = 0; i < lr.Length; i++)
76     {
77         lr[i].positionCount = 2;
78     }
79 }
80
81 private void Update()
82 {
83     Debug.Log("El Botón esta conectado: " + Num[10]);
84
85     Debug.Log("Valor de desactivados: " + desactivados);
86     Debug.Log("Valor de IsActiveGeneral: " + IsActiveGeneral);
87     Debug.Log("Valor de fuga: " + fuga);
88     Debug.Log("Valor de fuga2: " + fuga2);
89     Debug.Log("Valor de fuga3: " + fuga3);
90     Debug.Log("Valor de fuga4: " + fuga4);
91
92     //Si NO hay presión de salida en la unidad de mantenimiento
93     if (animMan.GetBool("ManBool") == false)
94     {
95         //Regresamos a su estado inicial al vastago

```

```

...Single - copia\Assets\Códigos\Tradicional\Man_Valv.cs 3
96     if (animResorte.GetBool("ResorteAnimBool") == true)
97     {
98         animVastago.SetBool("VastagoAnimBool", false);
99         animResorte.SetBool("ResorteAnimBool", false);
100    }
101
102    if (desactivados == false)
103    {
104        //Apaga el sonido, Ya no hay fuga, Apagas la unidad de  ↗
105        //mantenimiento y el piloto
106        sonidoFuga.Stop();
107        IsActiveGeneral = false;
108        botones[10].GetComponent<MeshRenderer>().material.color = ↗
109        Color.white;
110        animPiloto.SetBool("pilotoV2Bool", false);
111
112        //Para todos los botones
113        for (int i = 0; i < this.botones.Length; i++)
114        {
115            //Todo boton con conexión (Num!=100) lo hace color  ↗
116            Cyan
117            if (Num[i] != 100)
118            {
119                botones[i].GetComponent<MeshRenderer>  ↗
120                ().material.color = Color.cyan;
121                lr[Num[i]].material.color = Color.cyan;
122            }
123            //Todo boton sin conexión (Num=100) lo hace color  ↗
124            blanco
125            else
126            {
127                botones[i].GetComponent<MeshRenderer>  ↗
128                ().material.color = Color.white;
129            }
130            //No desactivamos el Active del piston de doble  ↗
131            efecto
132            if (i != 8)
133                isActive[i] = false;
134        }
135
136        //Habilitamos los collider de las conexiones
137        foreach (GameObject conexion in botones)
138        {
139            conexion.GetComponent<BoxCollider>().enabled = true;
140        }
141
142        //Para que solo haga esto una vez
143        desactivados = true;

```



```
138     }
139
140     //Pregunta continuamente a cada boton si ha sido presionado
141     for (int i = 0; i < this.botones.Length; i++)
142     {
143         if (isPresed[i] && !isOrigin[i] && !isTarget[i])
144         {
145             //Activar origen
146             if (indexOriginActive == 80)
147             {
148                 isOrigin[i] = true;
149                 botones[i].GetComponent<MeshRenderrer>
150                 ().material.color = Color.yellow;
151                 indexOriginActive = i;
152                 Num[i] = i;
153                 isPresed[i] = false;
154             }
155             //Activar target y realiza la conexión
156             else
157             {
158                 isTarget[i] = true;
159                 botones[i].GetComponent<MeshRenderrer>
160                 ().material.color = Color.cyan;
161                 lr[indexOriginActive].enabled = true;
162                 lr[indexOriginActive].SetPosition(0, botones
163                 [indexOriginActive].transform.position);
164                 lr[indexOriginActive].SetPosition(1, botones
165                 [i].transform.position);
166                 botones
167                 [indexOriginActive].GetComponent<MeshRenderrer>
168                 ().material.color = Color.cyan;
169                 Num[i] = indexOriginActive;
170                 indexOriginActive = 80;
171                 isPresed[i] = false;
172             }
173         }
174     }
175     //Desactivar
176     else if ((isOrigin[i] && isPresed[i] || isTarget[i] &&
177     isPresed[i]))
178     {
179         //Caso una sola conexión
180         if (botones[i].GetComponent<MeshRenderrer>
181         ().material.color == Color.yellow)
182         {
183             numarreglo = Num[i];
184             for (int j = 0; j < Num.Length; j++)
185             {
186                 if (Num[j] == numarreglo)
```

```

...Single - copia\Assets\Códigos\Tradicional\Man_Valv.cs 5
179         {
180             isOrigin[j] = false;
181             isTarget[j] = false;
182             isPresed[j] = false;
183             botones[j].GetComponent<MeshRenderer>
() .material.color = Color.white;
184             lr[j].enabled = false;
185             Num[j] = 100;
186
187         }
188     }
189     indexOriginActive = 80;
190 }
191
192 //Caso desconectar dos azules
193 else if (botones[i].GetComponent<MeshRenderer>
() .material.color == Color.cyan && indexOriginActive ==
80)
194 {
195     numarreglo = Num[i];
196     for (int j = 0; j < Num.Length; j++)
197     {
198         if (Num[j] == numarreglo)
199         {
200             isOrigin[j] = false;
201             isTarget[j] = false;
202             isPresed[j] = false;
203             botones[j].GetComponent<MeshRenderer>
() .material.color = Color.white;
204             lr[j].enabled = false;
205             Num[j] = 100;
206         }
207     }
208     indexOriginActive = 80;
209 }
210
211 //Caso desconectar un amarillo cuando se trata de
conectar con un azul
212 else if (botones[i].GetComponent<MeshRenderer>
() .material.color == Color.cyan && indexOriginActive !=
80)
213 {
214     isPresed[i] = false;
215     isOrigin[indexOriginActive] = false;
216     botones
[indexOriginActive].GetComponent<MeshRenderer>
() .material.color = Color.white;
217     Num[indexOriginActive] = 100;
218     indexOriginActive = 80;

```

```
219         }
220     }
221 }
222 }
223
224 //si hay presión de salida en la unidad de mantenimiento
225 else
226 {
227     //Desactiva las uniones para poder dar paso al aire
228     if (desactivados == true)
229     {
230         foreach (GameObject conexion in botones)
231         {
232             conexion.GetComponent<BoxCollider>().enabled = false;
233         }
234         desactivados = false;
235     }
236
237     //Activación de la unidad de mantenimiento
238     NoHayFuga(sonidoFuga, botones);
239     ActivacionUM(10, Num, isActive, botones);
240
241     //Botones
242     Botonsito(22, 23, "Boton1", animBoton, Num, isActive, botones);
243     Botonsito(20, 21, "Boton1", animBoton2, Num, isActive, botones);
244
245     //FINALES DE CARRERA
246     FinalCarrera(24, 25, "finalBool", animFinal, Num, isActive, botones);
247     FinalCarrera(26, 27, "finalBool", animFinal2, Num, isActive, botones);
248
249     //VALVULAS
250     Valvula52Biestable(19, 16, 17, 18, 15, animValv52B, "val52Bool", Num, isActive, botones);
251     Valvula52Monoestable(14, 11, 12, 13, animValv52M, "val52Bool", Num, isActive, botones);
252
253     //CONECTAR T
254     ConectarT(00, 01, 02, Num, isActive, botones);
255     ConectarT(03, 04, 05, Num, isActive, botones);
256     ConectarT(06, 07, 30, Num, isActive, botones);
257     ConectarT(31, 32, 33, Num, isActive, botones);
258
259     //PISTONES
260     ActivarPistonSimple(28, animVastago, "VastagoAnimBool", animResorte, "ResorteAnimBool", sonidoPiston, isActive,
```

```

...Single - copia\Assets\Códigos\Tradicional\Man_Valv.cs 7
    botones);
261     ActivarPistonDoble(8, 9, animVastago2, "VastDobAnimBool",
    sonidoPiston, isActive, botones);
262
263     //PPILOTO
264     ActivacionPiloto(29, isActive, botones);
265
266     //Timers Prueba
267     TimerNegativo(34, 35, 36, Num, isActive, botones);
268     TimerPositivo(37, 38, 39, Num, isActive, botones);
269
270 }
271
272 //Permite mover las mangueras en tiempo real de las uniones
273 ActualizarLineRenderer(00, 01, 02, Num, botones);
274 ActualizarLineRenderer(03, 04, 05, Num, botones);
275 ActualizarLineRenderer(06, 07, 30, Num, botones);
276 ActualizarLineRenderer(31, 32, 33, Num, botones);
277 }
278
279 public void ActivarPistonSimple(int Piston, Animator animVastago,
    string VastagoAnimBool, Animator animResorte, string
    ResorteAnimBool, AudioSource sonidoPiston, bool[] isActive,
    GameObject[] botones)
280 {
281     //Si NO hay fuga
282     if (IsActiveGeneral == false)
283     {
284         //Si es necesario que el piston se acciones
285         if (botones[Piston].GetComponent<MeshRenderer>
            ().material.color == Color.blue && isActive[Piston] ==
            false)
286         {
287             animVastago.SetBool(VastagoAnimBool, true);
288             animResorte.SetBool(ResorteAnimBool, true);
289             isActive[Piston] = true;
290             sonidoPiston.Play();
291         }
292         //si es necesario el piston regrese a su posicion inicial
293         else if (botones[Piston].GetComponent<MeshRenderer>
            ().material.color == Color.white && isActive[Piston] ==
            true)
294         {
295             animVastago.SetBool(VastagoAnimBool, false);
296             animResorte.SetBool(ResorteAnimBool, false);
297             isActive[Piston] = false;
298             sonidoPiston.Play();
299         }
300     }

```

```

...Single - copia\Assets\Códigos\Tradicional\Man_Valv.cs 8
301     //Si hay fuga
302     else
303     {
304         //Si el piston esta accionado lo regresa a su posicion inicial
305         if (animVastago.GetBool(VastagoAnimBool) == true)
306         {
307             isActive[Piston] = false;
308             animVastago.SetBool(VastagoAnimBool, false);
309             animResorte.SetBool(ResorteAnimBool, false);
310         }
311     }
312 }
313
314 public void ActivarPistonDoble(int Entrada, int Salida, Animator animVastago, string VastagoAnimBool, AudioSource sonidoPiston, bool [] isActive, GameObject[] botones)
315 {
316     //Si NO hay guga
317     if (IsActiveGeneral == false)
318     {
319         //Si es necesario pasar a la posición de trabajo
320         if (botones[Entrada].GetComponent<MeshRenderer>().material.color == Color.blue &&
321             botones[Salida].GetComponent<MeshRenderer>().material.color != Color.blue && isActive[Entrada] == false)
322         {
323             animVastago.SetBool(VastagoAnimBool, true);
324             sonidoPiston.Play();
325             isActive[Entrada] = true;
326         }
327
328         //Si es necesario pasar a la posición de reposo
329         if (botones[Salida].GetComponent<MeshRenderer>().material.color == Color.blue &&
330             botones[Entrada].GetComponent<MeshRenderer>().material.color != Color.blue && isActive[Entrada] == true)
331         {
332             animVastago.SetBool(VastagoAnimBool, false);
333             sonidoPiston.Play();
334             isActive[Entrada] = false;
335         }
336     }
337 }
338
339 public void Valvula52Monoestable(int alimentacion, int botIzq, int

```

```

...Single - copia\Assets\Códigos\Tradicional\Man_Valv.cs 9
camIzq, int camDer, Animator anim52Bool, string val52Bool, int[]  ↗
Num, bool[] isActive, GameObject[] botones)
341 {
342     //Comprueba si ya no hay fuga
343     NoHayFuga(sonidoFuga, botones);
344     //Si NO hay fuga
345     if (IsActiveGeneral == false)
346     {
347         //Si se esta alimentando a la valvula
348         if (botones[alimentacion].GetComponent<MeshRenderere>  ↗
349             ().material.color == Color.blue)
350         {
351             fuga = true;
352             // Si el pilotaje izquierdo esta accionado
353             if (botones[botIzq].GetComponent<MeshRenderere>  ↗
354                 ().material.color == Color.blue)
355                 anim52Bool.SetBool(val52Bool, false);
356             // Si el pilotaje izquierdo NO esta accionado
357             else
358                 anim52Bool.SetBool(val52Bool, true);
359
360             // Si esta activada la camara derecha
361             if (anim52Bool.GetBool(val52Bool) == true)
362             {
363                 isActive[camIzq] = false;
364                 //Si aun no ha dejado parar el aire a travez de la  ↗
365                 camara derecha
366                 if (isActive[camDer] == false)
367                 {
368                     botones[camDer].GetComponent<MeshRenderere>  ↗
369                     ().material.color = Color.blue;
370
371                     //Si tiene algo conectado la camara derecha para  ↗
372                     dejar pasar la presión de aire
373                     Indice = ConQuien(camDer, Num);
374                     if (Indice != 100)
375                     {
376                         botones[Indice].GetComponent<MeshRenderere>  ↗
377                         ().material.color = Color.blue;
378                         lr[Num[Indice]].material.color = Color.blue;
379                     }
380                     //Si hay fuga
381                     else
382                     {
383                         botones[camDer].GetComponent<MeshRenderere>  ↗
384                         ().material.color = Color.red;
385                         HayFuga(sonidoFuga);
386                     }
387                 }
388             }
389         }
390     }

```

```

...Single - copia\Assets\Códigos\Tradicional\Man_Valv.cs 10
381         botones[camIzq].GetComponent<MeshRender>  ↗
           ().material.color = Color.white;
382         //Si tiene algo conectado la camara izquierda  ↗
           para no dejar pasar la presión del aire
383         Indice = ConQuien(camIzq, Num);
384         if (Indice != 100)
385         {
386             botones[Indice].GetComponent<MeshRender>  ↗
           ().material.color = Color.white;
387             lr[Num[Indice]].material.color = Color.cyan;
388         }
389         isActive[camDer] = true;
390     }
391 }
392 //Si esta activada la camara izquierda
393 else
394 {
395     isActive[camDer] = false;
396     //Si aun no ha dejado parar el aire a travez de la  ↗
           camara izquierda
397     if (isActive[camIzq] == false)
398     {
399         botones[camIzq].GetComponent<MeshRender>  ↗
           ().material.color = Color.blue;
400
401         //Si tiene algo conectado la camara izquierda  ↗
           para dejar pasar la presión de aire
402         Indice = ConQuien(camIzq, Num);
403         if (Indice != 100)
404         {
405             botones[Indice].GetComponent<MeshRender>  ↗
           ().material.color = Color.blue;
406             lr[Num[Indice]].material.color = Color.blue;
407         }
408         //Si hay fuga
409         else
410         {
411             botones[camIzq].GetComponent<MeshRender>  ↗
           ().material.color = Color.red;
412             HayFuga(sonidoFuga);
413         }
414
415         botones[camDer].GetComponent<MeshRender>  ↗
           ().material.color = Color.white;
416
417         //Si tiene algo conectado la camara derecha para  ↗
           NO dejar pasar la presión de aire
418         Indice = ConQuien(camDer, Num);
419         if (Indice != 100)

```

