UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

SIMULACIÓN DE UN ENTORNO INDUSTRIAL MEDIANTE LA HERRAMIENTA DE TRABAJO COPPELIASIM (V-REP)

TRABAJO FINAL DEL

Grado en Ingeniería Electrónica Industrial y Automática

REALIZADO POR

Daniel Díaz Llorca

TUTORIZADO POR

Leopoldo Armesto Ángel

CURSO ACADÉMICO: 2019/2020

Agradecimientos

En primer lugar he de agradecer a mis padres todo el apoyo mostrado durante estos últimos años. Este proyecto es vuestro también.

Agradecer también a mi hermano el cual me ha guiado por el sendero correcto cuando las cosas se ponían difíciles. Eres un referente tanto en lo profesional como en lo personal.

Muchísimas gracias a mi tutor Leopoldo Armesto Ángel por la ayuda brindada en la realización de este proyecto.

Por último pero no menos importante agradecer a todos mis amigos el apoyo moral y la confianza depositada en mí en todo momento. Sin vosotros no lo habría conseguido.

Índice General

Documento 1. Memoria Técnica

Tabla de contenidos

1. Introducción

1.1. Objetivo

La elaboración de este proyecto comprende el diseño total y completo de la simulación de una planta industrial basada en la clasificación de elementos dependiendo del color.

La simulación ha sido creada a través de la herramienta de trabajo CoppeliaSim (V-Rep). Se ha escogido este software debido a la gran versatilidad de este y a su relativa poca complejidad a la hora de programar. A través de este proyecto se pretende dominar dicha herramienta de trabajo.

1.2. Resumen

El proyecto está basado en la recreación de una planta industrial formada por los siguientes elementos. Cuatro cintas transportadoras encargadas de transportar los diferentes elementos al lugar deseado. Siete sensores de posición, un sensor de visión y un sensor de color encargados de enviar a los demás elementos los distintos valores necesarios para su correcto funcionamiento. Un robot industrial cuya misión es la de seleccionar varios elementos y moverlos de manera precisa a una nueva posición. Mediante estos elementos se ha creado un sistema en el cual el robot industrial es capaz de clasificar una serie diferentes de elementos que serán transportados mediante una de las cintas, siendo desechados los elementos no adecuados.

2. Desarrollo del proyecto

2.1. Contexto

Este proyecto ha sido realizado con la finalidad de poder implementarlo en una nave industrial real. Debido a la generalidad de los elementos esto hace que las distintas modificaciones necesarias para la puesta en marcha no sean un problema.

2.2. Software utilizado: CoppeliaSim (V-Rep)

La herramienta de trabajo utilizada ha sido CoppeliaSim (V-Rep) en su versión EDU 4.0.0. Dicho simulador se basa en una arquitectura de control distribuido en la que cada objeto puede ser controlado individualmente a través de un script integrado.

CoppeliaSim se utiliza mayormente para el desarrollo rápido de algoritmos, simulaciones de automatización de fábricas, creación rápida de prototipos y verificación, monitoreo remoto y educación relacionada con la robótica.

2.2.1. Principales características

CoppeliaSim contiene las características ideales para la simulación de un entorno industrial. Gracias a su arquitectura de control distribuida en la cual cada elemento se puede controlar individualmente.

2.2.1.1. Métodos de programación

CoppeliaSim ofrece seis métodos distintos de programación o codificación diferentes, cada uno con sus ventajas particulares sobre los demás, pero los seis son mutuamente compatibles, es decir se puede utilizar al mismo tiempo.

Los seis métodos son los siguientes:

-*Scripts* embebidos: este método consiste en escribir *scripts* Lua, es muy fácil y flexible, con compatibilidad garantizada con todas las demás instalaciones CoppeliaSim predeterminadas. Permite personalizar una simulación en particular, una escena de simulación en particular, una escena de simulación y, en cierta medida, el propio simulador. Es el enfoque de programación más fácil y más utilizada.

Plugins: este método consiste básicamente en escribir un *plugin* para CoppeliaSim. A menudo, los *plugins* solo se usan para proporcionar una simulación con comandos Lua personalizados, por lo que se usan junto con el primer método. Otras veces, los complementos se utilizan para proporcionar a CoppeliaSim una funcionalidad especial que requiere una capacidad de cálculo rápido, una interfaz específica para un dispositivo de hardware o una interfaz de comunicación con el mundo exterior.

Cliente API remoto: este método permite que una aplicación externa se conecte a CoppeliaSim de una manera muy fácil, utilizando comandos de API remotos.

Nodo BlueZero: este método permite que una aplicación externa se conecte a CoppeliaSim a través de BlueZero.

Nodo ROS: este método permite que una aplicación externa se conecte a Coppelia sim a través de ROS, el sistema operativo del robot.

Complementos (*add-on*): este método, que consiste en escribir *scripts* Lua, permite personalizar rápidamente el propio simulador. Los complementos pueden iniciarse automáticamente y ejecutarse en segundo plano, o pueden llamarse como funciones. Los complementos no deben ser específicos de una determinada simulación o modelo, sino que deben ofrecer una funcionalidad más genérica, vinculada al simulador.

En la siguiente tabla se muestran las ventajas y desventajas de cada método:

	Embedded script	Add-on / sandbox script	Plugin	Remote API client	ROS node	BlueZero node
Control entity is external (i.e. can be located on a robot, different machine, etc.)	No	No	No	Yes	Yes	Yes
Difficulty to implement	Easiest	Easiest	Relatively easy	Easy	Relatively easy	Relatively easy
Supported programming language	Lua	Lua	C/C++	C/C++, Python, Java, Matlab, Octave, Lua	Any [1]	C++
Simulator functionality access (available API functions)	500+ functions, extendable	500+ functions, extendable	500+ functions	>100 functions + indirectly all embedded script functions	Indirectly all embedded script functions	Indirectly all embedded script functions
The control entity can control the simulation and simulation objects (models, robots, etc.)	Yes	Yes	Yes	Yes	Yes	Yes
The control entity can start, stop, pause and step a simulation	Start, stop, pause	Start, stop, pause	Start, stop, pause, step	Start, stop, pause, step	Start, stop, pause, step	Start, stop, pause, step
The control entity can customize the simulator	Yes	Yes	Yes	No	No	No
Code execution speed	Relativ. slow [2] (fast with JiT compiler)	Relativ. slow [2] (fast with JiT compiler)	Fast	Depends on programming language	Depends on programming language	Fast
Communication lag	None	None	None	Yes, reduced	Yes, reduced	Yes, reduced
Control entity is fully contained in a scene or model, and is highly portable	Yes	No	No	No	No	No
API mechanism	Regular API	Regular API	Regular API	B0-based remote API, or Legacy remote API	ROS	BlueZero
API can be extended	Yes, with custom Lua functions	Yes, with custom Lua functions	Yes, CoppeliaSim is open source	Yes, Remote API is open source	Yes, via embedded scripts	Yes, via embedded scripts
Control entity relies on	CoppeliaSim	CoppeliaSim	CoppeliaSim	BlueZero framework, or sockets	ROS framework	BlueZero framework
Synchronous operation [3]	Yes, inherent. No delays	Yes, inherent. No delays	Yes, inherent. No delays	Yes. Slower due to comm. Lag	Yes. Slower due to comm. Lag	Yes. Slower due to comm. Lag
Asynchronous operation [3]	Yes, via threaded scripts	No	No (threads available, but API access forbidden)	Yes, default operation mode	Yes, default operation mode	Yes, default operation mode

[1] Depends on ROS binding
[2] The execution of API functions is however very fast. Additionally, there is an optional JiT (Just in Time) compiler option that can be activated
[3] *Synchronous* in the sense that each simulation pass runs synchronously with the control entity, i.e. simulation step by step

Figura 1. Los 6 posibles métodos de control

La siguiente figura ilustra las diversas posibilidades de personalización en CoppeliaSim:

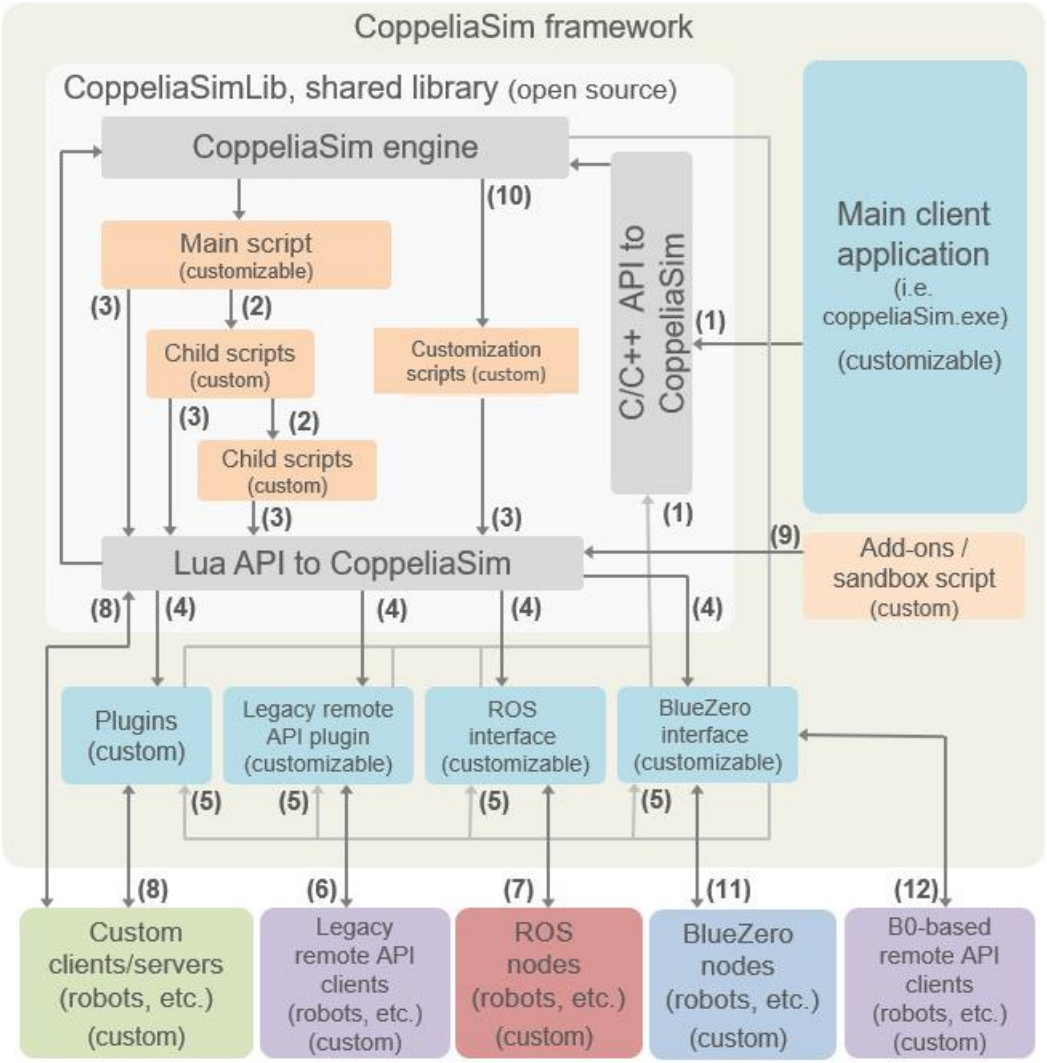

Figura 2. Posibilidades de personalización

2.2.1.2. Sistema dinámico

El módulo dinámico de CoppeliaSim admite cuatro motores físicos diferentes. En cualquier momento el usuario puede cambiar de un motor a otro de acuerdo con sus necesidades en la simulación. La razón de esta diversidad en el soporte de motores físicos es que la simulación es una tarea compleja, que se puede lograr con varios grados de precisión, velocidad o con soporte de diversas características:

-Librería Bullet physics: un motor físico de código abierto con detección de colisiones 3D, dinámica de cuerpo rígido y blando. Se usa en juegos y en efectos visuales en películas.

Figura 3. Bullet Physics

-Open Dynamics Engine (ODE): un motor físico de código abierto con dos componentes principales; dinámica de carrocería rígida y detección de colisiones. Se ha utilizado en muchas aplicaciones y juegos.

Figura 4. Open Dynamics Engine

-Vortex Studio: un motor físico comercial de código cerrado que produce simulaciones de física de alta fidelidad. Ofrece parámetros del mundo real para una gran cantidad de propiedades físicas, lo que hace que este motor sea realista y preciso. Vortex se utiliza principalmente en aplicaciones industriales y de investigación de alto rendimiento/precisión.

Figura 5. Vortex Studio

-Newton Dynamics: es una librería de simulación física realista multiplataforma. Implementa un solucionador determinista, que no se basa en métodos tradicionales LCP o iterativos, pero posee la estabilidad y la velocidad de ambos respectivamente. Esta característica convierte a Newton Dynamics en una herramienta no solo para juegos, sino también para cualquier simulación de física en tiempo real.

Figura 6. Newton Dynamics

El módulo dinámico de CoppeliaSim permite simular interacciones entre objetos cercanas a las del mundo real. Permite que los objetos caigan, colisionen, reboten, pero también permite que un robot manipulador agarre objetos, una cinta transportadora

pueda impulsar piezas hacia delante o que un vehículo ruede de manera realista sobre un terreno irregular. Las siguientes figuras ilustran una simulación dinámica:

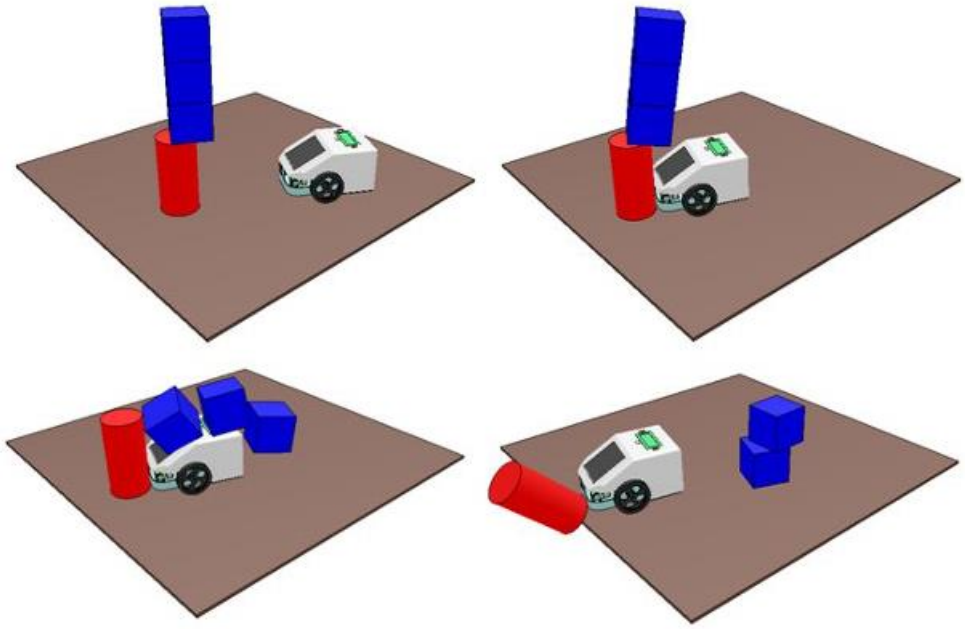

Figura 7. Simulación dinámica

2.2.1.3. Diseño de simulaciones dinámicas

En CoppeliaSim solo se simulará dinámicamente un número limitado de objetos. Los cuales son las formas, articulaciones y sensores de fuerza, pero dependerá de la estructura de la escena y las propiedades del objeto. Los objetos simulados dinámicamente son fácilmente reconocibles ya que aparecerá el siguiente icono junto al nombre del objeto en la jerarquía de escenas:

Figura 8. Icono de marcado de objetos dinámicamente simulados

Las formas se pueden clasificar en 4 grupos según su comportamiento durante la simulación dinámica:

	Static	Non-static
Non-respondable	☒	☐
Respondable	☒	☐

Figura 9. Formas principales de simulación dinámica

Durante la simulación dinámica, las formas estáticas no se verán afectadas, es decir, su posición relativa a su objeto principal es fija, mientras que las formas no estáticas se verán influenciadas directamente por la gravedad u otras restricciones. Las formas reactivas se influyen entre ellas durante una colisión dinámica. La siguiente figura ilustra los comportamientos estáticos/no estáticos, reactivos/no reactivos:

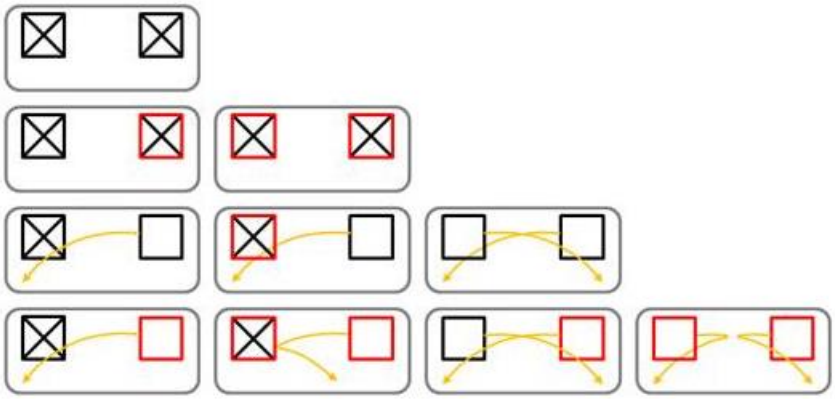

Figura 10. Interacciones según su comportamiento

2.2.1.4. Cinemática inversa

El módulo de cálculo de cinemática inversa de CoppeliaSim es muy potente y flexible. Permite manejar prácticamente cualquier tipo de mecanismo en modo de cinemática inversa o directa.

2.2.1.5. Interfaz de usuario

Los principales elementos por los que está compuesta la aplicación CoppeliaSim son:

-Una ventana de consola:

En Windows, se crea una ventana de consola al iniciar la aplicación pero se vuelve a ocultar directamente.

La ventana de la consola no es interactiva y solo se usa para generar información. El usuario puede enviar información directamente a la ventana de la consola con el comando Lua "*print*".

-Una ventana de la aplicación:

Es la ventana principal. Se utiliza para mostrar, editar, simular e interactuar con la escena.

-Cuadros de diálogo:

Junto a la ventana de la aplicación el usuario también puede editar e interactuar con una escena ajustando la configuración o los parámetros del cuadro de diálogo. Cada cuadro de diálogo agrupa un conjunto de funciones relacionadas o funciones que se aplican a un mismo objeto.

A continuación se ilustra una vista típica de la aplicación CoppeliaSim:

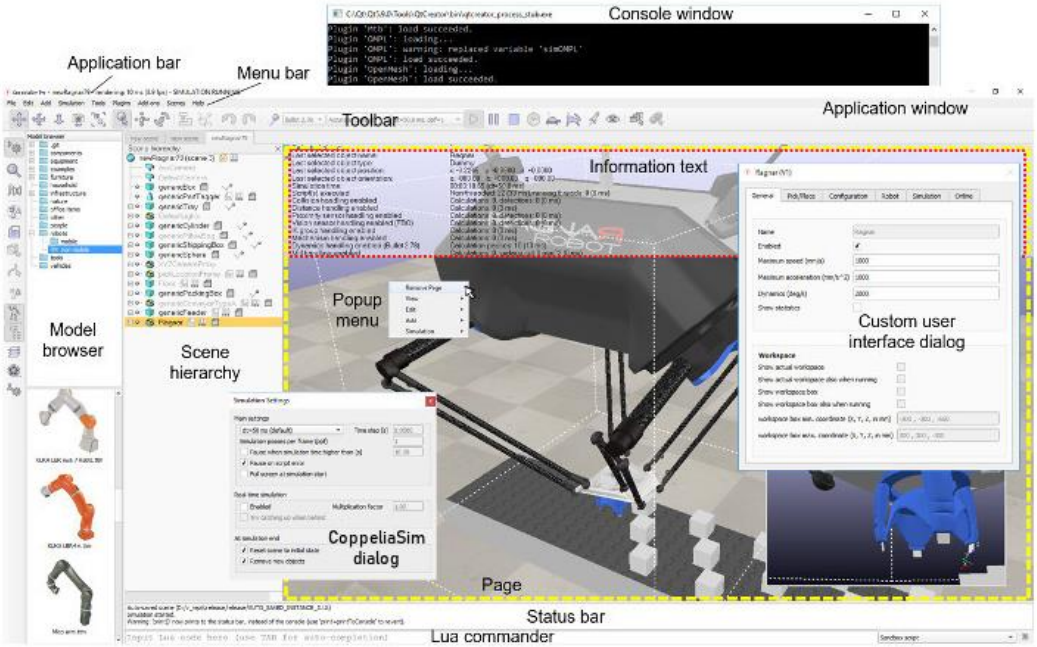

Figura 11. Elementos de la interfaz de usuario

Al ejecutar la aplicación CoppeliaSim se inicializará una escena predeterminada. El usuario es libre de abrir varias escenas en paralelo. Cada escena comparte la ventana de la aplicación y los diálogos con las otras escenas, pero solo el contenido de la escena activa será visible en la ventana de la aplicación.