```

...Single - copia\Assets\Códigos\Tradicional\Man_Valv.cs 11
420         {
421             botones[Indice].GetComponent<MeshRender>  ➤
422             ().material.color = Color.white;
423             lr[Num[Indice]].material.color = Color.cyan;
424         }
425         isActive[camIzq] = true;
426     }
427 }
428 }
429
430     isActive[alimentacion] = true;
431 }
432 //Si no se esta alimentando la valvula
433 else
434 {
435     //Si viene de un estado activad
436     if (isActive[alimentacion] == true)
437     {
438         botones[camIzq].GetComponent<MeshRender>  ➤
439         ().material.color = Color.white;
440         botones[camDer].GetComponent<MeshRender>  ➤
441         ().material.color = Color.white;
442         isActive[camIzq] = false;
443         isActive[camDer] = false;
444         anim52Bool.SetBool(val52Bool, false);
445         //Si la camara izquierda tiene algo conectado para NO  ➤
446         dejar pasar la presión de aire
447         Indice = ConQuien(camIzq, Num);
448         if (Indice != 100)
449         {
450             botones[Indice].GetComponent<MeshRender>  ➤
451             ().material.color = Color.white;
452             lr[Num[Indice]].material.color = Color.cyan;
453         }
454         //Si la camara derecha tiene algo conectado para NO  ➤
455         dejar pasar la presión de aire
456         Indice = ConQuien(camDer, Num);
457         if (Indice != 100)
458         {
459             botones[Indice].GetComponent<MeshRender>  ➤
460             ().material.color = Color.white;
461             lr[Num[Indice]].material.color = Color.cyan;
462         }
463         isActive[alimentacion] = false;
464     }
465 }
466 }
467 //Si hay fuga

```



```
462     else
463     {
464         isActive[camIzq] = false;
465         isActive[camDer] = false;
466         isActive[alimentacion] = false;
467         anim52Bool.SetBool(val52Bool, false);
468     }
469
470     //Si hay fuga pero previamente estaba siendo alimentada
471     if (botones[alimentacion].GetComponent<MeshRenderer>
472         ().material.color == Color.white && fuga == true)
473     {
474         botones[camIzq].GetComponent<MeshRenderer>().material.color =
475             Color.white;
476         botones[camDer].GetComponent<MeshRenderer>().material.color =
477             Color.white;
478         isActive[camIzq] = false;
479         isActive[camDer] = false;
480         //Si la camara izquierda tiene algo conectado para NO dejar
481         //pasar la presión de aire
482         Indice = ConQuien(camIzq, Num);
483         if (Indice != 100)
484         {
485             botones[Indice].GetComponent<MeshRenderer>
486                 ().material.color = Color.white;
487             lr[Num[Indice]].material.color = Color.cyan;
488         }
489         //Si la camara derecha tiene algo conectado para NO dejar
490         //pasar la presión de aire
491         Indice = ConQuien(camDer, Num);
492         if (Indice != 100)
493         {
494             botones[Indice].GetComponent<MeshRenderer>
495                 ().material.color = Color.white;
496             lr[Num[Indice]].material.color = Color.cyan;
497         }
498         fuga = false;
499     }
500 }
501
502 public void Valvula52Biestable(int alimentacion, int botIzq, int
503     camIzq, int camDer, int botDer, Animator anim52Bool, string
504     val52Bool, int[] Num, bool[] isActive, GameObject[] botones)
505 {
506     //Comprueba si ya no hay fuga
507     NoHayFuga(sonidoFuga, botones);
508     //Si NO hay fuga
509     if (IsActiveGeneral == false)
```

```

...Single - copia\Assets\Códigos\Tradicional\Man_Valv.cs 13
502     {
503         //Si se esta alimentando la valvula
504         if (botones[alimentacion].GetComponent<MeshRenderere>  ↗
505             ().material.color == Color.blue)
506         {
507             fuga2 = true;
508             //Si el pilotaje izquierdo esta activado y el derecho NO  ↗
509             if (botones[botIzq].GetComponent<MeshRenderere>  ↗
510                 ().material.color == Color.blue &&
511                 botones[botDer].GetComponent<MeshRenderere>  ↗
512                 ().material.color != Color.blue)
513             {
514                 anim52Bool.SetBool(val52Bool, false);
515             }
516
517             //Si el pilotaje derecho esta activado y el izquierdo NO  ↗
518             if (botones[botDer].GetComponent<MeshRenderere>  ↗
519                 ().material.color == Color.blue &&
520                 botones[botIzq].GetComponent<MeshRenderere>  ↗
521                 ().material.color != Color.blue)
522             {
523                 anim52Bool.SetBool(val52Bool, true);
524             }
525
526             //Si esta activa la camara derecha
527             if (anim52Bool.GetBool(val52Bool) == true)
528             {
529                 isActive[camIzq] = false;
530                 //Si aún no ha dejado pasar la presión la camara
531                 if (isActive[camDer] == false)
532                 {
533                     botones[camDer].GetComponent<MeshRenderere>  ↗
534                     ().material.color = Color.blue;
535
536                     //Si tiene algo conectado la camara derecha para  ↗
537                     dejar pasar la presión de aire
538                     Indice = ConQuien(camDer, Num);
539                     if (Indice != 100)
540                     {
541                         botones[Indice].GetComponent<MeshRenderere>  ↗
542                         ().material.color = Color.blue;
543                         lr[Num[Indice]].material.color = Color.blue;
544                     }
545                     //Si hay fuga
546                     else
547                     {
548                         botones[camDer].GetComponent<MeshRenderere>  ↗
549                         ().material.color = Color.red;
550                         HayFuga(sonidoFuga);
551                     }
552                 }
553             }
554         }
555     }

```

```
542     }
543
544     //Si tiene algo conectado la camara izquierda para NO dejar pasar la presión de aire
545     botones[camIzq].GetComponent<MeshRenderer>().material.color = Color.white;
546     Indice = ConQuien(camIzq, Num);
547     if (Indice != 100)
548     {
549         botones[Indice].GetComponent<MeshRenderer>().material.color = Color.white;
550         lr[Num[Indice]].material.color = Color.cyan;
551     }
552     isActive[camDer] = true;
553 }
554 }
555 //Si esta activada la camara izquierda
556 else
557 {
558     isActive[camDer] = false;
559     //Si aún no ha dejado pasar la presión la camara izquierda
560     if (isActive[camIzq] == false)
561     {
562         botones[camIzq].GetComponent<MeshRenderer>().material.color = Color.blue;
563
564         //Si tiene algo conectado la camara izquierda para dejar pasar la presión de aire
565         Indice = ConQuien(camIzq, Num);
566         if (Indice != 100)
567         {
568             botones[Indice].GetComponent<MeshRenderer>().material.color = Color.blue;
569             lr[Num[Indice]].material.color = Color.blue;
570         }
571         //Si hay fuga
572         else
573         {
574             botones[camIzq].GetComponent<MeshRenderer>().material.color = Color.red;
575             HayFuga(sonidoFuga);
576         }
577
578         botones[camDer].GetComponent<MeshRenderer>().material.color = Color.white;
579
580         Indice = ConQuien(camDer, Num);
581         if (Indice != 100)
```

```
582         {
583             botones[Indice].GetComponent<MeshRenderer>
584             ().material.color = Color.white;
585             lr[Num[Indice]].material.color = Color.cyan;
586         }
587         isActive[camIzq] = true;
588     }
589 }
590 }
591
592     isActive[alimentacion] = true;
593 }
594 //Si NO se esta alimentando la valvula
595 else
596 {
597     //Si viene de un estado anterior alimentado
598     if (isActive[alimentacion] == true)
599     {
600         botones[camIzq].GetComponent<MeshRenderer>
601         ().material.color = Color.white;
602         botones[camDer].GetComponent<MeshRenderer>
603         ().material.color = Color.white;
604         isActive[camIzq] = false;
605         isActive[camDer] = false;
606
607         //Si tiene algo conectado la camara izquierda para NO
608         //dejar pasar la presión de aire
609         Indice = ConQuien(camIzq, Num);
610         if (Indice != 100)
611         {
612             botones[Indice].GetComponent<MeshRenderer>
613             ().material.color = Color.white;
614             lr[Num[Indice]].material.color = Color.cyan;
615         }
616         //Si tiene algo conectado la camara derecha para NO
617         //dejar pasar la presión de aire
618         Indice = ConQuien(camDer, Num);
619         if (Indice != 100)
620         {
621             botones[Indice].GetComponent<MeshRenderer>
622             ().material.color = Color.white;
623             lr[Num[Indice]].material.color = Color.cyan;
624         }
625         isActive[alimentacion] = false;
626     }
627 }
628 }
```

```

...Single - copia\Assets\Códigos\Tradicional\Man_Valv.cs 16
624     //Si hay fuga
625     else
626     {
627         //Si viene de un estado previamente alimentado
628         if (botones[alimentacion].GetComponent<MeshRenderer>
        ( ).material.color == Color.white && fuga2 == true)
629         {
630             botones[camIzq].GetComponent<MeshRenderer>
        ( ).material.color = Color.white;
631             botones[camDer].GetComponent<MeshRenderer>
        ( ).material.color = Color.white;
632             isActive[camIzq] = false;
633             isActive[camDer] = false;
634             //Si tiene algo conectado la camara izquierda para NO
        dejar pasar la presión de aire
635             Indice = ConQuien(camIzq, Num);
636             if (Indice != 100)
637             {
638                 botones[Indice].GetComponent<MeshRenderer>
        ( ).material.color = Color.white;
639                 lr[Num[Indice]].material.color = Color.cyan;
640             }
641             //Si tiene algo conectado la camara derecha para NO dejar
        pasar la presión de aire
642             Indice = ConQuien(camDer, Num);
643             if (Indice != 100)
644             {
645                 botones[Indice].GetComponent<MeshRenderer>
        ( ).material.color = Color.white;
646                 lr[Num[Indice]].material.color = Color.cyan;
647             }
648             fuga2 = false;
649         }
650     }
651 }
652
653 public void ActivacionUM(int UM, int[] Num, bool[] isActive,
        GameObject[] botones)
654 {
655     //Si aún no ha dejado pasar el aire
656     if (isActive[UM] == false)
657     {
658         ////Si tiene algo conectado la unidad de mantenimiento para
        dejar pasar la presión de aire
659         Indice = ConQuien(UM, Num);
660         if (Indice != 100)
661         {
662             botones[UM].GetComponent<MeshRenderer>( ).material.color =
        Color.blue;

```

```

...Single - copia\Assets\Códigos\Tradicional\Man_Valv.cs 17
663         botones[Indice].GetComponent<MeshRenderere>
              ().material.color = Color.blue;
664         lr[Num[Indice]].material.color = Color.blue;
665     }
666     //Si hay fuga
667     else
668     {
669         botones[UM].GetComponent<MeshRenderere>().material.color =
              Color.red;
670         HayFuga(sonidoFuga);
671     }
672     isActive[UM] = true;
673
674 }
675 }
676
677 public void ActivacionPiloto(int Piloto, bool[] isActive, GameObject
[] botones)
678 {
679     //Si no hay fuga
680     if (IsActiveGeneral == false)
681     {
682         //Si necesita accionar su mecanismo para indicar que hay
              presión
683         if (botones[Piloto].GetComponent<MeshRenderere>
              ().material.color == Color.blue && isActive[Piloto] ==
              false)
684         {
685             animPiloto.SetBool("pilotoV2Bool", true);
686             HayFuga(sonidoFuga);
687             isActive[Piloto] = true;
688         }
689         //Si necesita accionar su mecanismo para indicar que NO hay
              presión
690         else if (botones[Piloto].GetComponent<MeshRenderere>
              ().material.color == Color.white && isActive[Piloto] ==
              true)
691         {
692             //piloto.GetComponent<MeshRenderere>().material.color =
              Color.black;
693             animPiloto.SetBool("pilotoV2Bool", false);
694             isActive[Piloto] = false;
695         }
696     }
697     //Si hay fuga
698     else
699     {
700         //Si estaba previamente accionado
701         if (animPiloto.GetBool("pilotoV2Bool") == true)

```

```

702     {
703         isActive[Piloto] = false;
704         animPiloto.SetBool("pilotoV2Bool", false);
705     }
706 }
707 }
708
709 public void Botonsito(int alimentación, int salida, string animBool,
    Animator animBoton, int[] Num, bool[] isActive, GameObject[]
    botones)
710 {
711     //Si se enclava el boton y se esta alimentando
712     if (botones[alimentación].GetComponent<MeshRenderer>
    ().material.color == Color.blue && isActive[alimentación] ==
    false && animBoton.GetBool(animBool) == true)
713     {
714         //Si esta conectado el boton para dejar pasar la presión
715         Indice = ConQuien(salida, Num);
716         if (Indice != 100)
717         {
718             //Verifica que no hay fuga
719             NoHayFuga(sonidoFuga, botones);
720
721             //Si no hay fuga
722             if (IsActiveGeneral == false)
723             {
724                 botones[salida].GetComponent<MeshRenderer>
    ().material.color = Color.blue;
725                 botones[Indice].GetComponent<MeshRenderer>
    ().material.color = Color.blue;
726                 lr[Num[Indice]].material.color = Color.blue;
727             }
728         }
729         //Si hay fuga
730         else
731         {
732             botones[salida].GetComponent<MeshRenderer>
    ().material.color = Color.red;
733             HayFuga(sonidoFuga);
734         }
735         isActive[alimentación] = true;
736         isActive[salida] = true;
737     }
738     //Si se desenclava el boton y se esta alimentando
739     else if (botones[alimentación].GetComponent<MeshRenderer>
    ().material.color == Color.blue && isActive[alimentación] ==
    true && animBoton.GetBool(animBool) == false)
740     {
741         botones[salida].GetComponent<MeshRenderer>().material.color =

```

```
        Color.white;
742         isActive[alimentación] = false;
743         isActive[salida] = false;
744         //Verifica que no hay fuga
745         NoHayFuga(sonidoFuga, botones);
746
747         //Si esta conectado el boton para NO dejar pasar la presión
748         Indice = ConQuien(salida, Num);
749         if (Indice != 100)
750         {
751             botones[Indice].GetComponent<MeshRenderere>          ↗
752                 ().material.color = Color.white;
753             lr[Num[Indice]].material.color = Color.cyan;
754         }
755         //Cuando esta conectado en serie y deja de haber paso de aire
756
757         if (botones[alimentación].GetComponent<MeshRenderere>    ↗
758             ().material.color == Color.white && isActive[alimentación] == ↗
759             true)
760         {
761             botones[salida].GetComponent<MeshRenderere>().material.color = ↗
762                 Color.white;
763             isActive[alimentación] = false;
764             isActive[salida] = false;
765
766             //Evitamos paso de aire
767             Indice = ConQuien(salida, Num);
768             if (Indice != 100)
769             {
770                 botones[Indice].GetComponent<MeshRenderere>          ↗
771                     ().material.color = Color.white;
772                 lr[Num[Indice]].material.color = Color.cyan;
773             }
774         }
775     }
776     public void TimerNegativo(int entrada, int salida, int control, int[] ↗
777         Num, bool[] isActive, GameObject[] botones)
778     {
779         //Se Comprueba que no hay fuga
780         Count = (float)slider_N.GetProgramVariable("_sliderValue");
781         NoHayFuga(sonidoFuga, botones);
782         //Si NO hay fuga
783         if (IsActiveGeneral == false)
784         {
785             //Cuando se alimenta el Timer
786             if (botones[entrada].GetComponent<MeshRenderere>          ↗
```



```
().material.color == Color.blue)
784     {
785         fuga3 = true;
786         isActive[entrada] = true;
787         if (isActive[control] == false)
788         {
789             //Cuando esté activada la entrada se activa la salida
790
791             Indice = ConQuien(salida, Num);
792             if (Indice != 100)
793             {
794                 botones[salida].GetComponent<MeshRenderer>
795                 ().material.color = Color.blue;
796                 botones[Indice].GetComponent<MeshRenderer>
797                 ().material.color = Color.blue;
798                 lr[Num[Indice]].material.color = Color.blue;
799             }
800             else
801             {
802                 botones[salida].GetComponent<MeshRenderer>
803                 ().material.color = Color.red;
804                 HayFuga(sonidoFuga);
805             }
806         }
807     }
808     //Cuando no esta alimentado el Timer
809     else
810     {
811         //Si viene de un estado anterior alimentado
812         if (isActive[entrada] == true)
813         {
814             botones[salida].GetComponent<MeshRenderer>
815             ().material.color = Color.white;
816             //Si tiene algo conectado a la Salida, evitamos paso
817             de aire
818             Indice = ConQuien(salida, Num);
819             if (Indice != 100)
820             {
821                 botones[Indice].GetComponent<MeshRenderer>
822                 ().material.color = Color.white;
823                 lr[Num[Indice]].material.color = Color.cyan;
824             }
825             isActive[entrada] = false;
826         }
827     }
828     //Botón Independiente de Control, activará el Timer
829     if (botones[control].GetComponent<MeshRenderer>
830     ().material.color == Color.blue)
831     {
```

```

...Single - copia\Assets\Códigos\Tradicional\Man_Valv.cs 21
824     isActive[control] = true;
825     //La entrada y control están activos?
826     if (isActive[entrada] == true && isActive[control] == true)
827     {
828         //Pregunta Si la salida está activa
829         if (botones[salida].GetComponent<MeshRender>
830             ().material.color == Color.blue)
831         {
832             //Al estar activa, se desactiva cuando el valor
833             //del TimerCount sea igual al Count = (por ahora igual a
834             //7 segundos)
835             if (TimerCount >= Count)
836             {
837                 botones[salida].GetComponent<MeshRender>
838                 ().material.color = Color.white;
839                 botones[Indice].GetComponent<MeshRender>
840                 ().material.color = Color.white;
841                 Indice = ConQuien(salida, Num);
842                 if (Indice != 100)
843                 {
844                     lr[Num[Indice]].material.color =
845                     Color.cyan;
846                     TimerCount = 0;
847                     isActive[salida] = false;
848                 }
849             }
850             //Si el TimerCount no es igual a (7) este irá
851             //contando segundos
852             else
853             {
854                 TimerCount += Time.deltaTime;
855             }
856         }
857         else
858         {
859             isActive[salida] = true;
860             //No está activada la salida, la volvemos true y
861             //vuelve a preguntar
862         }
863     }
864     //El AND no aplica
865     else
866     {
867         //Si la entrada está activa activamos la salida
868         if (isActive[entrada] == true)
869         {
870             isActive[salida] = true;

```

```
864         Indice = ConQuien(salida, Num);
865         if (Indice != 100)
866         {
867             botones[salida].GetComponent<MeshRenderer>
868             ().material.color = Color.blue;
869             botones[Indice].GetComponent<MeshRenderer>
870             ().material.color = Color.blue;
871             lr[Num[Indice]].material.color = Color.blue;
872         }
873         else
874         {
875             botones[salida].GetComponent<MeshRenderer>
876             ().material.color = Color.red;
877             HayFuga(sonidoFuga);
878         }
879     }
880     else
881     {
882         isActive[control] = false;
883     }
884     //Si hay fuga
885     else
886     {
887         //Si viene de un estado previamente alimentado
888         if (botones[entrada].GetComponent<MeshRenderer>
889             ().material.color == Color.white && fuga3 == true)
890         {
891             botones[salida].GetComponent<MeshRenderer>
892             ().material.color = Color.white;
893             botones[control].GetComponent<MeshRenderer>
894             ().material.color = Color.white;
895             isActive[salida] = false;
896             isActive[control] = false;
897
898             //Si tiene algo conectado a la Salida, evitaremos paso de
899             //aire
900             Indice = ConQuien(salida, Num);
901             if (Indice != 100)
902             {
903                 botones[Indice].GetComponent<MeshRenderer>
904                 ().material.color = Color.white;
905                 lr[Num[Indice]].material.color = Color.cyan;
906             }
907             fuga3 = false;
908         }
909     }
910 }
```

```
905     }
906
907 }
908
909 public void TimerPositivo(int entrada, int salida, int control, int[] ↗
    Num, bool[] isActive, GameObject[] botones)
910 {
911     //Se Comprueba que no hay fuga
912     Count = (float)slider_P.GetProgramVariable("_sliderValue");
913     NoHayFuga(sonidoFuga, botones);
914     //Si NO hay fuga
915     if (IsActiveGeneral == false)
916     {
917         //Cuando se alimenta el Timer
918         if (botones[entrada].GetComponent<MeshRenderere> ↗
            ().material.color == Color.blue)
919         {
920             fuga4 = true;
921             isActive[entrada] = true;
922             if (isActive[control] == true)
923             {
924                 if (isActive[salida] == false)
925                 {
926                     //Cuando esté activada la entrada y el control, ↗
927                     se activa la salida
928                     Indice = ConQuien(salida, Num);
929                     if (Indice != 100)
930                     {
931                         botones[salida].GetComponent<MeshRenderere> ↗
932                         ().material.color = Color.blue;
933                         botones[Indice].GetComponent<MeshRenderere> ↗
934                         ().material.color = Color.blue;
935                         lr[Num[Indice]].material.color = Color.blue;
936                         isActive[salida] = true;
937                     }
938                     else
939                     {
940                         botones[salida].GetComponent<MeshRenderere> ↗
941                         ().material.color = Color.red;
942                         HayFuga(sonidoFuga);
943                     }
944                 }
945             }
946         }
947     }
948     //Cuando no esta alimentado el Timer
949     else
950     {
```

```
948 //Si viene de un estado anterior alimentado
949 if (isActive[entrada] == true)
950 {
951     botones[salida].GetComponent<MeshRenderer>
952     ().material.color = Color.white;
953     //Si tiene algo conectado a la Salida, evitamos paso
954     de aire
955     Indice = ConQuien(salida, Num);
956     if (Indice != 100)
957     {
958         botones[Indice].GetComponent<MeshRenderer>
959         ().material.color = Color.white;
960         lr[Num[Indice]].material.color = Color.cyan;
961     }
962     isActive[entrada] = false;
963 }
964
965 if (isActive[control] == true)
966 {
967     if (isActive[salida] == true)
968     {
969         botones[salida].GetComponent<MeshRenderer>
970         ().material.color = Color.white;
971         //Cuando esté activada la entrada y el control,
972         se activa la salida
973         Indice = ConQuien(salida, Num);
974         if (Indice != 100)
975         {
976             botones[Indice].GetComponent<MeshRenderer>
977             ().material.color = Color.white;
978             lr[Num[Indice]].material.color = Color.cyan;
979         }
980         isActive[salida] = false;
981     }
982 }
983
984 //Botón Independiente de Control, activará el Timer
985 if (botones[control].GetComponent<MeshRenderer>
986     ().material.color == Color.blue)
987 {
988     isActive[control] = true;
989     //La entrada y control están activos?
990     if (isActive[entrada] == true && isActive[control] ==
991         true)
992     {
993         //Pregunta Si la salida está activa
```

```
989         if (isActive[salida] == true)
990         {
991             //Al estar activa, se desactiva cuando el valor
992             //del TimerCount sea igual al Count = (por ahora igual a
993             //7 segundos)
994             if (TimerCount >= Count)
995             {
996                 botones[salida].GetComponent<MeshRenderere>
997                 ().material.color = Color.blue;
998                 botones[Indice].GetComponent<MeshRenderere>
999                 ().material.color = Color.blue;
1000                 Indice = ConQuien(salida, Num);
1001                 if (Indice != 100)
1002                 {
1003                     lr[Num[Indice]].material.color =
1004                     Color.blue;
1005                     TimerCount = 0;
1006                     isActive[salida] = false;
1007                 }
1008             }
1009             //Si el TimerCount no es igual a (7) este irá
1010             //contando segundos
1011             else
1012             {
1013                 TimerCount += Time.deltaTime;
1014             }
1015         }
1016         else
1017         {
1018             isActive[salida] = true;
1019         }
1020         //No está activada la salida, la volvemos true y
1021         //vuelve a preguntar
1022     }
1023     //El AND no aplica
1024     else
1025     {
1026         //Si la entrada está activa activamos la salida
1027         if (isActive[salida] == false)
1028         {
```

```

...Single - copia\Assets\Códigos\Tradicional\Man_Valv.cs 26
1029         isActive[salida] = false;
1030         botones[salida].GetComponent<MeshRenderer>
1031         ().material.color = Color.white;
1032         //Si tiene algo conectado a la Salida, evitamos
1033         paso de aire
1034         Indice = ConQuien(salida, Num);
1035         if (Indice != 100)
1036         {
1037             botones[Indice].GetComponent<MeshRenderer>
1038             ().material.color = Color.white;
1039             lr[Num[Indice]].material.color = Color.cyan;
1040         }
1041     }
1042     else
1043     {
1044         isActive[control] = false;
1045         if (isActive[salida] == true)
1046         {
1047             isActive[salida] = false;
1048             botones[salida].GetComponent<MeshRenderer>
1049             ().material.color = Color.white;
1050             TimerCount = 0;
1051             //Si tiene algo conectado a la Salida, evitamos paso
1052             de aire
1053             Indice = ConQuien(salida, Num);
1054             if (Indice != 100)
1055             {
1056                 botones[Indice].GetComponent<MeshRenderer>
1057                 ().material.color = Color.white;
1058                 lr[Num[Indice]].material.color = Color.cyan;
1059             }
1060         }
1061     }
1062     //Si hay fuga
1063     else
1064     {
1065         //Si viene de un estado previamente alimentado
1066         if (botones[entrada].GetComponent<MeshRenderer>
1067         ().material.color == Color.white && fuga4 == true)
1068         {
1069             botones[salida].GetComponent<MeshRenderer>
1070             ().material.color = Color.white;
1071             botones[control].GetComponent<MeshRenderer>
1072             ().material.color = Color.white;
1073             isActive[salida] = false;
1074             isActive[control] = false;

```

```
1069
1070         //Si tiene algo conectado a la Salida, evitaremos paso de ↗
           aire
1071         Indice = ConQuien(salida, Num);
1072         if (Indice != 100)
1073         {
1074             botones[Indice].GetComponent<MeshRenderer> ↗
               ().material.color = Color.white;
1075             lr[Num[Indice]].material.color = Color.cyan;
1076         }
1077         fuga4 = false;
1078     }
1079 }
1080 }
1081 }
1082 }
1083 }
1084 public void FinalCarrera(int alimentación, int salida, string ↗
           animBool, Animator animBoton, int[] Num, bool[] isActive, ↗
           GameObject[] botones)
1085 {
1086     //Si no hay fuga
1087     if (IsActiveGeneral == false)
1088     {
1089         //Si es necesario activar la salida del final de carrera
1090         if (botones[alimentación].GetComponent<MeshRenderer> ↗
               ().material.color == Color.blue && isActive[alimentación] ↗
               == false && animBoton.GetBool(animBool) == true)
1091         {
1092             botones[salida].GetComponent<MeshRenderer> ↗
               ().material.color = Color.blue;
1093             isActive[alimentación] = true;
1094             isActive[salida] = true;
1095         }
1096         //Si esta conectado el final de carrera para dejar pasar ↗
           la presión
1097         Indice = ConQuien(salida, Num);
1098         if (Indice != 100)
1099         {
1100             botones[Indice].GetComponent<MeshRenderer> ↗
               ().material.color = Color.blue;
1101             lr[Num[Indice]].material.color = Color.blue;
1102         }
1103         //Si hay fuga
1104         else
1105         {
1106             botones[salida].GetComponent<MeshRenderer> ↗
               ().material.color = Color.red;
1107             HayFuga(sonidoFuga);
```