Los elementos más importantes de la ventana de aplicación son los siguientes:

-Barra de aplicación:

Indica el tipo de licencia de su copia de CoppeliaSim, el nombre de archivo de la escena que se está mostrando en el momento actual, el tiempo utilizado para una pasada de renderizado y el estado actual del simulador.

La barra de aplicación así como cualquier superficie dentro de la ventana de aplicación, también se puede utilizar para arrastrar y soltar archivos relacionados con CoppeliaSim en la escena.

-Barra de menú:

Permite acceder a casi todas las funcionalidades del simulador. La mayoría de las veces, los elementos de la barra de menú activan un diálogo.

-Barras de herramientas:

Presentan funciones a las que se accede con frecuencia. La siguiente figura explica la función de cada botón de la barra de herramientas:

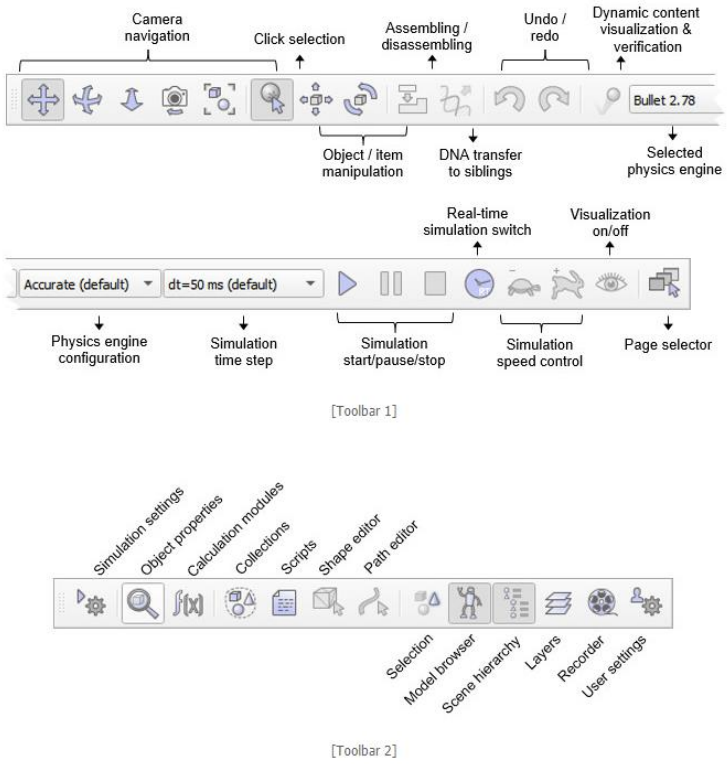

Figura 12. Funciones de la barra de herramientas

-Navegador de modelos:

Muestra en su parte superior una estructura de carpetas del modelo CoppeliaSim, y en su parte inferior, miniaturas de los modelos contenidos en la carpeta seleccionada. Las miniaturas se pueden arrastrar y soltar en la escena para cargar automáticamente el modelo relacionado.

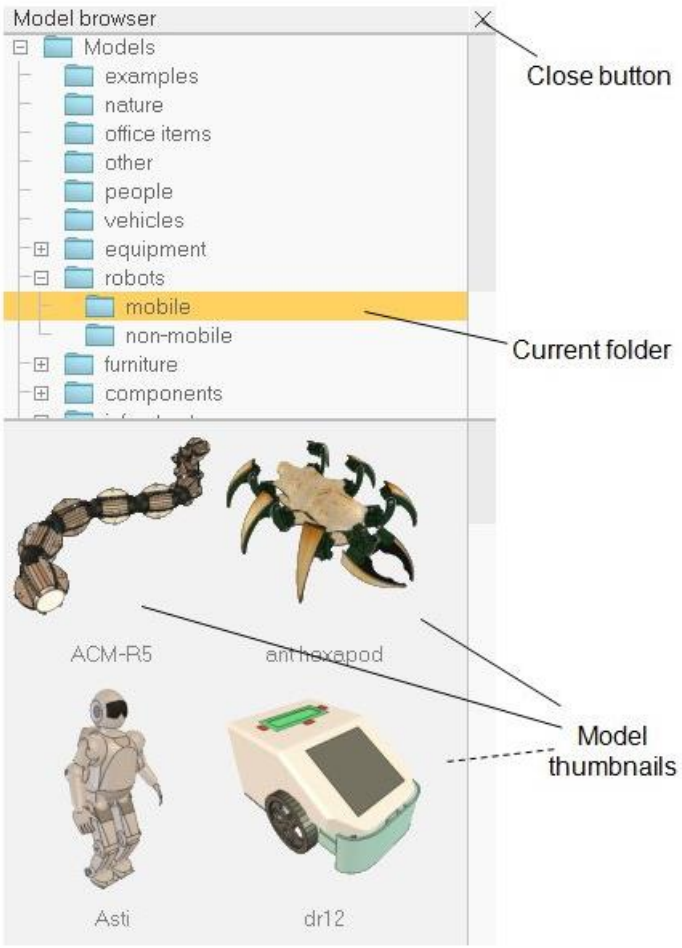

Figura 13. Navegador de modelos

-Jerarquía de escenas:

Muestra el contenido de una escena, es decir, todos los objetos que componen una escena. Dado que los objetos de una escena se construyen en una estructura similar a una jerarquía, la cual muestra los elementos individuales los cuales se pueden expandir o contraer.

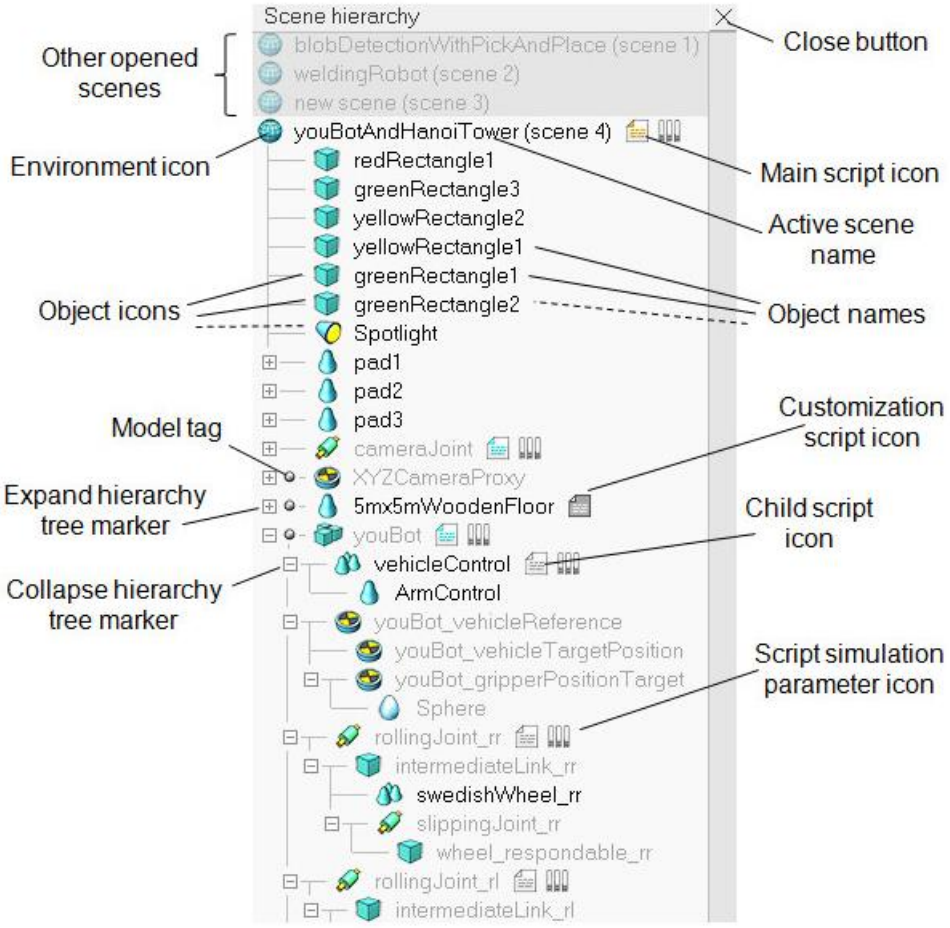

Figura 14. Jerarquía de escenas

Los objetos de la jerarquía de escenas se pueden arrastrar y soltar sobre otro objeto para crear una relación entre padres e hijos.

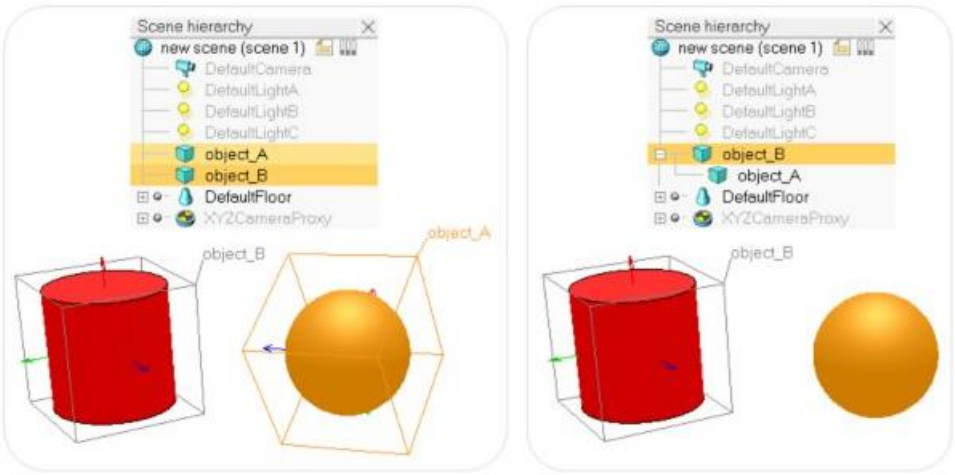

Figura 15. Relación padre-hijo

Cuando un objeto se convierte en el hijo de otro este se moverá a la par que su padre debido a que están unidos. Sin embargo si el objeto que se decide mover es el hijo el padre no se verá afectado.