```

...Single - copia\Assets\Códigos\Tradicional\Man_Valv.cs 28
1108     }
1109     }
1110     //Si es necesario desactivar la salida del final de carrera
1111     else if (botones[alimentación].GetComponent<MeshRender>
           ().material.color == Color.blue && isActive[alimentación]
           == true && animBoton.GetBool(animBool) == false)
1112     {
1113         botones[salida].GetComponent<MeshRender>
           ().material.color = Color.white;
1114         isActive[alimentación] = false;
1115         isActive[salida] = false;
1116         //Si esta conectado el final de carrera para NO dejar
           pasar la presión
1117         Indice = ConQuien(salida, Num);
1118         if (Indice != 100)
1119         {
1120             botones[Indice].GetComponent<MeshRender>
           ().material.color = Color.white;
1121             lr[Num[Indice]].material.color = Color.cyan;
1122         }
1123     }
1124 }
1125 }
1126 //Si se deja de alimentar el final de carrera
1127 if (botones[alimentación].GetComponent<MeshRender>
           ().material.color == Color.white && isActive[alimentación] ==
           true)
1128 {
1129     botones[salida].GetComponent<MeshRender>().material.color =
           Color.white;
1130     isActive[alimentación] = false;
1131     isActive[salida] = false;
1132     //Si esta conectado el final de carrera para NO dejar pasar
           la presión
1133     Indice = ConQuien(salida, Num);
1134     if (Indice != 100)
1135     {
1136         botones[Indice].GetComponent<MeshRender>
           ().material.color = Color.white;
1137     }
1138 }
1139 }
1140
1141 public int ConQuien(int x, int[] ListaNum)
1142 {
1143     //Contador 1
1144     int i = 100;
1145
1146     //si esta conecado con otra conexion

```

```
1147     if (ListaNum[x] != 100)
1148     {
1149         i = 0;
1150         //Contador 2
1151         bool j = true;
1152         while (j)
1153         {
1154             //Si encuentra con quien esta conectado
1155             if (ListaNum[x] == ListaNum[i] && x != i)
1156             {
1157                 j = false; //Salir del ciclo
1158             }
1159             else
1160             {
1161                 i++;
1162             }
1163         }
1164     }
1165     //Regresa con quien esta conectado
1166     return i;
1167 }
1168
1169 public void ConectarT(int i, int j, int k, int[] Num, bool[]
1170     isActive, GameObject[] botones)
1171 {
1172     //Entrada de T por i
1173     if (botones[i].GetComponent<MeshRenderer>().material.color ==
1174         Color.blue && isActive[i] == false)
1175     {
1176         //Verificación de 1|0
1177         Indice = ConQuien(j, Num);
1178         if (Indice != 100)
1179         {
1180             botones[j].GetComponent<MeshRenderer>().material.color =
1181                 Color.blue;
1182             botones[Indice].GetComponent<MeshRenderer>
1183                 ().material.color = Color.blue;
1184             isActive[j] = true;
1185             lr[Num[Indice]].material.color = Color.blue;
1186         }
1187         else
1188         {
1189             botones[j].GetComponent<MeshRenderer>().material.color =
1190                 Color.red;
1191             HayFuga(sonidoFuga);
1192         }
1193     }
1194 }
```

```

...Single - copia\Assets\Códigos\Tradicional\Man_Valv.cs 30
1191 //Verificación de 2|0
1192 Indice = ConQuien(k, Num);
1193 if (Indice != 100)
1194 {
1195     botones[k].GetComponent<MeshRenderer>().material.color = ↗
        Color.blue;
1196     botones[Indice].GetComponent<MeshRenderer>
        ()material.color = Color.blue; ↗
1197     isActive[k] = true;
1198     lr[Num[Indice]].material.color = Color.blue;
1199 }
1200 else
1201 {
1202     botones[k].GetComponent<MeshRenderer>().material.color = ↗
        Color.red;
1203     HayFuga(sonidoFuga);
1204 }
1205
1206
1207     isActive[i] = true;
1208 }
1209
1210 //Entrada de T por j
1211 if (botones[j].GetComponent<MeshRenderer>().material.color == ↗
    Color.blue && isActive[j] == false)
1212 {
1213     //Verificación de 0|1
1214     Indice = ConQuien(i, Num);
1215     if (Indice != 100)
1216     {
1217         botones[i].GetComponent<MeshRenderer>().material.color = ↗
            Color.blue;
1218         botones[Indice].GetComponent<MeshRenderer>
            ()material.color = Color.blue; ↗
1219         isActive[i] = true;
1220         lr[Num[Indice]].material.color = Color.blue;
1221     }
1222     else
1223     {
1224         botones[i].GetComponent<MeshRenderer>().material.color = ↗
            Color.red;
1225         HayFuga(sonidoFuga);
1226     }
1227
1228 //Verificación de 2|1
1229 Indice = ConQuien(k, Num);
1230 if (Indice != 100)
1231 {
1232     botones[k].GetComponent<MeshRenderer>().material.color = ↗

```

```

...Single - copia\Assets\Códigos\Tradicional\Man_Valv.cs 31
        Color.blue;
1233     botones[Indice].GetComponent<MeshRenderer>           ↗
        ().material.color = Color.blue;
1234     isActive[k] = true;
1235     lr[Num[Indice]].material.color = Color.blue;
1236     }
1237     else
1238     {
1239         botones[k].GetComponent<MeshRenderer>().material.color = ↗
        Color.red;
1240         HayFuga(sonidoFuga);
1241     }
1242
1243
1244     isActive[j] = true;
1245 }
1246
1247 //Entrada de T por k
1248 if (botones[k].GetComponent<MeshRenderer>().material.color == ↗
    Color.blue && isActive[k] == false)
1249 {
1250     //Verificación de 0|2
1251     Indice = ConQuien(i, Num);
1252     if (Indice != 100)
1253     {
1254         botones[i].GetComponent<MeshRenderer>().material.color = ↗
        Color.blue;
1255         botones[Indice].GetComponent<MeshRenderer>           ↗
        ().material.color = Color.blue;
1256         isActive[i] = true;
1257         lr[Num[Indice]].material.color = Color.blue;
1258     }
1259     else
1260     {
1261         botones[i].GetComponent<MeshRenderer>().material.color = ↗
        Color.red;
1262         HayFuga(sonidoFuga);
1263     }
1264
1265     //Verificación de 1|2
1266     Indice = ConQuien(j, Num);
1267     if (Indice != 100)
1268     {
1269         botones[j].GetComponent<MeshRenderer>().material.color = ↗
        Color.blue;
1270         botones[Indice].GetComponent<MeshRenderer>           ↗
        ().material.color = Color.blue;
1271         isActive[j] = true;
1272         lr[Num[Indice]].material.color = Color.blue;

```

```

...Single - copia\Assets\Códigos\Tradicional\Man_Valv.cs 32
1273     }
1274     else
1275     {
1276         botones[j].GetComponent<MeshRenderer>().material.color = ↗
            Color.red;
1277         HayFuga(sonidoFuga);
1278     }
1279
1280
1281     isActive[k] = true;
1282 }
1283
1284
1285 //DESCONECTAR CONEXIONES
1286
1287 if ((botones[i].GetComponent<MeshRenderer>().material.color == ↗
    Color.white && isActive[i] == true) || (botones
    [j].GetComponent<MeshRenderer>().material.color == Color.white ↗
    && isActive[j] == true) ||
1288     (botones[k].GetComponent<MeshRenderer>().material.color == ↗
    Color.white && isActive[k] == true))
1289 {
1290     botones[i].GetComponent<MeshRenderer>().material.color = ↗
    Color.white;
1291     botones[j].GetComponent<MeshRenderer>().material.color = ↗
    Color.white;
1292     botones[k].GetComponent<MeshRenderer>().material.color = ↗
    Color.white;
1293     isActive[i] = false;
1294     isActive[j] = false;
1295     isActive[k] = false;
1296
1297     //Busca las conexiones de las T para volverlas blancas
1298     Indice = ConQuien(i, Num);
1299     if (Indice != 100)
1300     {
1301         botones[Indice].GetComponent<MeshRenderer>
            ()).material.color = Color.white; ↗
1302         lr[Num[Indice]].material.color = Color.cyan;
1303     }
1304
1305     Indice = ConQuien(j, Num);
1306     if (Indice != 100)
1307     {
1308         botones[Indice].GetComponent<MeshRenderer>
            ()).material.color = Color.white; ↗
1309         lr[Num[Indice]].material.color = Color.cyan;
1310     }
1311

```

```

...Single - copia\Assets\Códigos\Tradicional\Man_Valv.cs 33
1312     Indice = ConQuien(k, Num);
1313     if (Indice != 100)
1314     {
1315         botones[Indice].GetComponent<MeshRenderer>
            ().material.color = Color.white;
1316         lr[Num[Indice]].material.color = Color.cyan;
1317     }
1318
1319
1320     }
1321
1322 }
1323
1324 public void HayFuga(AudioSource sonidoFuga)
1325 {
1326     //En el momento que encuentre una conexión roja
1327     sonidoFuga.Play();
1328     IsActiveGeneral = true;
1329 }
1330
1331 public void NoHayFuga(AudioSource sonidoFuga, GameObject[] Botones)
1332 {
1333     //Booleano auxiliar
1334     bool bol = false;
1335     //Recorre las conexiones para buscar si alguna conexión tiene
            fuga (Es roja)
1336     foreach (GameObject conexion in Botones)
1337     {
1338         if (conexion.GetComponent<MeshRenderer>().material.color ==
            Color.red)
1339         {
1340             bol = true;
1341             break;
1342         }
1343     }
1344     //Si no encuentra fuga (Conexión roja)
1345     if (bol == false)
1346     {
1347         sonidoFuga.Stop();
1348         IsActiveGeneral = false;
1349     }
1350 }
1351
1352 public void ActualizarLineRenderer(int a, int b, int c, int[] Num,
            GameObject[] botones)
1353 {
1354     int i = 0;
1355     int Indice = 100;
1356     //Si tiene alguna conexión

```

```
1357     if (Num[a] != 100)
1358     {
1359         //Busca con quien esta conectado
1360         for (i = 0; i < Num.Length; i++)
1361         {
1362             if (Num[i] == Num[a] && a != i)
1363             {
1364                 Indice = i;
1365             }
1366         }
1367         //Si lo encuentra lo esta cambiando de posicion en todo momento
1368         if (Indice != 100)
1369         {
1370             lr[Num[a]].SetPosition(0, botones[a].transform.position);
1371             lr[Num[a]].SetPosition(1, botones
1372                 [Indice].transform.position);
1373         }
1374     }
1375     i = 0;
1376     Indice = 100;
1377     //Si tiene alguna conexión
1378     if (Num[b] != 100)
1379     {
1380         //Busca con quien esta conectado
1381         for (i = 0; i < Num.Length; i++)
1382         {
1383             if (Num[i] == Num[b] && b != i)
1384             {
1385                 Indice = i;
1386             }
1387         }
1388     }
1389     //Si lo encuentra lo esta cambiando de posicion en todo momento
1390     if (Indice != 100)
1391     {
1392         lr[Num[b]].SetPosition(0, botones[b].transform.position);
1393         lr[Num[b]].SetPosition(1, botones
1394             [Indice].transform.position);
1395     }
1396 }
1397
1398 i = 0;
1399 Indice = 100;
1400 //Si tiene alguna conexión
1401 if (Num[c] != 100)
```

```
1402     {
1403         //Busca con quien esta conectado
1404         for (i = 0; i < Num.Length; i++)
1405         {
1406             if (Num[i] == Num[c] && c != i)
1407             {
1408                 Indice = i;
1409             }
1410         }
1411         //Si lo encuentra lo esta cambiando de posicion en todo momento
1412         if (Indice != 100)
1413         {
1414             lr[Num[c]].SetPosition(0, botones[c].transform.position);
1415             lr[Num[c]].SetPosition(1, botones
1416                 [Indice].transform.position);
1417         }
1418     }
1419 }
1420
1421 public override void OnPlayerJoined(VRCPlayerApi player)
1422 {
1423
1424     //SendCustomNetworkEvent(NetworkEventTarget.All,
1425     "ActualizarPanel");
1426     //if (Networking.IsMaster)
1427     //{
1428     //    if (animMan.GetBool("ManBool") == true)
1429     //    {
1430     //        SendCustomNetworkEvent(NetworkEventTarget.All,
1431     //        "ManBoolTrue");
1432     //    }
1433     //    else
1434     //    {
1435     //        SendCustomNetworkEvent(NetworkEventTarget.All,
1436     //        "ManBoolFalse");
1437     //    }
1438     //}
1439     RequestSerialization();
1440 }
1441
1442 public void ManBoolTrue()
1443 {
1444     for (int i = 0; i < this.botones.Length; i++)
1445     {
1446         botones[i].GetComponent<BoxCollider>().enabled = false;
1447     }
1448 }
```



```
1446     }
1447 }
1448
1449 public void ManBoolFalse()
1450 {
1451
1452 }
1453 public void ActualizarPanel()
1454 {
1455     int IndiceNet;
1456     //     izqOder = izqOder;
1457     //     izqOder2 = izqOder2;
1458     //     izqOderPiston = izqOderPiston;
1459     if (Networking.IsMaster)
1460     {
1461         for (int i = 0; i < Num.Length; i++)
1462         {
1463
1464             if (Num[i] != 100)
1465             {
1466                 Num[i] = Num[i];
1467
1468                 //Busca quien a que indice le corresponde cada Num
1469                 for (int busca = 0; busca < Num.Length; busca++)
1470                 {
1471                     if (Num[busca] == Num[i] && i != busca)
1472                     {
1473                         IndiceNet = i;
1474                         Num[IndiceNet] = Num[i];
1475                         lr[Num[i]].SetPosition(0, botones
1476 [i].transform.position);
1477                         lr[Num[i]].SetPosition(1, botones
1478 [IndiceNet].transform.position);
1479                     }
1480                     //Si lo encuentra lo esta cambiando de posicion en
1481                     todo momento
1482                 }
1483             }
1484         }
1485     }
1486 }
1487
1488 }
```

14. Apéndice F. Código Modificado de la Mesas de Trabajo Neumáticas que controlan los gemelos digitales de la empacadora y la indentadora

```

1 using UdonSharp;
2 using UnityEngine;
3 using VRC.SDKBase;
4 using VRC.Udon;
5 using TMPPro;
6 using UnityEngine.UI;
7 using System;
8 using VRC.Udon.Common.Interfaces;
9
10
11 public class Man_Valv : UdonSharpBehaviour
12 {
13     //Un arreglo de líneas que actúan como mangueras
14     public LineRenderer[] lr = new LineRenderer[102];
15     //Un arreglo de Objetos que son las conexiones de los elementos
16     public GameObject[] botones = new GameObject[102];
17     // Variables Sincronizadas
18     [UdonSynced]
19     public bool[] isPresed = new bool[102]; // variable que indica si un ↗
20     boton fue presionado
21     [UdonSynced]
22     public bool[] isOrigin = new bool[102]; // variable que indica cual ↗
23     es la primera conexión
24     [UdonSynced]
25     public bool[] isTarget = new bool[102]; // variable que indica cual es ↗
26     la segunda conexión
27     [UdonSynced]
28     public bool[] isActive = new bool[102]; // variable que indica que ↗
29     variables reciben presión de aire
30     [UdonSynced]
31     public bool desactivados = false; // variable que indica cuando se ↗
32     activa la unidad de mantenimiento y hay presión de aire
33     [UdonSynced]
34     public bool IsActiveGeneral = false; // detector de fugas. Es true ↗
35     cuando hay fuga
36     [UdonSynced]
37     public bool fuga = false;
38     [UdonSynced]
39     public bool fuga2 = false;
40     [UdonSynced]
41     public bool fuga3 = false;
42     [UdonSynced]
43     public bool fuga4 = false;
44     [UdonSynced]
45     private float TimerCount;
46     private float Count;
47
48     public int[] Num = new int[102];
49     [UdonSynced]

```

```

44 public int indexOriginActive = 800; //
45 [UdonSynced]
46 public int Indice;
47 [UdonSynced]
48 public int numarreglo;
49 public Animator animMan, animBoton, animVastago2,
50     animVastago3, animValv52B, animValv52B_2,
51     animValv52B_3, animValv52B_4,
52     animFinal, animFinal2, animFinal3, // Agregué
53     animVastago3, Agregue animValv52B_2
54     animFinal4 , animPiloto; // Agregué animFinal3 y
55     animFinal4
56 public AudioSource sonidoPiston, sonidoFuga;
57
58 public UdonBehaviour slider_N; // control Deslizante para el
59     temporizador negativo
60 public UdonBehaviour slider_P; // control Deslizante para el
61     temporizador positivo
62
63
64
65 public void Start()
66 {
67     //Inicializamos animaciones para colocarlas en su estado inicial
68     animValv52B.SetBool("anim52Bool", true);
69     animPiloto.SetBool("pilotoV2Bool", false);
70
71     //Inicializamos en falso lo arreglos de booleanos para conectar
72     for (int i = 0; i < isPresed.Length; i++)
73     {
74         isPresed[i] = false;
75         isOrigin[i] = false;
76         isActive[i] = false;
77     }
78     //Inicializamos los num en 100 para que no esten conectados con
79     nada
80     for (int i = 0; i < Num.Length; i++)
81     {
82         Num[i] = 1000; //
83     }
84     //Inicializamos el tamaño de los linerenderer
85     for (int i = 0; i < lr.Length; i++)
86     {
87         lr[i].positionCount = 2;
88     }
89 }
90
91 void Update()
92 {
93     /*

```

```

87     Debug.Log("El Botón esta conectado: " + Num[10]);
88
89     Debug.Log("Valor de desactivados: " + desactivados);
90     Debug.Log("Valor de IsActiveGeneral: " + IsActiveGeneral);
91     Debug.Log("Valor de fuga: " + fuga);
92     Debug.Log("Valor de fuga2: " + fuga2);
93     Debug.Log("Valor de fuga3: " + fuga3);
94     Debug.Log("Valor de fuga4: " + fuga4);*/
95
96     Debug.Log("animMan.GetBool(\"ManBool\")" + animMan.GetBool
97         ("ManBool"));
98
99     //Si NO hay presión de salida en la unidad de mantenimiento
100    // No se ha activa la entrada de aire a la unidad de
101    mantenimiento
102    if (animMan.GetBool("ManBool") == false) // "ManBool es la
103    variable que activa la animación de la unidad de mantenimiento"
104    {
105        // La variable desactivados es true cuando esta cerrada la
106        válvula de paso
107        // y es false cuando se activa la válvula de paso
108        if (desactivados == false) // La válvula no está activa, no
109        hay paso de aire
110        {
111            //Apaga el sonido, Ya no hay fuga, Apagas la unidad de
112            mantenimiento y el piloto
113            sonidoFuga.Stop();
114            IsActiveGeneral = false; // está variable solo he visto
115            que se activa cuando hay fugas
116            botones[101].GetComponent<MeshRenderer>().material.color
117            = Color.white; // no hay paso de aire y todos los
118            conectores se tornan de color blanco
119            animPiloto.SetBool("pilotoV2Bool", false); // no hay paso
120            de aire el piloto no se activa
121
122            //Para todos los botones
123            for (int i = 0; i < this.botones.Length; i++)
124            {
125                //Todo boton con conexión (Num!=100) lo hace color
126                Cyan
127                if (Num[i] != 1000) // si boton conectado
128                {
129                    botones[i].GetComponent<MeshRenderer>
130                    ().material.color = Color.cyan;
131                    lr[Num[i]].material.color = Color.cyan;
132                }
133            }
134            //Todo boton sin conexión (Num=100) lo hace color

```

```

124         blanco
125         else
126         {
127             botones[i].GetComponent<MeshRenderrer>
128             ().material.color = Color.white;
129         }
130         //No desactivamos el Active del piston de doble
131         efecto
132         /*if (i != 89)
133             isActive[i] = false;*/
134         // Desactivamos todos los active excepto el de los
135         pistones de doble efecto
136         if( i != 89)
137         {
138             if( i != 95)
139             {
140                 isActive[i] = false;
141             }
142         }
143         //Habilitamos los collider de las conexiones
144         foreach (GameObject conexion in botones)
145         {
146             conexion.GetComponent<BoxCollider>().enabled = true;
147         }
148         //Para que solo haga esto una vez
149         desactivados = true;
150     }
151
152     //Pregunta continuamente a cada boton si ha sido presionado
153     for (int i = 0; i < this.botones.Length; i++)
154     {
155         if (isPresed[i] && !isOrigin[i] && !isTarget[i])
156         {
157             //Activar origen
158             if (indexOriginActive == 800) // 80 está conectadp
159             {
160                 isOrigin[i] = true;
161                 botones[i].GetComponent<MeshRenderrer>
162                 ().material.color = Color.yellow;
163                 indexOriginActive = i;
164                 Num[i] = i;
165                 isPresed[i] = false;
166             }
167             //Activar target y realiza la conexión
168         }
169         else

```

```

...elos Digitales\Assets\Códigos\Tradicional\Man_Valv.cs 5
168         {
169             isTarget[i] = true;
170             botones[i].GetComponent<MeshRenderer>
171             ().material.color = Color.cyan;
172             lr[indexOriginActive].enabled = true;
173             lr[indexOriginActive].SetPosition(0, botones
174             [indexOriginActive].transform.position);
175             lr[indexOriginActive].SetPosition(1, botones
176             [i].transform.position);
177
178             botones
179             [indexOriginActive].GetComponent<MeshRenderer>
180             ().material.color = Color.cyan;
181             Num[i] = indexOriginActive;
182             indexOriginActive = 800;
183             isPresed[i] = false;
184         }
185     }
186     //Desactivar
187     // Desactiva un boton con conexión
188     else if ((isOrigin[i] && isPresed[i] || isTarget[i] &&
189     isPresed[i]))
190     {
191         //Caso una sola conexión
192         if (botones[i].GetComponent<MeshRenderer>
193         ().material.color == Color.yellow)
194         {
195             numarreglo = Num[i];
196             for (int j = 0; j < Num.Length; j++)
197             {
198                 if (Num[j] == numarreglo)
199                 {
200                     isOrigin[j] = false;
201                     isTarget[j] = false;
202                     isPresed[j] = false;
203                     botones[j].GetComponent<MeshRenderer>
204                     ().material.color = Color.white;
205                     lr[j].enabled = false;
206                     Num[j] = 1000; //
207                 }
208             }
209             indexOriginActive = 800; //
210         }
211     }
212
213     //Caso desconectar dos azules
214     else if (botones[i].GetComponent<MeshRenderer>
215     ().material.color == Color.cyan && indexOriginActive ==
216     800)

```

```

207     {
208         numarreglo = Num[i];
209         for (int j = 0; j < Num.Length; j++)
210         {
211             if (Num[j] == numarreglo)
212             {
213                 isOrigin[j] = false;
214                 isTarget[j] = false;
215                 isPresed[j] = false;
216                 botones[j].GetComponent<MeshRenderer>
217                 ().material.color = Color.white;
218                 lr[j].enabled = false;
219                 Num[j] = 1000; //
220             }
221             indexOriginActive = 800; //
222         }
223
224         //Caso desconectar un amarillo cuando se trata de
225         conectar con un azul
226         else if (botones[i].GetComponent<MeshRenderer>
227         ().material.color == Color.cyan && indexOriginActive !=
228         800)
229         {
230             isPresed[i] = false;
231             isOrigin[indexOriginActive] = false;
232             botones
233             [indexOriginActive].GetComponent<MeshRenderer>
234             ().material.color = Color.white;
235             Num[indexOriginActive] = 1000; //
236             indexOriginActive = 800; //
237         }
238     }
239
240     //si hay presión de salida en la unidad de mantenimiento
241     else
242     {
243         //Desactiva las uniones para poder dar paso al aire
244         if (desactivados == true)
245         {
246             foreach (GameObject conexion in botones)
247             {
248                 conexion.GetComponent<BoxCollider>().enabled = false;
249             }
250             desactivados = false;
251         }
252     }

```



```

...elos Digitales\Assets\Códigos\Tradicional\Man_Valv.cs 7
250 //Activación de la unidad de mantenimiento
251 NoHayFuga(sonidoFuga, botones);
252 ActivacionUM(101, Num, isActive, botones); // unidad de  ➤
    mantenimiento
253
254 //Botones
255 Botonsito(61, 62, "Boton1", animBoton, Num, isActive,  ➤
    botones);
256 //Botonsito(20, 21, "Boton1", animBoton2, Num, isActive,  ➤
    botones); Se quitó esta línea
257
258 //FINALES DE CARRERA
259 FinalCarrera(91, 92, "finalBool", animFinal, Num, isActive,  ➤
    botones);
260 FinalCarrera(93, 94, "finalBool", animFinal2, Num, isActive,  ➤
    botones);
261 FinalCarrera(97, 98, "finalBool", animFinal3, Num, isActive,  ➤
    botones); // Se Agregó esta línea
262 FinalCarrera(99, 100, "finalBool", animFinal4, Num, isActive,  ➤
    botones); // Se Agregó esta línea
263
264 //VALVULAS
265 Valvula52Biestable(69, 70, 71, 72, 73, animValv52B,  ➤
    "val52Bool", Num, isActive, botones);
266 Valvula52Biestable(74, 75, 76, 77, 78, animValv52B_2,  ➤
    "val52Bool", Num, isActive, botones);// Se Agregó esta  ➤
    línea
267 Valvula52Biestable(79, 80, 81, 82, 83, animValv52B_3,  ➤
    "val52Bool", Num, isActive, botones);// Se Agregó esta  ➤
    línea
268 Valvula52Biestable(84, 85, 86, 87, 88, animValv52B_4,  ➤
    "val52Bool", Num, isActive, botones);// Se Agregó esta  ➤
    línea
269
270 // Se quitó la válvula monoestable
271 // Valvula52Monoestable(14, 11, 12, 13, animValv52M,  ➤
    "val52Bool", Num, isActive, botones);
272
273 //CONECTAR T
274 ConectarT(00, 01, 02, Num, isActive, botones);
275 ConectarT(03, 04, 05, Num, isActive, botones);
276 ConectarT(06, 07, 08, Num, isActive, botones);
277 ConectarT(09, 10, 11, Num, isActive, botones);
278 ConectarT(12, 13, 14, Num, isActive, botones); // Agregé esta  ➤
    línea
279 ConectarT(15, 16, 17, Num, isActive, botones); // Agregué  ➤
    esta línea
280 ConectarT(18, 19, 20, Num, isActive, botones); // Agregué  ➤
    esta línea

```

```

...elos Digitales\Assets\Códigos\Tradicional\Man_Valv.cs 8
281 ConectarT(21, 22, 23, Num, isActive, botones); // Agregué ↗
    esta línea
282 ConectarT(24, 25, 26, Num, isActive, botones); // Agregué ↗
    esta línea
283 ConectarT(27, 28, 29, Num, isActive, botones); // Agregué ↗
    esta línea
284 ConectarT(30, 31, 32, Num, isActive, botones); // Agregué ↗
    esta línea
285 ConectarT(33, 34, 35, Num, isActive, botones); // Agregué ↗
    esta línea
286 ConectarT(36, 37, 38, Num, isActive, botones); // Agregué ↗
    esta línea
287 ConectarT(39, 40, 41, Num, isActive, botones); // Agregué ↗
    esta línea
288 ConectarT(42, 43, 44, Num, isActive, botones); // Agregué ↗
    esta línea
289 ConectarT(45, 46, 47, Num, isActive, botones); // Agregué ↗
    esta línea
290 ConectarT(48, 49, 50, Num, isActive, botones); // Agregué ↗
    esta línea
291 ConectarT(51, 52, 53, Num, isActive, botones); // Agregué ↗
    esta línea
292 ConectarT(54, 55, 56, Num, isActive, botones); // Agregué ↗
    esta línea
293 ConectarT(57, 58, 59, Num, isActive, botones); // Agregué ↗
    esta línea
294
295
296 // Movimiento de los Cilindros
297
298 ActivarPistonDoble(89, 90, animVastago2, "VastDobAnimBool", ↗
    sonidoPiston, isActive, botones);
299 ActivarPistonDoble(95, 96, animVastago3, "VastDobAnimBool", ↗
    sonidoPiston, isActive, botones);
300
301 //PPILOTO
302 ActivacionPiloto(60, isActive, botones);
303
304 //Timers Prueba
305 TimerNegativo(66, 68, 67, Num, isActive, botones);
306 TimerPositivo(65, 63, 64, Num, isActive, botones);
307
308 }
309 /*
310 //Permite mover las mangueras en tiempo real de las uniones
311 ActualizarLineRenderer(00, 01, 02, Num, botones);
312 ActualizarLineRenderer(03, 04, 05, Num, botones);
313 ActualizarLineRenderer(06, 07, 08, Num, botones);
314 ActualizarLineRenderer(09, 10, 11, Num, botones);

```