2.2.1.6. Elementos de la escena

Los principales elementos que componen una escena son llamados objetos. Existen multitud de tipos diferentes de objetos, cada cual con sus características propias. Los más importantes se muestran en la siguiente figura:

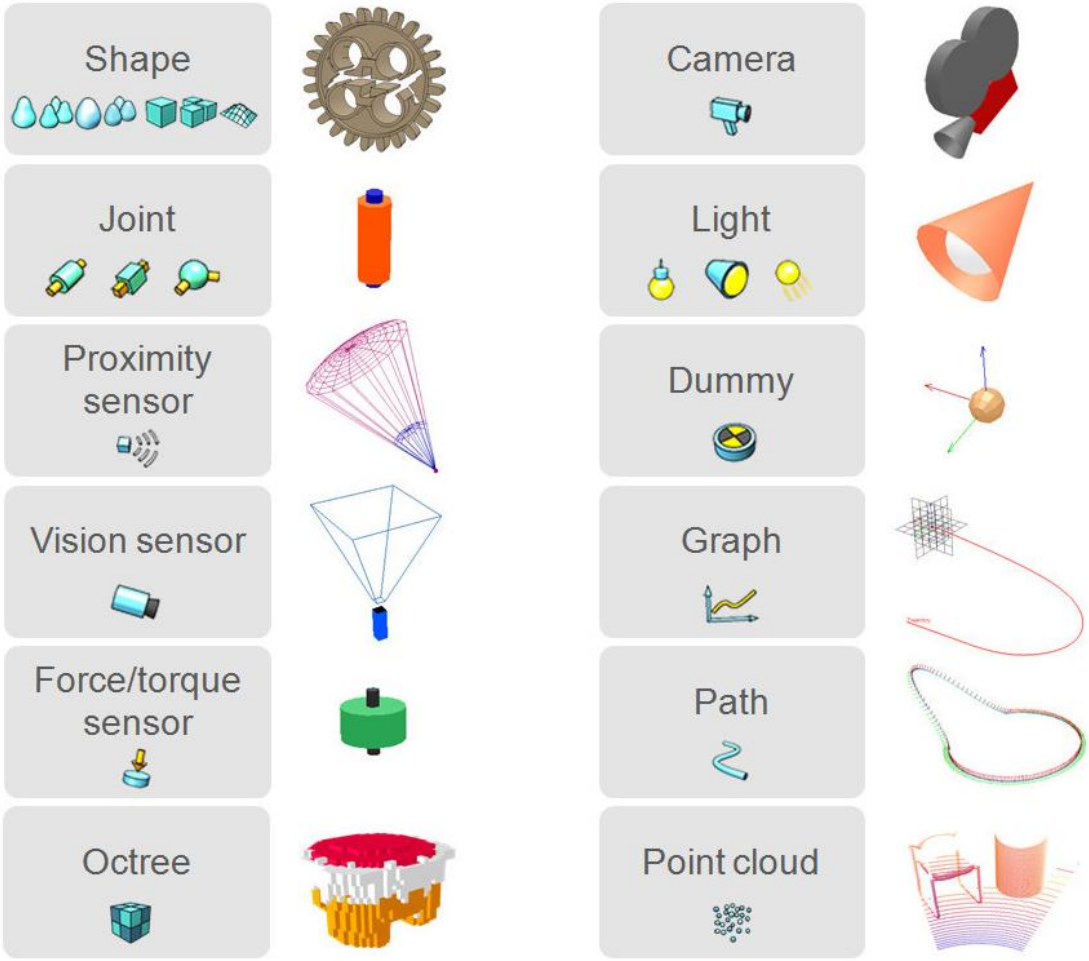

Figura 16. Tipos de objetos

A continuación se procede a dar una pequeña explicación sobre cada uno:

-Formas: malla rígida que se compone de caras triangulares.

-Articulaciones: existen del tipo junta prismática, junta angular, junta esférica y tornillo.

-Gráficos: registra y visualiza datos de la simulación.

-*Dummies*: punto con orientación.

-Cámaras: permite ver la escena de simulación desde varios puntos de vista.

-Sensores de proximidad: detecta objetos de una forma geométricamente exacta. Existen del tipo pirámide, cilindro, disco, cono y rayo.

-Sensores de visión: tipo de cámara que reacciona a la luz, los colores y las imágenes.

-Sensores de fuerza: capaz de medir las fuerzas y pares que se le aplican. Tiene la característica de romperse si se sobrepasa un umbral determinado.

-Camino: define una trayectoria en el espacio.

Gran parte de los objetos nombrado anteriormente que pueden tener características especiales las cuales les permiten interactuar con otros objetos o módulos de cálculo. Dichas características especiales son:

-Colisionable: permite que se puedan detectar colisiones entre objetos que tengan esta propiedad.

-Medible: autoriza el cálculo de distancias mínimas con otros objetos medibles.

-Detectables: pueden ser detectados por sensores de proximidad.

-Renderizable: pueden ser vistos o detectados por sensores de visión.

-Visible: su contenido puede ser visualizado.

2.3. Diseño del entorno industrial

Una vez ya explicadas las características del software utilizado para el desarrollo del proyecto se procede a mostrar los detalles de este.

La simulación consistirá en la implementación de un entorno industrial con la finalidad de la clasificación de determinados objetos los cuales dependiendo de sus características serán considerados como buenos y la eliminación de los objetos restantes considerados como erróneos.

Para crear dicha simulación se utilizarán los siguientes elementos:

4 cintas transportadoras

7 sensores de posición

1 sensor de color

1 sensor de visión

1 robot manipulador

1 pinza adaptativa

2.3.1. Creación de la escena

Una vez distribuidos los diferentes elementos de las escena y conociendo sus dimensiones y posiciones relativas en la escena, se procede a la creación paso a paso de la escena.

El primer paso será el posicionamiento del robot industrial KUKA LBR iiwa 14 R820 el cual se representa en CoppeliaSim de la siguiente manera:

Figura 17. Simulación KUKA LBR iiwa 14 R820

Se situará en la posición x: -0.775, y: 0.1, z: 0.075 y con una orientación de alpha: 0°, beta: 0°, y gamma: -45°.

Una vez posicionado el robot industrial, se le añadirá una pinza mediante la cual se cogerán los diferentes objetos de la escena.

Para realizar esto hay que seguir una determinada serie de pasos. Primero se elegirá la pinza que se quiere acoplar el robot industrial. El modelo más adecuado para la simulación es el ROBOTIQ 2F-85 disponible en la versión EDU de CoppeliaSim para su simulación:

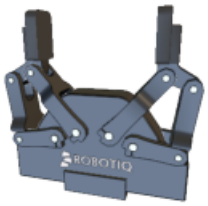

Figura 18. Simulación Pinza ROBOTIC 2F-85

A continuación se ajustará la posición y orientación de la pinza para que quede perfectamente acoplada al robot. Posteriormente se conectará la pinza con el último eje del robot consiguiendo así una relación padre-hijo, quedando la jerarquía de la siguiente forma:

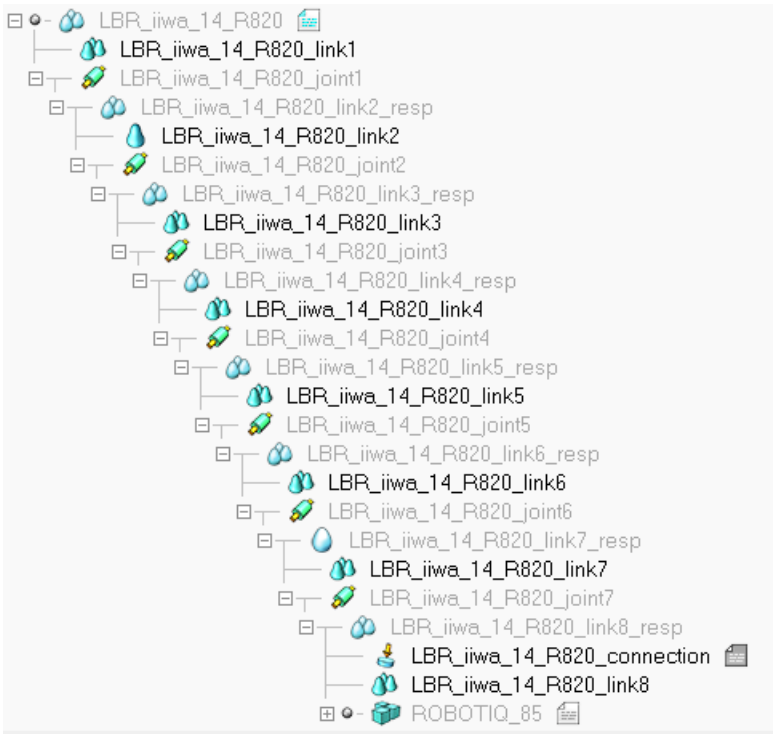

Figura 19. Jerarquía de escenas

Seguidamente se ajustarán las características del objeto haciendo doble clic sobre el símbolo situado a la izquierda del nombre de la pinza en la jerarquía de escenas. Al realizar esta acción se abrirá la siguiente ventana emergente:

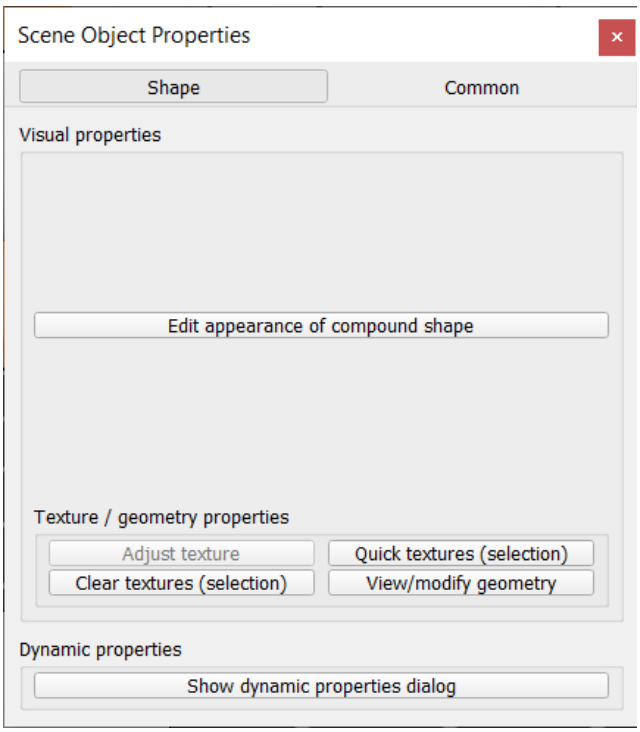

Figura 20. Propiedades pinza

Al pulsar en el botón de la parte inferior en el que pone "*Show dynamic properties dialog*" se abrirá otra ventana emergente en la que se debe desmarcar la casilla de *"Body is dynamic*" como aparece en la siguiente imagen:

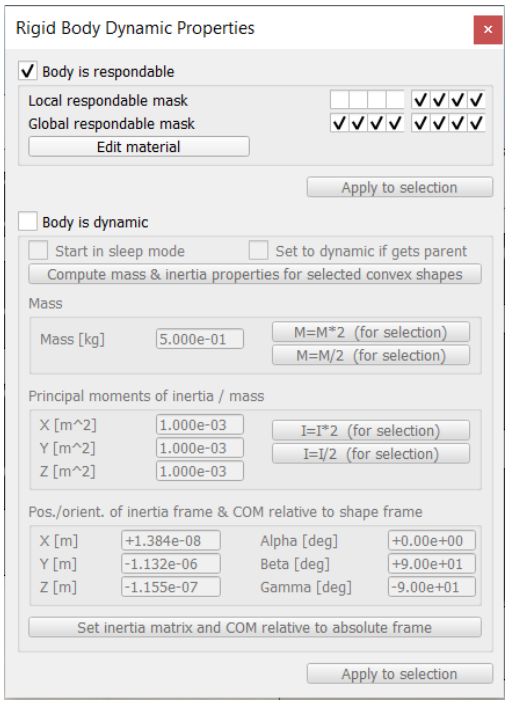

Figura 21. Propiedades dinámicas pinza

De esta forma la pinza quedará completamente equipada al robot industrial, como se muestra en la siguiente imagen:

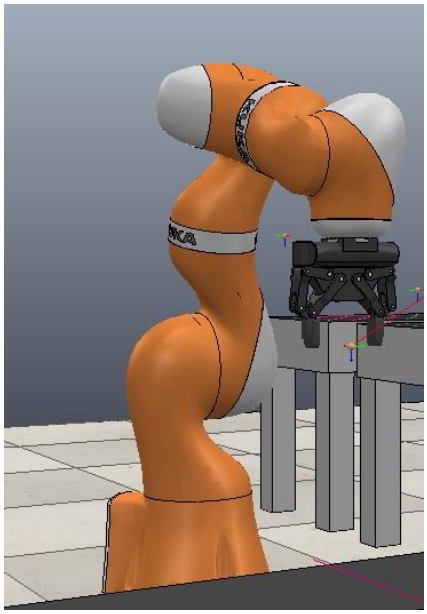

Figura 22. Pinza acoplada al robot industrial

El siguiente paso es crear una configuración en la que la pinza sea capaz de dirigirse a la posición deseada en cualquier momento.

Para conseguir esto se crean tres puntos en el espacio (*Dummies*) los cuales se llamarán connector, tip y target. Ahora se conectará cada *dummy* en el lugar correspondiente quedando de la siguiente manera:

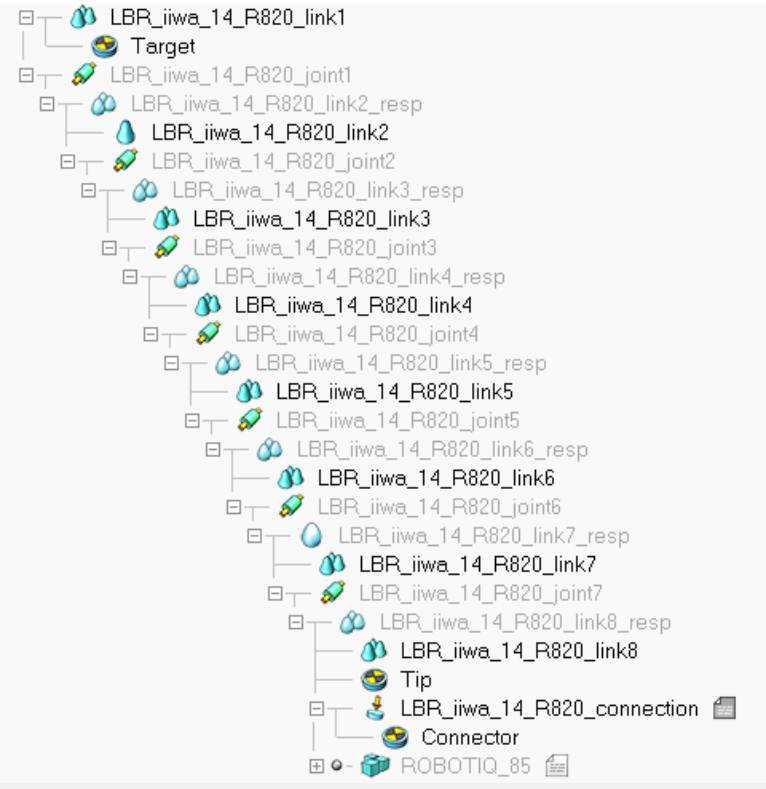

Figura 23. Jerarquía de escenas

El siguiente paso será hacer doble clic en el icono a la izquierda del elemento llamado "Tip", al hacer esto aparecerá la siguiente ventana emergente:

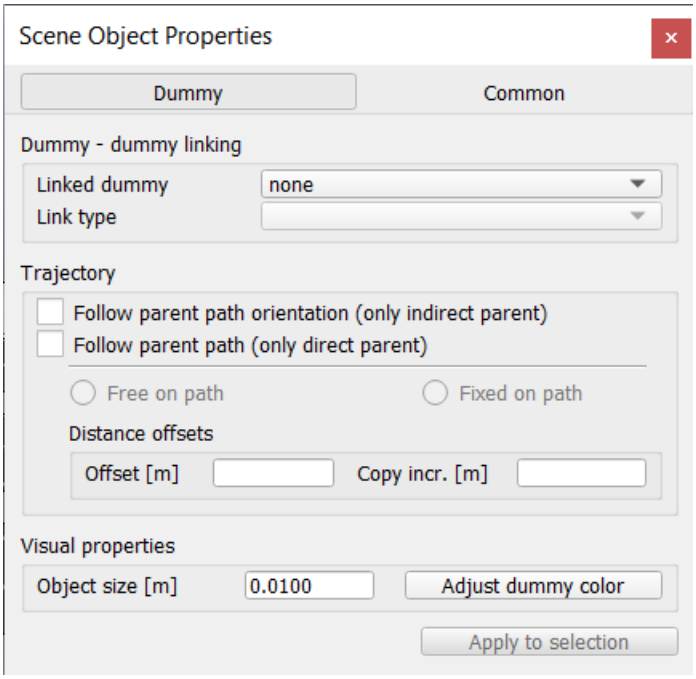

Figura 24. Propiedades dummy

Se deberán modificar las dos pestañas superiores, en la primera se seleccionará al *dummy* al que se quiere unir, en la segunda se definirá el tipo de unión realizado. Una vez configuradas estas dos pestañas la ventana emergente quedará de la siguiente manera:

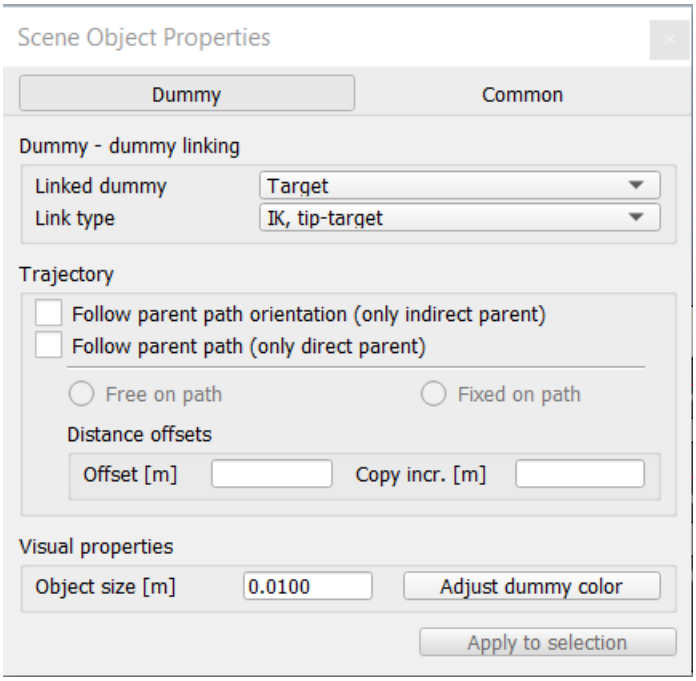

Figura 25. Conexión dummies

Una vez configurado estos últimos parámetros se podrá observar la aparición de una flecha vertical roja la cual unirá los elementos tip y target, quedando de la siguiente manera:

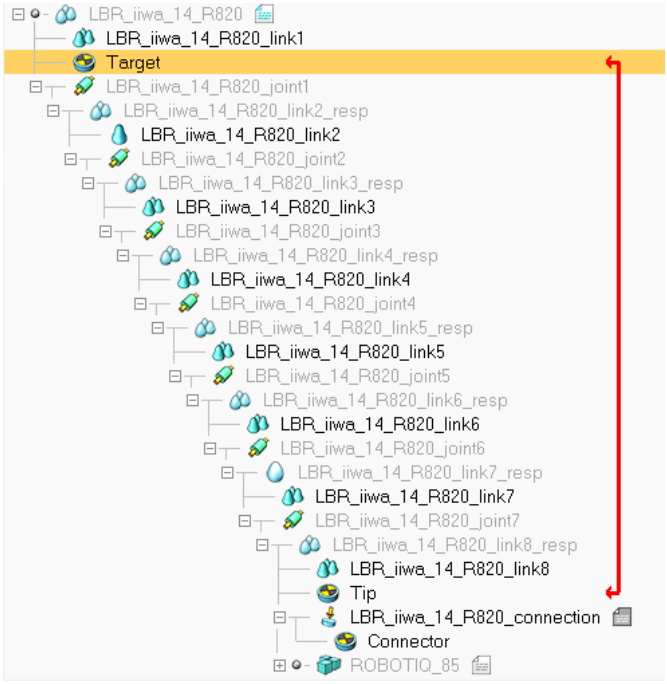

Figura 26. Conexión de dummies en jerarquía de escenas

Una vez hecho esto debemos acudir a la barra de aplicación y pulsar sobre el botón llamado "*calculation module propertis*" representado mediante el símbolo  . Al acceder a este menú se abrirá una ventana emergente como la siguiente:

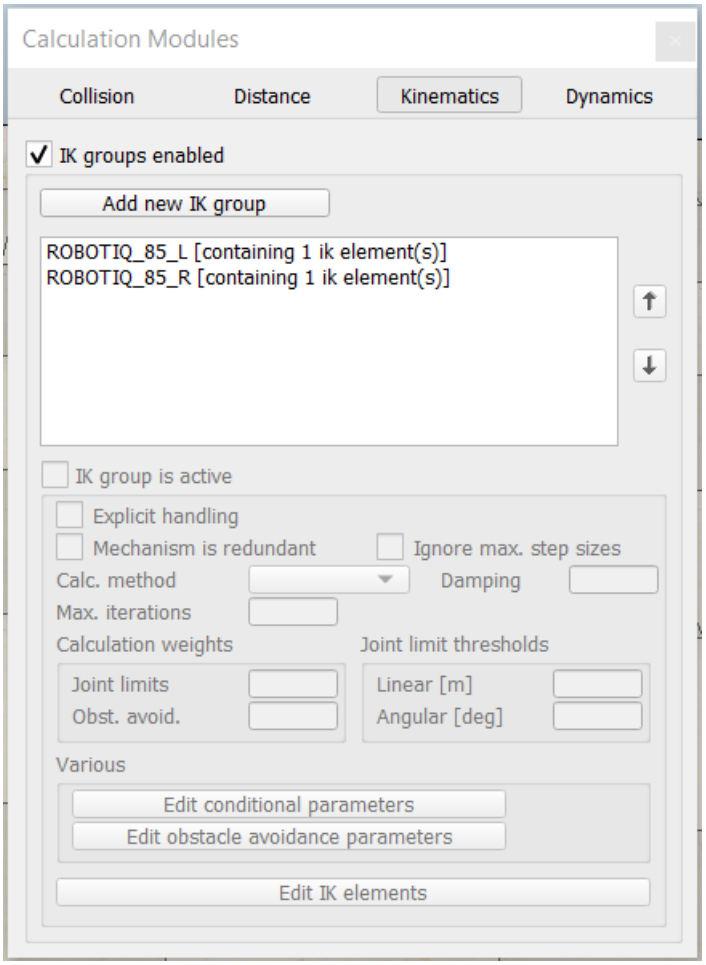

Figura 27. Propiedades del módulo de cálculo

Se deberá pulsar el botón de la parte superior "*Add new IK group*" y se modificará su nombre a "KUKA" quedando la ventana de la siguiente forma:

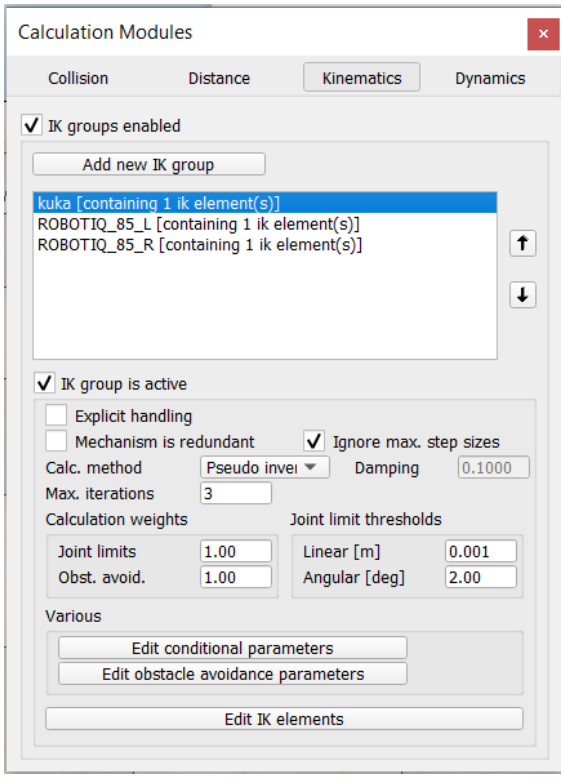

Figura 28. Propiedades del módulo de cálculo modificado

No se modificará ningún valor de la ventana mostrada anteriormente. Por último acudiremos al botón inferior "*Edit IK elements*" mediante el cual se mostrará una nueva ventana auxiliar:

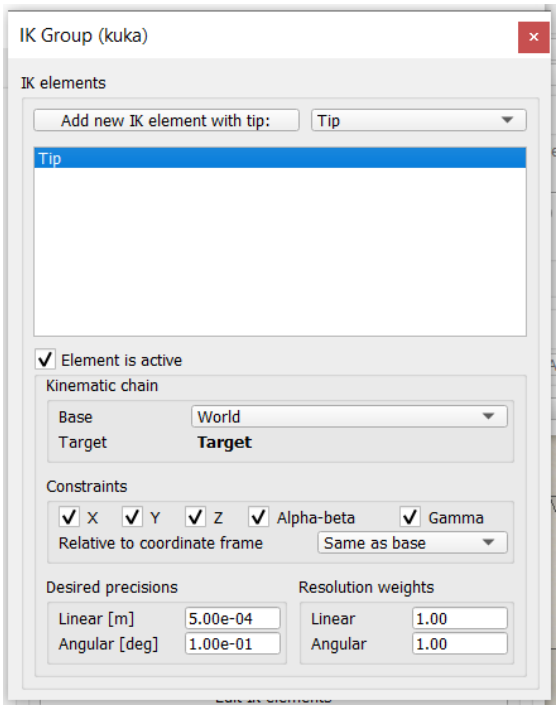

Figura 29. Edit IK elements

En dicha ventana se deberá seleccionar el elemento "Tip" en el desplegable superior derecho y seguidamente pulsar el botón "*Add new IK element with tip:*". Por último se

puede observar que por defecto los botones “Alpha – beta” y “Gamma” están desactivados. Estos botones son los encargados de modificar la orientación del robot industrial para alcanzar el elemento deseado, por tanto deberán estar activados. Si solo están activados los botones “X”, “Y” y “Z” el robot industrial alcanzará la posición deseada pero no modificará su orientación.

A continuación se establecerán los valores de la posición de los ejes del robot para su correcto funcionamiento.

Para realizar este proceso se debe hacer doble clic sobre el icono, al hacer esto emergerá la siguiente ventana:

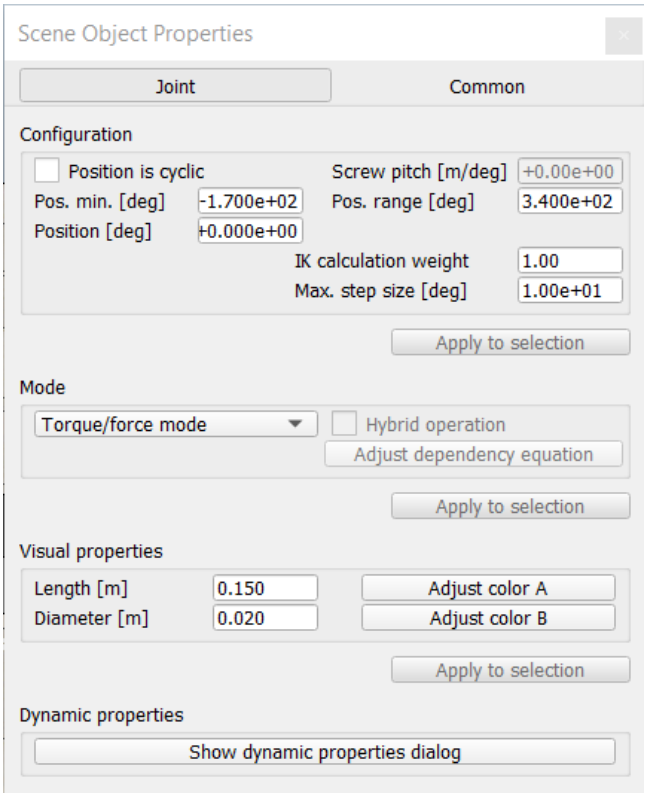

Figura 30. Propiedades eje

Como se puede observar el modo predeterminado en el que aparece es “*Torque/force mode*” el cual se deberá cambiar por “*Inverse kinematics mode*”. Además se deberá insertar el valor anteriormente calculado en la casilla superior de la ventana emergente en el apartado “Position[deg]”. Una vez cambiados estos valores la ventana emergente deberá quedar de la siguiente manera:

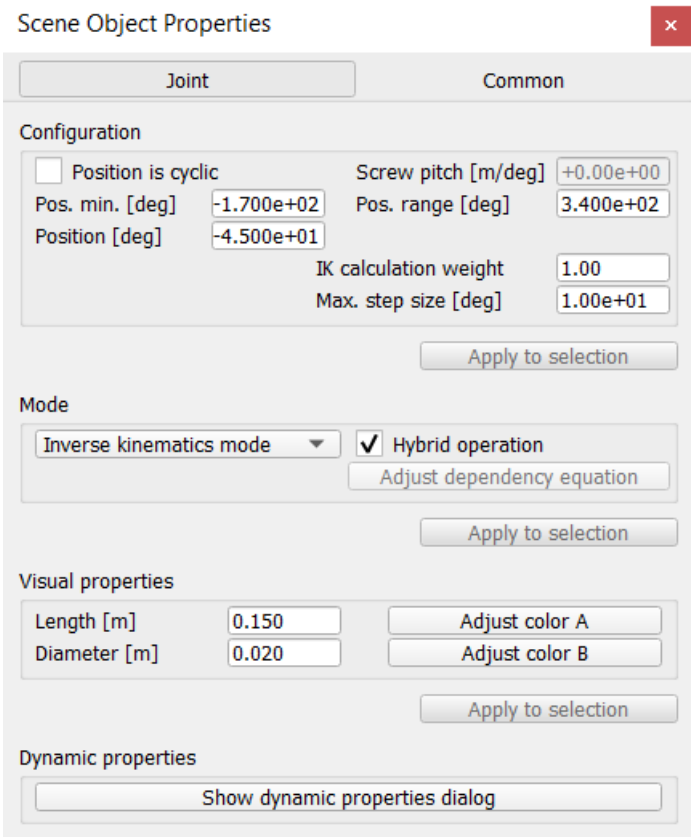

Figura 31. Configuración Eje 1

Los valores de los ejes son los siguientes:

Eje 1	-45º
Eje 2	14.7º
Eje 3	0º
Eje 4	-61.38º
Eje 5	0º
Eje 6	103.9º
Eje 7	0º

Tabla 1. Posición ejes robot

Una vez finalizada completamente la configuración del robot industrial y su correspondiente pinza, se procede a la inserción de las cintas transportadoras. Se utilizarán dos tipos de cintas transportadoras.