```
315 ActualizarLineRenderer(12, 13, 14, Num, botones);
316 ActualizarLineRenderer(15, 16, 17, Num, botones);
317 ActualizarLineRenderer(18, 19, 20, Num, botones);
318 ActualizarLineRenderer(21, 22, 23, Num, botones);
319 ActualizarLineRenderer(24, 25, 26, Num, botones);
320 ActualizarLineRenderer(27, 28, 29, Num, botones);
321 ActualizarLineRenderer(30, 31, 32, Num, botones);
322 ActualizarLineRenderer(33, 34, 35, Num, botones);
323 ActualizarLineRenderer(36, 37, 38, Num, botones);
324 */
325 for(int i = 0; i<=19; i++)
326 {
327     ActualizarLineRenderer(i*3, i*3+1, i*3+2, Num, botones);
328 }
329 }
330 /*
331 public void ActivarPistonSimple(int Piston, Animator animVastago,
332     string VastagoAnimBool, Animator animResorte, string
333     ResorteAnimBool, AudioSource sonidoPiston, bool[] isActive,
334     GameObject[] botones)
335 {
336     //Si NO hay fuga
337     if (IsActiveGeneral == false)
338     {
339         //Si es necesario que el piston se acciones
340         if (botones[Piston].GetComponent<MeshRenderer>
341             ().material.color == Color.blue && isActive[Piston] ==
342             false)
343         {
344             animVastago.SetBool(VastagoAnimBool, true);
345             animResorte.SetBool(ResorteAnimBool, true);
346             isActive[Piston] = true;
347             sonidoPiston.Play();
348         }
349         //si es necesario el piston regrese a su posicion inicial
350         else if (botones[Piston].GetComponent<MeshRenderer>
351             ().material.color == Color.white && isActive[Piston] ==
352             true)
353         {
354             animVastago.SetBool(VastagoAnimBool, false);
355             animResorte.SetBool(ResorteAnimBool, false);
356             isActive[Piston] = false;
357             sonidoPiston.Play();
358         }
359     }
360 }
361 //Si hay fuga
362 else
363 {
```

```

...elos Digitales\Assets\Códigos\Tradicional\Man_Valv.cs 10
357 //Si el piston esta accionado lo regresa a su posicion inicial
358 if (animVastago.GetBool(VastagoAnimBool) == true)
359 {
360     isActive[Piston] = false;
361     animVastago.SetBool(VastagoAnimBool, false);
362     animResorte.SetBool(ResorteAnimBool, false);
363 }
364 }
365 */
366
367 public void ActivarPistonDoble(int Entrada, int Salida, Animator
animVastago, string VastagoAnimBool, AudioSource sonidoPiston, bool
[] isActive, GameObject[] botones)
368 {
369     //Si NO hay fuga
370     if (IsActiveGeneral == false) // IsActiveGeneral (true) se activa
cuando hay fuga
371     {
372         //Si es necesario pasar a la posición de trabajo
373         if (botones[Entrada].GetComponent<MeshRenderer>
().material.color == Color.blue &&
374             botones[Salida].GetComponent<MeshRenderer>
().material.color != Color.blue &&
375             isActive[Entrada] == false)
376         {
377             animVastago.SetBool(VastagoAnimBool, true);
378             sonidoPiston.Play();
379             isActive[Entrada] = true;
380         }
381
382         //Si es necesario pasar a la posición de reposo
383         if (botones[Salida].GetComponent<MeshRenderer>
().material.color == Color.blue &&
384             botones[Entrada].GetComponent<MeshRenderer>
().material.color != Color.blue &&
385             isActive[Entrada] == true)
386         {
387             animVastago.SetBool(VastagoAnimBool, false);
388             sonidoPiston.Play();
389             isActive[Entrada] = false;
390         }
391     }
392 }
393 }
394
395 public void Valvula52Biestable(int alimentacion, int botIzq, int
camIzq, int camDer, int botDer, Animator anim52Bool, string
val52Bool, int[] Num, bool[] isActive, GameObject[] botones)

```

```
396     {
397         // Comprueba si ya no hay fuga
398         // si hay fuga se establece IsActiveGeneral == true
399         // creo que no es necesario volver a llamar la función pues se ↗
           ejecuta continuamente en el Update
400         NoHayFuga(sonidoFuga, botones);
401
402         //Si NO hay fuga
403         if (IsActiveGeneral == false)
404         {
405
406             if (botones[botDer].GetComponent<MeshRenderer> ↗
           ().material.color != Color.blue &&
407             botones[botIzq].GetComponent<MeshRenderer> ↗
           ().material.color == Color.blue)
408         {
409             anim52Bool.SetBool(val52Bool, true);
410         }
411
412             if (botones[botDer].GetComponent<MeshRenderer> ↗
           ().material.color == Color.blue &&
413             botones[botIzq].GetComponent<MeshRenderer> ↗
           ().material.color != Color.blue)
414         {
415             anim52Bool.SetBool(val52Bool, false);
416         }
417
418             //Se está alimentando la valvula
419             if (botones[alimentacion].GetComponent<MeshRenderer> ↗
           ().material.color == Color.blue)
420         {
421             fuga2 = true;
422
423
424             //Si el pilotaje 14 esta activado y el 12 NO, SE ACTIVA ↗
           LA UTILIZACIÓN IZQUIERDA (4)
425             //Si esta activa la UTILIZACIÓN derecha (2)
426             if (anim52Bool.GetBool(val52Bool) == false)
427             {
428                 isActive[camIzq] = false;
429                 isActive[camDer] = true;
430
431
432                 botones[camIzq].GetComponent<MeshRenderer> ↗
           ().material.color = Color.white;
433                 // Si aún no ha dejado pasar la presión la camara (2)
434                 // al principio isActive [camDer] es falso, pues aún ↗
           no se activado
435
```

```

...elos Digitales\Assets\Códigos\Tradicional\Man_Valv.cs 12
436     Indice = ConQuien(camDer, Num);
437     if (Indice != 1000)
438     {
439         botones[camDer].GetComponent<MeshRenderrer>
440     ().material.color = Color.blue;
441         botones[Indice].GetComponent<MeshRenderrer>
442     ().material.color = Color.blue;
443         lr[Num[Indice]].material.color = Color.blue;
444     }
445     else
446     {
447         botones[camDer].GetComponent<MeshRenderrer>
448     ().material.color = Color.red;
449         HayFuga(sonidoFuga);
450     }
451     Indice = ConQuien(camIzq, Num);
452     if (Indice != 1000)
453     {
454         botones[Indice].GetComponent<MeshRenderrer>
455     ().material.color = Color.white;
456         lr[Num[Indice]].material.color = Color.cyan;
457     }
458     //Si esta activada la UTILIZACIÓN 4 (izquierda)
459     if (anim52Bool.GetBool(val52Bool) == true)
460     {
461         isActive[camDer] = false;
462         isActive[camIzq] = true;
463
464         anim52Bool.SetBool(val52Bool, true);
465         Debug.Log("Entro aqui en cam izq true por favor dime
466     o gran unity");
467
468         botones[camDer].GetComponent<MeshRenderrer>
469     ().material.color = Color.white;
470
471         Indice = ConQuien(camIzq, Num);
472         if (Indice != 1000)
473         {
474             botones[camIzq].GetComponent<MeshRenderrer>
475         ().material.color = Color.blue;
476             botones[Indice].GetComponent<MeshRenderrer>
477         ().material.color = Color.blue;
478             lr[Num[Indice]].material.color = Color.blue;

```

```

477     }
478     //Si hay fuga
479     else
480     {
481         botones[camIzq].GetComponent<MeshRenderer>
482         ().material.color = Color.red;
483         HayFuga(sonidoFuga);
484     }
485     Indice = ConQuien(camDer, Num);
486     if (Indice != 1000)
487     {
488         botones[Indice].GetComponent<MeshRenderer>
489         ().material.color = Color.white;
490         lr[Num[Indice]].material.color = Color.cyan;
491     }
492 }
493
494 }
495 //Si NO se esta alimentando la valvula
496
497 if (botones[alimentacion].GetComponent<MeshRenderer>
498     ().material.color != Color.blue)
499 {
500     isActive[alimentacion] = false;
501     //isActive[camIzq] = false;
502     //isActive[camDer] = false;
503     botones[camIzq].GetComponent<MeshRenderer>
504     ().material.color = Color.white;
505     botones[camDer].GetComponent<MeshRenderer>
506     ().material.color = Color.white;
507
508     Indice = ConQuien(camIzq, Num);
509     if (Indice != 1000)
510     {
511         botones[Indice].GetComponent<MeshRenderer>
512         ().material.color = Color.white;
513         lr[Num[Indice]].material.color = Color.cyan;
514     }
515     //Si tiene algo conectado la camara derecha para NO dejar
516     //pasar la presión de aire
517     Indice = ConQuien(camDer, Num);
518     if (Indice != 1000)
519     {
520         botones[Indice].GetComponent<MeshRenderer>
521         ().material.color = Color.white;
522         lr[Num[Indice]].material.color = Color.cyan;
523     }
524 }

```

```
518
519
520     }
521 }
522 //Si hay fuga
523 else
524 {
525     // Si viene de un estado previamente alimentado
526     // aquí utiliza la variable fuga, fuga 2 se establecio true  ➤
527     // en 595
528     if (botones[alimentacion].GetComponent<MeshRender> ➤
529         ().material.color == Color.white && fuga2 == true)
530     {
531         botones[camIzq].GetComponent<MeshRender> ➤
532             ().material.color = Color.white;
533         botones[camDer].GetComponent<MeshRender> ➤
534             ().material.color = Color.white;
535
536         //Si tiene algo conectado la camara izquierda para NO ➤
537         //dejar pasar la presión de aire
538         Indice = ConQuien(camIzq, Num);
539         if (Indice != 1000)
540         {
541             botones[Indice].GetComponent<MeshRender> ➤
542                 ().material.color = Color.white;
543             lr[Num[Indice]].material.color = Color.cyan;
544         }
545         //Si tiene algo conectado la camara derecha para NO dejar ➤
546         //pasar la presión de aire
547         Indice = ConQuien(camDer, Num);
548         if (Indice != 1000)
549         {
550             botones[Indice].GetComponent<MeshRender> ➤
551                 ().material.color = Color.white;
552             lr[Num[Indice]].material.color = Color.cyan;
553         }
554         fuga2 = false;
555     }
556 }
557 }
558 }
559 }
560
561 public void ActivacionUM(int UM, int[] Num, bool[] isActive, ➤
562     GameObject[] botones)
563 {
564     //Si aún no ha dejado pasar el aire
565     if (isActive[UM] == false)
566     {
567         ////Si tiene algo conectado la unidad de mantenimiento para ➤
568         //dejar pasar la presión de aire
```



```
557     Indice = ConQuien(UM, Num);
558     if (Indice != 1000)
559     {
560         botones[UM].GetComponent<MeshRenderer>().material.color = Color.blue;
561         botones[Indice].GetComponent<MeshRenderer>().material.color = Color.blue;
562         lr[Num[Indice]].material.color = Color.blue;
563     }
564     //Si hay fuga
565     else
566     {
567         botones[UM].GetComponent<MeshRenderer>().material.color = Color.red;
568         HayFuga(sonidoFuga);
569     }
570     isActive[UM] = true;
571 }
572 }
573 }
574
575 public void ActivacionPiloto(int Piloto, bool[] isActive, GameObject[] botones)
576 {
577     //Si no hay fuga
578     if (IsActiveGeneral == false)
579     {
580         //Si necesita accionar su mecanismo para indicar que hay presión
581         if (botones[Piloto].GetComponent<MeshRenderer>().material.color == Color.blue && isActive[Piloto] == false)
582         {
583             animPiloto.SetBool("pilotoV2Bool", true);
584             HayFuga(sonidoFuga);
585             isActive[Piloto] = true;
586         }
587         //Si necesita accionar su mecanismo para indicar que NO hay presión
588         else if (botones[Piloto].GetComponent<MeshRenderer>().material.color == Color.white && isActive[Piloto] == true)
589         {
590             //piloto.GetComponent<MeshRenderer>().material.color = Color.black;
591             animPiloto.SetBool("pilotoV2Bool", false);
592             isActive[Piloto] = false;
593         }
594     }
```

```
595     //Si hay fuga
596     else
597     {
598         //Si estaba previamente accionado
599         if (animPiloto.GetBool("pilotoV2Bool") == true)
600         {
601             isActive[Piloto] = false;
602             animPiloto.SetBool("pilotoV2Bool", false);
603         }
604     }
605 }
606
607 public void Botonsito(int alimentación, int salida, string animBool,
608     Animator animBoton, int[] Num, bool[] isActive, GameObject[]
609     botones)
610 {
611     //Si se enclava el boton y se esta alimentando
612     if (botones[alimentación].GetComponent<MeshRenderer>
613         ().material.color == Color.blue && isActive[alimentación] ==
614         false && animBoton.GetBool(animBool) == true)
615     {
616         //Si esta conectado el boton para dejar pasar la presión
617         Indice = ConQuien(salida, Num);
618         if (Indice != 1000) // cambié 100 por 1000
619         {
620             //Verifica que no hay fuga
621             NoHayFuga(sonidoFuga, botones);
622
623             //Si no hay fuga
624             if (IsActiveGeneral == false)
625             {
626                 botones[salida].GetComponent<MeshRenderer>
627                 ().material.color = Color.blue;
628                 botones[Indice].GetComponent<MeshRenderer>
629                 ().material.color = Color.blue;
630                 lr[Num[Indice]].material.color = Color.blue;
631             }
632         }
633         //Si hay fuga
634         else
635         {
636             botones[salida].GetComponent<MeshRenderer>
637             ().material.color = Color.red;
638             HayFuga(sonidoFuga);
639         }
640         isActive[alimentación] = true;
641         isActive[salida] = true;
642     }
643     //Si se desenclava el boton y se esta alimentando
```

```

...elos Digitales\Assets\Códigos\Tradicional\Man_Valv.cs 17
637     else if (botones[alimentación].GetComponent<MeshRender>  ↗
        ().material.color == Color.blue && isActive[alimentación] ==  ↗
        true && animBoton.GetBool(animBool) == false)
638     {
639         botones[salida].GetComponent<MeshRender>().material.color =  ↗
        Color.white;
640         isActive[alimentación] = false;
641         isActive[salida] = false;
642         //Verifica que no hay fuga
643         NoHayFuga(sonidoFuga, botones);
644
645         //Si esta conectado el boton para NO dejar pasar la presión
646         Indice = ConQuien(salida, Num);
647         if (Indice != 1000)
648         {
649             botones[Indice].GetComponent<MeshRender>  ↗
        ().material.color = Color.white;
650             lr[Num[Indice]].material.color = Color.cyan;
651         }
652     }
653     //Cuando esta conectado en serie y deja de haber paso de aire
654
655     if (botones[alimentación].GetComponent<MeshRender>  ↗
        ().material.color == Color.white && isActive[alimentación] ==  ↗
        true)
656     {
657         botones[salida].GetComponent<MeshRender>().material.color =  ↗
        Color.white;
658         isActive[alimentación] = false;
659         isActive[salida] = false;
660
661         //Evitamos paso de aire
662         Indice = ConQuien(salida, Num);
663         if(Indice !=1000)
664         {
665             botones[Indice].GetComponent<MeshRender>  ↗
        ().material.color = Color.white;
666             lr[Num[Indice]].material.color = Color.cyan;
667         }
668     }
669
670 }
671
672 public void TimerNegativo(int entrada, int salida, int control, int[]  ↗
        Num, bool[] isActive, GameObject[] botones)
673 {
674     //Se Comprueba que no hay fuga
675     Count = (float)slider_N.GetProgramVariable("_sliderValue");
676     NoHayFuga(sonidoFuga, botones);

```

```
677
678     //Si NO hay fuga
679     if (IsActiveGeneral == false)
680     {
681         // Cuando se alimenta el Timer
682         // cuando está alimentado el timer la salida del temporizador >
683         // está activa
684         if (botones[entrada].GetComponent<MeshRenderer> >
685             ().material.color == Color.blue) // Entra aire a la entrada >
686         del temporizador
687     {
688         fuga3 = true; // Sirve para indicar que el timer ha >
689         estado en funcionamiento
690         isActive[entrada] = true; // indica que isActive es >
691         verdadero para la entrada
692         if (isActive[control] == false) // No se ha activado el >
693         control
694     {
695         //Cuando esté activada la entrada está activa la >
696         salida
697         // Un temporizador negativo es normalmete abierto
698         Indice = ConQuien(salida, Num); // Busca con quien >
699         está conectada la salida
700         if (Indice != 1000)
701         {
702             botones[salida].GetComponent<MeshRenderer> >
703             ().material.color = Color.blue;
704             botones[Indice].GetComponent<MeshRenderer> >
705             ().material.color = Color.blue;
706             lr[Num[Indice]].material.color = Color.blue;
707         }
708         else // si no encuentra conexión se establece que hay >
709         fuga
710         {
711             botones[salida].GetComponent<MeshRenderer> >
712             ().material.color = Color.red;
713             HayFuga(sonidoFuga);
714         }
715     }
716 } // hasta aqui todo bien
717 //Cuando no esta alimentado el Timer
718 //No hay salida de aire
719 if (botones[entrada].GetComponent<MeshRenderer> >
720     ().material.color != Color.blue)
721 {
722     //Si viene de un estado anterior alimentado
723     if (isActive[entrada] == true)
724     {
725         botones[salida].GetComponent<MeshRenderer> >
```

```
    ().material.color = Color.white;
713     //Si tiene algo conectado a la Salida, evitamos paso ↗
    de aire
714     Indice = ConQuien(salida, Num);
715     if (Indice != 1000)
716     {
717         botones[Indice].GetComponent<MeshRenderer> ↗
718         ().material.color = Color.white;
719         lr[Num[Indice]].material.color = Color.cyan;
720     }
721     isActive[entrada] = false;
722 }
723 }
724 //Botón Independiente de Control, activará el Timer
725 if (botones[control].GetComponent<MeshRenderer> ↗
726     ().material.color == Color.blue)
727 {
728     isActive[control] = true;
729     //Si la entrada y el control están activos
730     if (isActive[entrada] == true && isActive[control] == ↗
731         true)
732     {
733         // y si la salida está activa, entonces empieza a ↗
734         contar el timer para desactivar la salida
735         if (botones[salida].GetComponent<MeshRenderer> ↗
736             ().material.color == Color.blue)
737         {
738             //Al estar activa, se desactiva cuando el valor ↗
739             del TimerCount sea igual al Count
740             if (TimerCount >= Count + 0.2f)
741             {
742                 botones[salida].GetComponent<MeshRenderer> ↗
743                 ().material.color = Color.white;
744
745                 Indice = ConQuien(salida, Num);
746
747                 botones[Indice].GetComponent<MeshRenderer> ↗
748                 ().material.color = Color.white;
749                 if (Indice != 1000)
750                 {
751                     lr[Num[Indice]].material.color = ↗
752                     Color.cyan;
753                     TimerCount = 0;
754                     isActive[salida] = false;
755                 }
756             }
757         }
758     }
759     //Si el TimerCount no es igual a (7) este irá ↗
760     contando segundos
```

```
750         else
751         {
752             TimerCount += Time.deltaTime;
753         }
754     }
755     if (botones[salida].GetComponent<MeshRenderrer>
756     ().material.color != Color.blue)
757     {
758         isActive[salida] = false;
759         //No está activada la salida, la volvemos true y
760         //vuelve a preguntar
761     }
762     //El AND no aplica
763     else
764     {
765
766         //Si la entrada está activa activamos la salida
767         if (isActive[entrada] == true)
768         {
769             isActive[salida] = true;
770             Indice = ConQuien(salida, Num);
771             if (Indice != 1000)
772             {
773                 botones[salida].GetComponent<MeshRenderrer>
774                 ().material.color = Color.blue;
775                 botones[Indice].GetComponent<MeshRenderrer>
776                 ().material.color = Color.blue;
777                 lr[Num[Indice]].material.color = Color.blue;
778             }
779             else
780             {
781                 botones[salida].GetComponent<MeshRenderrer>
782                 ().material.color = Color.red;
783                 HayFuga(sonidoFuga);
784             }
785         }
786     }
787     else
788     {
789         isActive[control] = false;
790     }
791     //Si hay fuga
792     else
793     {
794         //Si viene de un estado previamente alimentado
```

```

...elos Digitales\Assets\Códigos\Tradicional\Man_Valv.cs 21
794         if (botones[entrada].GetComponent<MeshRenderer> ➤
            ().material.color == Color.white && fuga3 == true)
795         {
796             botones[salida].GetComponent<MeshRenderer> ➤
            ().material.color = Color.white;
797             botones[control].GetComponent<MeshRenderer> ➤
            ().material.color = Color.white;
798             isActive[salida] = false;
799             isActive[control] = false;
800
801             //Si tiene algo conectado a la Salida, evitaremos paso de ➤
            aire
802             Indice = ConQuien(salida, Num);
803             if (Indice != 1000)
804             {
805                 botones[Indice].GetComponent<MeshRenderer> ➤
            ().material.color = Color.white;
806                 lr[Num[Indice]].material.color = Color.cyan;
807             }
808             fuga3 = false;
809         }
810     }
811 }
812 }
813 }
814
815 public void TimerPositivo(int entrada, int salida, int control, int[] ➤
    Num, bool[] isActive, GameObject[] botones)
816 {
817     //Se Comprueba que no hay fuga
818     Count = (float)slider_P.GetProgramVariable("_sliderValue");
819     NoHayFuga(sonidoFuga, botones);
820     //Si NO hay fuga
821     if (IsActiveGeneral == false)
822     {
823         //Cuando se alimenta el Timer
824         if (botones[entrada].GetComponent<MeshRenderer> ➤
            ().material.color == Color.blue)
825         {
826             fuga4 = true;
827             isActive[entrada] = true;
828             if (isActive[control] == true)
829             {
830                 if (isActive[salida]==false)
831                 {
832                     //Cuando esté activada la entrada y el control, ➤
            se activa la salida
833                     Indice = ConQuien(salida, Num);
834                     if (Indice != 1000)

```

```
835         {
836             botones[salida].GetComponent<MeshRenderer>
837             ().material.color = Color.blue;
838             botones[Indice].GetComponent<MeshRenderer>
839             ().material.color = Color.blue;
840             lr[Num[Indice]].material.color = Color.blue;
841             isActive[salida] = true;
842         }
843         else
844         {
845             botones[salida].GetComponent<MeshRenderer>
846             ().material.color = Color.red;
847             HayFuga(sonidoFuga);
848         }
849     }
850 }
851 //Cuando no esta alimentado el Timer
852 else
853 {
854     //Si viene de un estado anterior alimentado
855     if (isActive[entrada] == true)
856     {
857         botones[salida].GetComponent<MeshRenderer>
858         ().material.color = Color.white;
859         //Si tiene algo conectado a la Salida, evitamos paso
860         de aire
861         Indice = ConQuien(salida, Num);
862         if (Indice != 1000)
863         {
864             botones[Indice].GetComponent<MeshRenderer>
865             ().material.color = Color.white;
866             lr[Num[Indice]].material.color = Color.cyan;
867         }
868         isActive[entrada] = false;
869     }
870 }
871 if (isActive[control] == true)
872 {
873     if (isActive[salida] == true)
874     {
875         botones[salida].GetComponent<MeshRenderer>
876         ().material.color = Color.white;
877         //Cuando esté activada la entrada y el control,
878         se activa la salida
879         Indice = ConQuien(salida, Num);
```



```

876         if (Indice != 1000)
877         {
878             botones[Indice].GetComponent<MeshRenderer>
879             ().material.color = Color.white;
880             lr[Num[Indice]].material.color = Color.cyan;
881         }
882         isActive[salida] = false;
883     }
884 }
885 }
886 }
887 //Botón Independiente de Control, activará el Timer
888 if (botones[control].GetComponent<MeshRenderer>
889     ().material.color == Color.blue)
890 {
891     isActive[control] = true;
892     //La entrada y control están activos?
893     if (isActive[entrada] == true && isActive[control] ==
894         true)
895     {
896         //Pregunta Si la salida está activa
897         if (isActive[salida] == true)
898         {
899             //Al estar activa, se desactiva cuando el valor
900             //del TimerCount sea igual al Count = (por ahora igual a
901             //7 segundos)
902             if (TimerCount >= Count)
903             {
904                 botones[salida].GetComponent<MeshRenderer>
905                 ().material.color = Color.blue;
906                 Indice = ConQuien(salida, Num);
907                 botones[Indice].GetComponent<MeshRenderer>
908                 ().material.color = Color.blue;
909             }
910             if (Indice != 1000)
911             {
912                 lr[Num[Indice]].material.color =
913                 Color.blue;
914                 TimerCount = 0;
915                 isActive[salida] = false;
916             }
917             else
918             {
919                 botones
920                 [salida].GetComponent<MeshRenderer>().material.color =
921                 Color.red;
922                 HayFuga(sonidoFuga);
923             }
924         }
925     }
926 }

```

```
915     }
916     //Si el TimerCount no es igual a (7) este irá ▶
contando segundos
917     else
918     {
919         TimerCount += Time.deltaTime;
920     }
921
922     }
923     else
924     {
925         isActive[salida] = true;
926     }
927     //No está activada la salida, la volvemos true y ▶
vuelve a preguntar
928
929 }
930 //El AND no aplica
931 else
932 {
933     //Si la entrada está activa activamos la salida
934     if (isActive[salida] == false)
935     {
936         isActive[salida] = false;
937         botones[salida].GetComponent<MeshRenderrer> ▶
().material.color = Color.white;
938         //Si tiene algo conectado a la Salida, evitamos ▶
paso de aire
939         Indice = ConQuien(salida, Num);
940         if (Indice != 1000)
941         {
942             botones[Indice].GetComponent<MeshRenderrer> ▶
().material.color = Color.white;
943             lr[Num[Indice]].material.color = Color.cyan;
944         }
945     }
946 }
947 }
948 else
949 {
950     isActive[control] = false;
951     if (isActive[salida]== true)
952     {
953         isActive[salida] = false;
954         botones[salida].GetComponent<MeshRenderrer> ▶
().material.color = Color.white;
955         TimerCount = 0;
956         //Si tiene algo conectado a la Salida, evitamos paso ▶
de aire
```

```

957         Indice = ConQuien(salida, Num);
958         if (Indice != 1000)
959         {
960             botones[Indice].GetComponent<MeshRenderer>
961             ().material.color = Color.white;
962             lr[Num[Indice]].material.color = Color.cyan;
963         }
964     }
965 }
966 //Si hay fuga
967 else
968 {
969     //Si viene de un estado previamente alimentado
970     if (botones[entrada].GetComponent<MeshRenderer>
971     ().material.color == Color.white && fuga4 == true)
972     {
973         botones[salida].GetComponent<MeshRenderer>
974         ().material.color = Color.white;
975         botones[control].GetComponent<MeshRenderer>
976         ().material.color = Color.white;
977         isActive[salida] = false;
978         isActive[control] = false;
979
980         //Si tiene algo conectado a la Salida, evitaremos paso de
981         //aire
982         Indice = ConQuien(salida, Num);
983         if (Indice != 1000)
984         {
985             botones[Indice].GetComponent<MeshRenderer>
986             ().material.color = Color.white;
987             lr[Num[Indice]].material.color = Color.cyan;
988         }
989         fuga4 = false;
990     }
991 }
992
993 public void FinalCarrera(int alimentación, int salida, string
994 animBool, Animator animBoton, int[] Num, bool[] isActive,
995 GameObject[] botones)
996 {
997     //Si no hay fuga
998     if (IsActiveGeneral == false)
999     {
1000         //Si es necesario activar la salida del final de carrera
1001         if (botones[alimentación].GetComponent<MeshRenderer>

```

```

...elos Digitales\Assets\Códigos\Tradicional\Man_Valv.cs 26
    ().material.color == Color.blue &&
998     isActive[alimentación] == false && animBoton.GetBool   ↗
        (animBool) == true)
999     {
1000     botones[salida].GetComponent<MeshRenderrer>           ↗
        ().material.color = Color.blue;
1001     isActive[alimentación] = true;
1002     isActive[salida] = true;
1003
1004     //Si esta conectado el final de carrera para dejar pasar   ↗
        la presión
1005     Indice = ConQuien(salida, Num);
1006     if (Indice != 1000)
1007     {
1008     botones[Indice].GetComponent<MeshRenderrer>           ↗
        ().material.color = Color.blue;
1009     lr[Num[Indice]].material.color = Color.blue;
1010     }
1011     //Si hay fuga
1012     else
1013     {
1014     botones[salida].GetComponent<MeshRenderrer>           ↗
        ().material.color = Color.red;
1015     HayFuga(sonidoFuga);
1016     }
1017     }
1018     //Si es nescesario desactivar la salida del final de carrera
1019     else if (botones[alimentación].GetComponent<MeshRenderrer>   ↗
        ().material.color == Color.blue && isActive[alimentación]   ↗
        == true && animBoton.GetBool(animBool) == false)
1020     {
1021     botones[salida].GetComponent<MeshRenderrer>           ↗
        ().material.color = Color.white;
1022     isActive[alimentación] = false;
1023     isActive[salida] = false;
1024     //Si esta conectado el final de carrera para NO dejar   ↗
        pasar la presión
1025     Indice = ConQuien(salida, Num);
1026     if (Indice != 1000)
1027     {
1028     botones[Indice].GetComponent<MeshRenderrer>           ↗
        ().material.color = Color.white;
1029     lr[Num[Indice]].material.color = Color.cyan;
1030     }
1031     }
1032     }
1033     }
1034     //Si se deja de alimentar el final de carrara
1035     if (botones[alimentación].GetComponent<MeshRenderrer>           ↗

```

```
().material.color == Color.white && isActive[alimentación] == true)
1036     {
1037         botones[salida].GetComponent<MeshRender>().material.color = Color.white;
1038         isActive[alimentación] = false;
1039         isActive[salida] = false;
1040         //Si esta conectado el final de carrera para NO dejar pasar la presión
1041         Indice = ConQuien(salida, Num);
1042         if (Indice != 1000)
1043         {
1044             botones[Indice].GetComponent<MeshRender>().material.color = Color.white;
1045         }
1046     }
1047 }
1048
1049 public int ConQuien(int x, int[] ListaNum) // con quien está conectado Num[x]
1050 {
1051     //Contador 1
1052     int i = 1000; // si no esta conectado a nada regresa el valor de 100
1053
1054     //si esta conectado con otra conexion
1055     if (ListaNum[x] != 1000)
1056     {
1057         i = 0;
1058         //Contador 2
1059         bool j = true;
1060         while (j)
1061         {
1062             //Si encuentra con quien esta conectado
1063             if (ListaNum[x] == ListaNum[i] && x != i)
1064             {
1065                 j = false; //Salir del ciclo
1066             }
1067             else
1068             {
1069                 i++;
1070             }
1071         }
1072     }
1073     //Regresa con quien esta conectado
1074     return i;
1075 }
1076 }
1077
```

```

...eLos Digitales\Assets\Códigos\Tradicional\Man_Valv.cs 28
1078 public void ConectarT(int i, int j, int k, int[] Num, bool[] ▶
    isActive, GameObject[] botones)
1079 {
1080
1081     //Entrada de T por i
1082     if (botones[i].GetComponent<MeshRenderer>().material.color == ▶
        Color.blue && isActive[i] == false)
1083     {
1084         //Verificación de 1|0
1085         Indice = ConQuien(j, Num);
1086         if (Indice != 1000)
1087         {
1088             botones[j].GetComponent<MeshRenderer>().material.color = ▶
                Color.blue;
1089             botones[Indice].GetComponent<MeshRenderer> ▶
                ().material.color = Color.blue;
1090             isActive[j] = true;
1091             lr[Num[Indice]].material.color = Color.blue;
1092         }
1093     } else
1094     {
1095         botones[j].GetComponent<MeshRenderer>().material.color = ▶
            Color.red;
1096         HayFuga(sonidoFuga);
1097     }
1098
1099     //Verificación de 2|0
1100     Indice = ConQuien(k, Num);
1101     if (Indice != 1000)
1102     {
1103         botones[k].GetComponent<MeshRenderer>().material.color = ▶
            Color.blue;
1104         botones[Indice].GetComponent<MeshRenderer> ▶
            ().material.color = Color.blue;
1105         isActive[k] = true;
1106         lr[Num[Indice]].material.color = Color.blue;
1107     }
1108 } else
1109 {
1110     botones[k].GetComponent<MeshRenderer>().material.color = ▶
        Color.red;
1111     HayFuga(sonidoFuga);
1112 }
1113
1114     isActive[i] = true;
1115 }
1116
1117 //Entrada de T por j
1118

```

```

...elos Digitales\Assets\Códigos\Tradicional\Man_Valv.cs 29
1119     if (botones[j].GetComponent<MeshRenderrer>().material.color == Color.blue && isActive[j] == false)
1120     {
1121         //Verificación de 0|1
1122         Indice = ConQuien(i, Num);
1123         if (Indice != 1000)
1124         {
1125             botones[i].GetComponent<MeshRenderrer>().material.color = Color.blue;
1126             botones[Indice].GetComponent<MeshRenderrer>().material.color = Color.blue;
1127             isActive[i] = true;
1128             lr[Num[Indice]].material.color = Color.blue;
1129         }
1130     else
1131     {
1132         botones[i].GetComponent<MeshRenderrer>().material.color = Color.red;
1133         HayFuga(sonidoFuga);
1134     }
1135
1136     //Verificación de 2|1
1137     Indice = ConQuien(k, Num);
1138     if (Indice != 1000)
1139     {
1140         botones[k].GetComponent<MeshRenderrer>().material.color = Color.blue;
1141         botones[Indice].GetComponent<MeshRenderrer>().material.color = Color.blue;
1142         isActive[k] = true;
1143         lr[Num[Indice]].material.color = Color.blue;
1144     }
1145     else
1146     {
1147         botones[k].GetComponent<MeshRenderrer>().material.color = Color.red;
1148         HayFuga(sonidoFuga);
1149     }
1150
1151     isActive[j] = true;
1152 }
1153
1154
1155 //Entrada de T por k
1156 if (botones[k].GetComponent<MeshRenderrer>().material.color == Color.blue && isActive[k] == false)
1157 {
1158     //Verificación de 0|2
1159     Indice = ConQuien(i, Num);

```

```

1160     if (Indice != 1000)
1161     {
1162         botones[i].GetComponent<MeshRender>().material.color = Color.blue;
1163         botones[Indice].GetComponent<MeshRender>().material.color = Color.blue;
1164         isActive[i] = true;
1165         lr[Num[Indice]].material.color = Color.blue;
1166     }
1167     else
1168     {
1169         botones[i].GetComponent<MeshRender>().material.color = Color.red;
1170         HayFuga(sonidoFuga);
1171     }
1172
1173     //Verificación de 1|2
1174     Indice = ConQuien(j, Num);
1175     if (Indice != 1000)
1176     {
1177         botones[j].GetComponent<MeshRender>().material.color = Color.blue;
1178         botones[Indice].GetComponent<MeshRender>().material.color = Color.blue;
1179         isActive[j] = true;
1180         lr[Num[Indice]].material.color = Color.blue;
1181     }
1182     else
1183     {
1184         botones[j].GetComponent<MeshRender>().material.color = Color.red;
1185         HayFuga(sonidoFuga);
1186     }
1187
1188
1189     isActive[k] = true;
1190 }
1191
1192
1193 //DESCONECTAR CONEXIONES
1194
1195 if ((botones[i].GetComponent<MeshRender>().material.color == Color.white &&
1196     isActive[i] == true) ||
1197     (botones[j].GetComponent<MeshRender>().material.color == Color.white &&
1198     isActive[j] == true) ||
1199     (botones[k].GetComponent<MeshRender>().material.color == Color.white &&
1200     isActive[k] == true))
1201 {
1202     botones[i].GetComponent<MeshRender>().material.color =

```



```

...elos Digitales\Assets\Códigos\Tradicional\Man_Valv.cs 31
    Color.white;
1200     botones[j].GetComponent<MeshRender>().material.color = Color.white;
    Color.white;
1201     botones[k].GetComponent<MeshRender>().material.color = Color.white;
    Color.white;
1202     isActive[i] = false;
1203     isActive[j] = false;
1204     isActive[k] = false;
1205
1206     //Busca las conexiones de las T para volverlas blancas
1207     Indice = ConQuien(i, Num);
1208     if (Indice != 1000)
1209     {
1210         botones[Indice].GetComponent<MeshRender>().material.color = Color.white;
1211         lr[Num[Indice]].material.color = Color.cyan;
1212     }
1213
1214     Indice = ConQuien(j, Num);
1215     if (Indice != 1000)
1216     {
1217         botones[Indice].GetComponent<MeshRender>().material.color = Color.white;
1218         lr[Num[Indice]].material.color = Color.cyan;
1219     }
1220
1221     Indice = ConQuien(k, Num);
1222     if (Indice != 1000)
1223     {
1224         botones[Indice].GetComponent<MeshRender>().material.color = Color.white;
1225         lr[Num[Indice]].material.color = Color.cyan;
1226     }
1227
1228
1229     }
1230
1231 }
1232
1233 public void HayFuga(AudioSource sonidoFuga)
1234 {
1235     //En el momento que encuentre una conexión roja
1236     sonidoFuga.Play();
1237     IsActiveGeneral = true;
1238 }
1239
1240 public void NoHayFuga(AudioSource sonidoFuga, GameObject[] Botones)
1241 {
1242     //Booleano auxiliar

```

```
1243     bool bol = false;
1244     //Recorre las conexiones para buscar si alguna conexion tiene fuga (Es roja)
1245     foreach (GameObject conexion in Botones)
1246     {
1247         if (conexion.GetComponent<MeshRenderer>().material.color == Color.red)
1248         {
1249             bol = true;
1250             break;
1251         }
1252     }
1253     //Si no encuentra fuga (Conexion roja)
1254     if (bol == false)
1255     {
1256         sonidoFuga.Stop();
1257         IsActiveGeneral = false;
1258     }
1259 }
1260
1261 public void ActualizarLineRenderer(int a, int b, int c, int[] Num, GameObject[] botones)
1262 {
1263     int i = 0;
1264     int Indice = 1000;
1265     //Si tiene alguna conexión
1266     if (Num[a] != 1000)
1267     {
1268         //Busca con quien esta conectado
1269         for (i = 0; i < Num.Length; i++)
1270         {
1271             if (Num[i] == Num[a] && a != i)
1272             {
1273                 Indice = i;
1274             }
1275         }
1276         //Si lo encuentra lo esta cambiando de posicion en todo momento
1277         if (Indice != 1000)
1278         {
1279             lr[Num[a]].SetPosition(0, botones[a].transform.position);
1280             lr[Num[a]].SetPosition(1, botones[Indice].transform.position);
1281         }
1282     }
1283 }
1284
1285 i = 0;
```

```

1287     Indice = 1000;
1288     //Si tiene alguna conexión
1289     if (Num[b] != 1000)
1290     {
1291         //Busca con quien esta conectado
1292         for (i = 0; i < Num.Length; i++)
1293         {
1294             if (Num[i] == Num[b] && b != i)
1295             {
1296                 Indice = i;
1297             }
1298         }
1299         //Si lo encuentra lo esta cambiando de posicion en todo momento ➤
1300         if (Indice != 1000)
1301         {
1302             lr[Num[b]].SetPosition(0, botones[b].transform.position);
1303             lr[Num[b]].SetPosition(1, botones ➤
1304                 [Indice].transform.position);
1305         }
1306     }
1307     i = 0;
1308     Indice = 1000;
1309     //Si tiene alguna conexión
1310     if (Num[c] != 1000)
1311     {
1312         //Busca con quien esta conectado
1313         for (i = 0; i < Num.Length; i++)
1314         {
1315             if (Num[i] == Num[c] && c != i)
1316             {
1317                 Indice = i;
1318             }
1319         }
1320         //Si lo encuentra lo esta cambiando de posicion en todo momento ➤
1321         if (Indice != 1000)
1322         {
1323             lr[Num[c]].SetPosition(0, botones[c].transform.position);
1324             lr[Num[c]].SetPosition(1, botones ➤
1325                 [Indice].transform.position);
1326         }
1327     }
1328 }
1329 }
1330
1331

```

```

1332 public override void OnPlayerJoined(VRCPlayerApi player)
1333 {
1334
1335     //SendCustomNetworkEvent(NetworkEventTarget.All,           ➤
1336     //    "ActualizarPanel");
1337     //if (Networking.IsMaster)
1338     //{
1339     //    if(animMan.GetBool("ManBool") == true)
1340     //    {
1341     //        SendCustomNetworkEvent(NetworkEventTarget.All,     ➤
1342     //            "ManBoolTrue");
1343     //    }
1344     //    else
1345     //    {
1346     //        SendCustomNetworkEvent(NetworkEventTarget.All,     ➤
1347     //            "ManBoolFalse");
1348     //    }
1349     //}
1350     RequestSerialization();
1351 }
1352
1353 public void ManBoolTrue()
1354 {
1355     for (int i = 0; i < this.botones.Length; i++)
1356     {
1357         botones[i].GetComponent<BoxCollider>().enabled = false;
1358     }
1359 }
1360
1361 public void ManBoolFalse()
1362 {
1363 }
1364
1365 public void ActualizarPanel()
1366 {
1367     int IndiceNet;
1368     //    izqOder = izqOder;
1369     //    izqOder2 = izqOder2;
1370     //    izqOderPiston = izqOderPiston;
1371     if (Networking.IsMaster)
1372     {
1373         for (int i = 0; i < Num.Length; i++)
1374         {
1375             if (Num[i] != 1000)
1376             {
1377                 Num[i] = Num[i];

```

```

1378
1379         //Busca quien a que indice le corresponde cada Num
1380         for (int busca = 0; busca < Num.Length; busca++)
1381         {
1382             if (Num[busca] == Num[i] && i != busca)
1383             {
1384                 IndiceNet = i;
1385                 Num[IndiceNet] = Num[i];
1386                 lr[Num[i]].SetPosition(0, botones
1387                 [i].transform.position);
1388                 lr[Num[i]].SetPosition(1, botones
1389                 [IndiceNet].transform.position);
1390             }
1391             //Si lo encuentra lo esta cambiando de posicion en
1392             todo momento
1393         }
1394     }
1395 }
1396
1397 }
1398
1399 }
```