La primera cinta transportadora se simulará mediante la cinta simulada cuya nomenclatura es “Conveyor Belt”, cuya representación en el navegador de modelos es la siguiente:

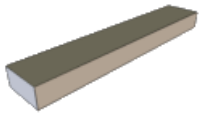

Figura 32. Conveyor belt

Para el correcto funcionamiento de la simulación se situará dicha cinta en la posición x: -1.375, y: -0.5, z: 0.05 y con una orientación de alpha: 0°, beta: 0°, y gamma: 0°.

Seguidamente se procede a la inserción de las otras tres cintas transportadoras. Dichas cintas tendrán las mismas características en cuanto a tamaño. Esta vez se utilizará la cinta transportadora denominada "Customizable Conveyor" cuya representación es la siguiente:

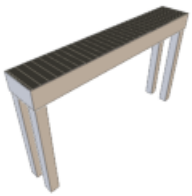

Figura 33. Customizable conveyor

Las diferentes cintas tendrán los siguientes posicionamientos.

La primera cinta; x: 0.2747 y: -0.2249 z: 0.3598 con una orientación de alpha: 0°, beta: 0°, y gamma: 0°.

La segunda cinta; x: 0.2747 y: 0.1251 z: 0.3598 con una orientación de alpha: 0°, beta: 0°, y gamma: 0°.

La tercera cinta; x: 0.2747 y: 0.4751 z: 0.3598 con una orientación de alpha: 0°, beta: 0°, y gamma: 0°.

Cada cinta se encargará de deslizar sobre sí misma un tipo de pieza diferenciada por su color. La primera cinta recibirá piezas de color azul, la segunda de color verde y la tercera de color rojo.

Cada una de dichas cintas contará con dos sensores de posición los cuales se encargarán de medir la posición de las piezas. Dichos sensores serán de tipo "*Ray*" y estarán situados al comienzo y al final de cada una de las cintas transportadoras. Además la primera cinta del proceso ("*Conveyor Belt*") también contará con uno de estos sensores en la parte posterior de esta, el cual se encargará de detectar las piezas idóneas y de desechar las erróneas. Dicho sensor se encuentra en la barra de aplicación entrando en la pestaña de "*Add*", como se muestra en la siguiente imagen:

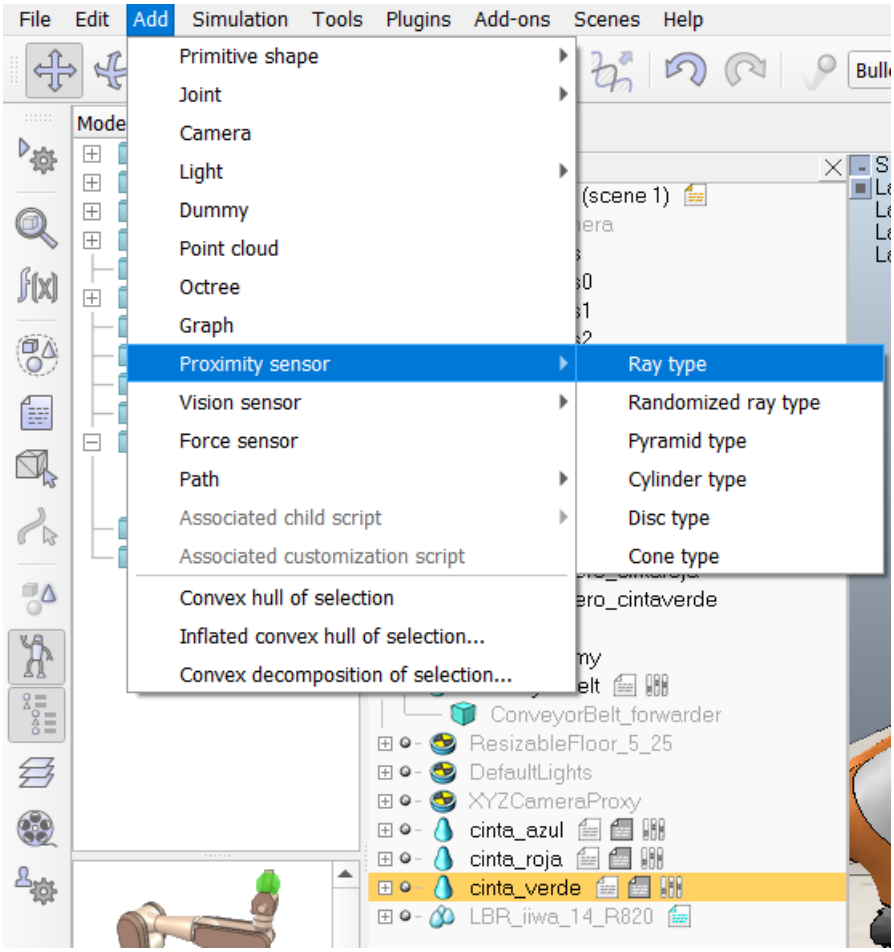

Figura 34. Inserción sensor de proximidad

Las posiciones y orientaciones de los distintos sensores son las siguientes:

Sensor trasero "*Converyor Belt*": x: -0.8 y: -0.85 z: 0.13 alpha: -90°, beta: 0°, y gamma: 0°.

Sensor delantero primera cinta (roja): x: -0.1013 y: -0.4339 z: 0.4148 alpha: -90°, beta: 0°, y gamma: -90°.

Sensor trasero primera cinta (roja): x: -0.7998 y: -0.5750 z: 0.4149 alpha: -90°, beta: 0°, y gamma: 0°.

Sensor delantero segunda cinta (verde): x: -0.172 y: -0.0749 z: 0.4148 alpha: -90°, beta: 0°, y ganma: -90°.

Sensor trasero segunda cinta (verde): x: -0.7981 y: -0.1001 z: 0.4141 alpha: -90°, beta: 0°, y gamma: 0°.

Sensor delantero tercera cinta (azul): x: -0.172 y: -0.8251 z: 0.4147 alpha: 90°, beta: 0°, y gamma: 0°.

Sensor trasero tercera cinta (azul): x: -0.7981 y: -0.8248 z: 0.4141 alpha: 90°, beta: 0°, y gamma: 0°.

Para finalizar con los elementos de medición se procede a insertar el sensor de visión el cual se encargará de proporcionar la información de la orientación y la posición de las piezas que se deslizarán a través de la cinta "*Conveyor Belt*".

Para insertar este sensor debemos acudir como en el caso anterior a la barra de aplicación y en el apartado "*Add*" como se muestra en la siguiente imagen:

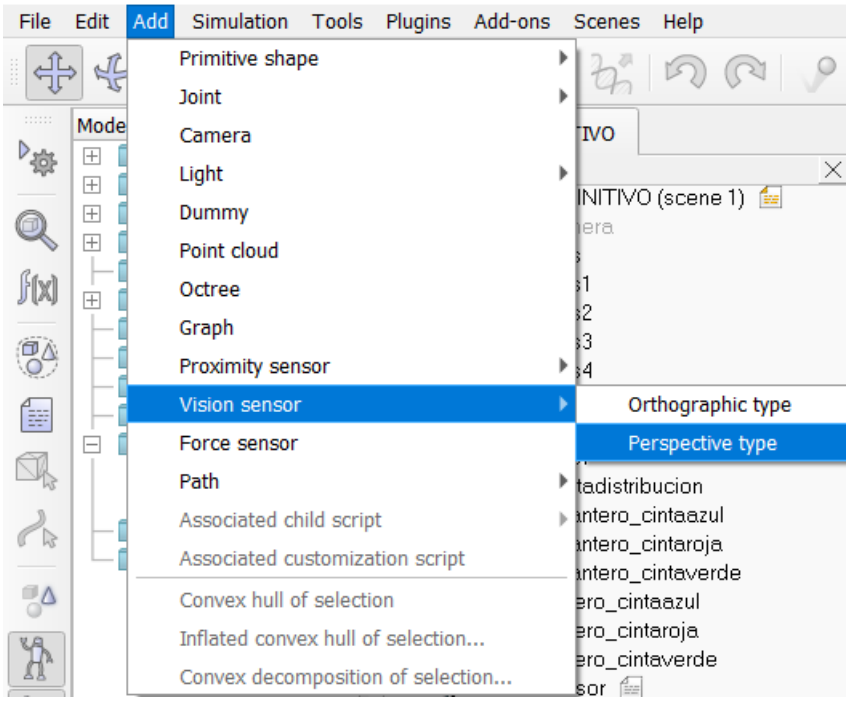

Figura 35. Inserción sensor de visión

Una vez introducido este sensor deberemos modificar alguna de sus características. Pulsando sobre el icono situado a la izquierda del elemento "Vision_sensor" en la jerarquía de escenas se abrirá una ventana emergente donde se deberá activar la opción "*Explicit handling*" y se cambiará el valor de "*Persp. Angle [deg]/ ortho. Size [m]*" a 45, quedando de la siguiente manera:

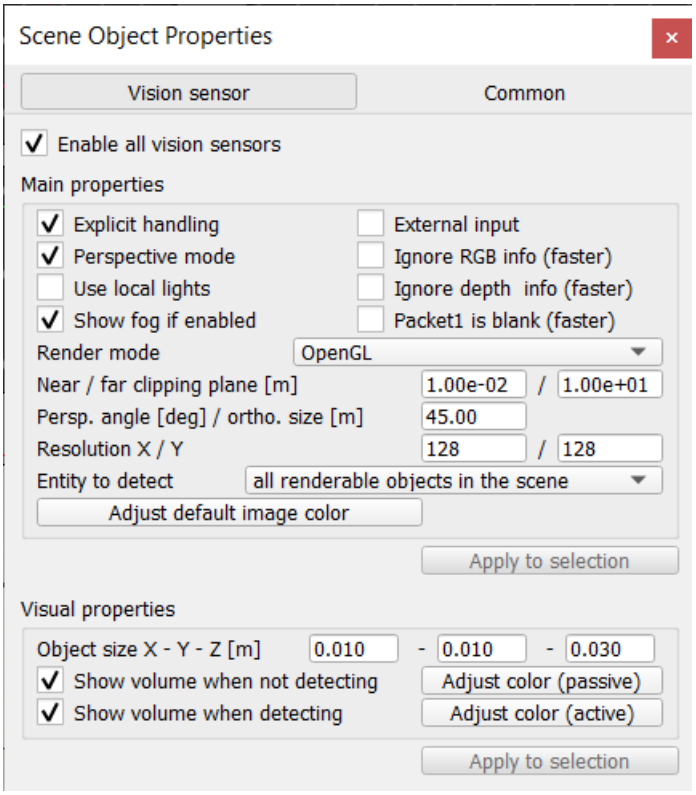

Figura 36. Propiedades sensor de visión

Continuando con el proceso de creación de los elementos de la escena se insertará un cuboide al cual denominaremos "Box" Para crear un cuboide se debe acceder al panel de la barra de aplicación. A través de la pestaña "*Add*", pulsando sobre "*Primitive shape*" y sobre la opción "*Cuboid*". En la siguiente imagen mostramos su selección:

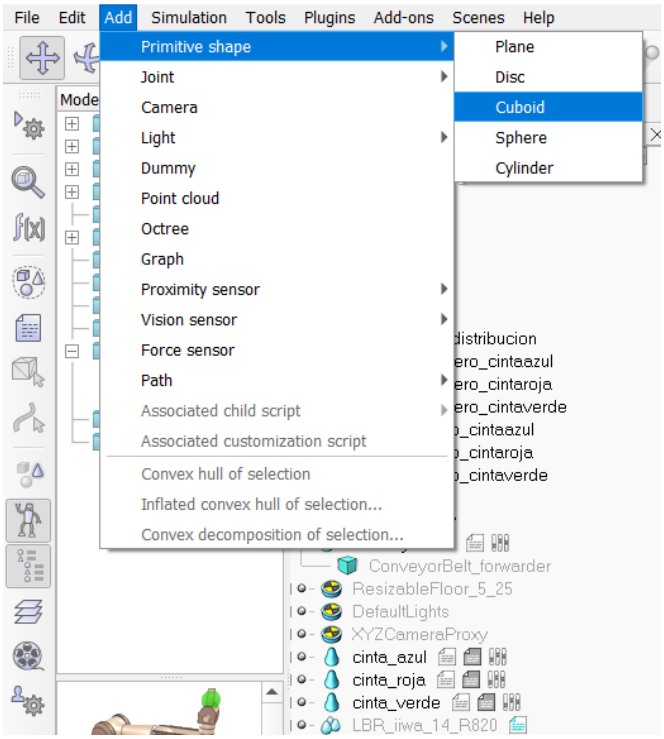

Figura 37. Inserción cuboide

Una vez creado el elemento se debe adaptar su estructura para un correcto funcionamiento de la simulación. Para ello se debe realizar doble clic sobre el icono situado a la izquierda del nombre del elemento recién creado. A continuación aparecerá la siguiente ventana emergente:

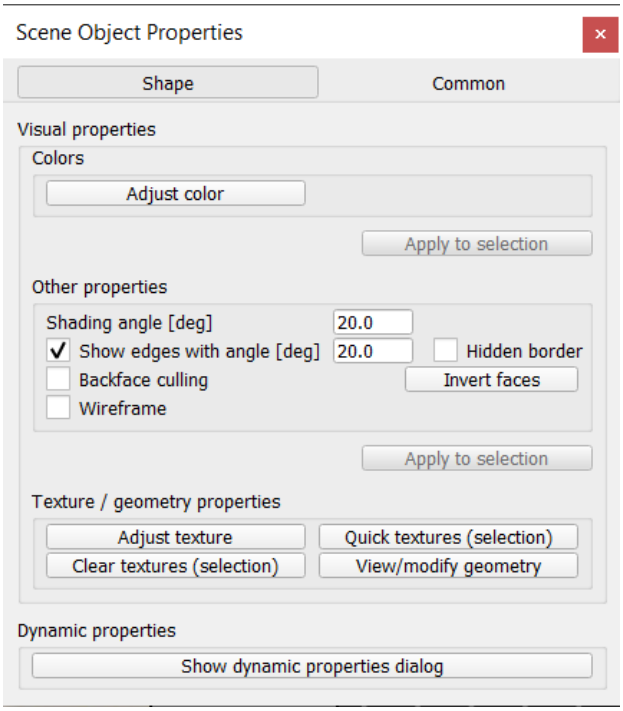

Figura 38. Propiedades cuboide

Se procede a cambiar la geometría del elemento a través de la pestaña situada en la parte inferior derecha, en la cual pone "*View/modify geometry*". Al pulsar dicha pestaña se verá el siguiente cuadro:

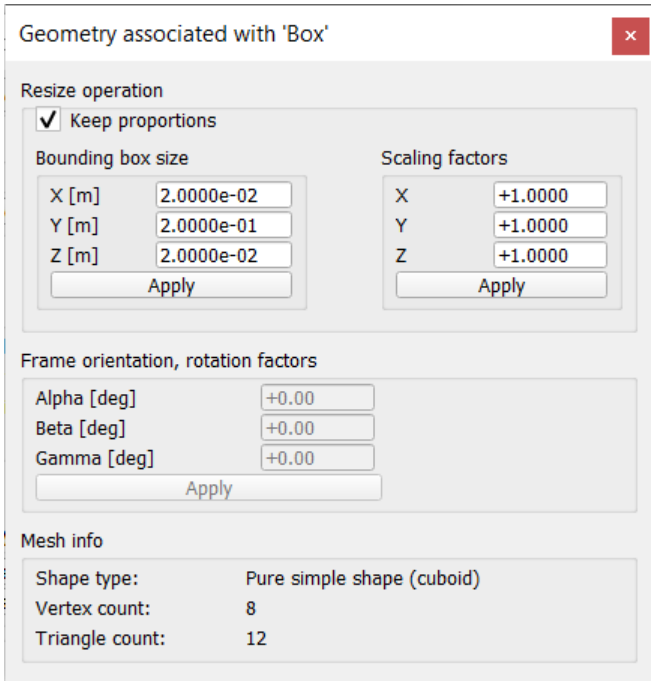

Figura 39. Modificación geometría cuboide

Las propiedades a modificar son el tamaño del elemento. Mediante los cuadros situados en la parte superior izquierda los cuales están en el apartado "*Bounding box size*" se deben introducir los valores x= 2e-02, y=2e-01, z=2e-02.

A continuación se añadirá un *dummy* situado en la parte central del elemento "*Box*" creado anteriormente. Esto proporcionará un lugar exacto donde la pinza del robot industrial lo cual facilitará en mayor medida la creación del código.

Para fijar la posición exacta a la que se dirigirá el robot cada vez que vaya a colocar una pieza se crearán una serie de *dummies* con tal de cumplir esta misión. La posición y la orientación de cada uno de estos *dummies* es la siguiente:

"releasePos": x: -0.27 y: 0.1251 z: 0.4748 alpha: -180º beta: 0º gamma: 180º.

"releasePos1": x: -0.27 y: -0.4001 z: 0.4748 alpha: -180° beta: 0° gamma: 180°.

"releasePos2": x: -0.27 y: -0.4751 z: 0.4148 alpha: -180° beta: 0° gamma: 180°.

"releasePos3": x: -0.27 y: -0.5501 z: 0.4148 alpha: -180° beta: 0° gamma: 180°.

"releasePos4": x: -0.324 y: -0.225 z: 0.4749 alpha: -180° beta: 0° gamma: -90°.

"releasePos5": x: -0.102 y: -0.225 z: 0.4949 alpha: -180° beta: 0° gamma: -90°.

"releasePos6": x: -0.215 y: -0.1380 z: 0.4749 alpha: -180° beta: 0° gamma: 180°.

"releasePos7": x: -0.215 y: -0.3150 z: 0.4749 alpha: -180° beta: 0° gamma: 180°.

Una vez introducidos todos los elementos anteriores la escena estará completa en cuanto a objetos en la escena se refiere. A continuación, se explicará el código introducido en el software CoppeliaSim con el cual se completará el correcto funcionamiento de la aplicación.

2.4. Programación del código

Para la simulación de este proyecto se ha utilizado el método de programación de *script*s embebidos debido a las numerables ventajas ya explicadas en la introducción de este documento.

En total contamos con 5 *scripts* los cuales se deberán programar para el funcionamiento óptimo del programa.

Para poder empezar a desarrollar los entresijos de los *scripts* primero se debe dar a conocer la nomenclatura de los elementos que intervienen en la escena.

Los nombres de los elementos de la escena son los mostrados en la jerarquía de escenas. Los podemos observar en la siguiente imagen:

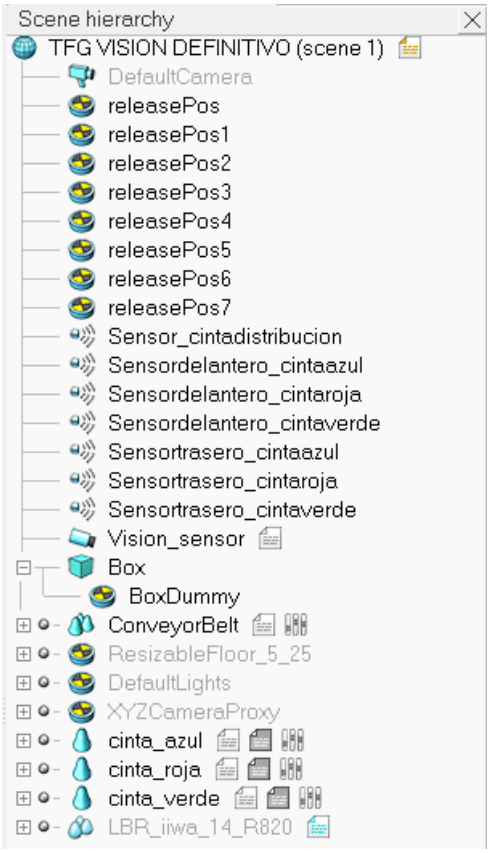

Figura 40. Jerarquía de escenas final

2.4.1. Nomenclatura de los elementos de la escena

A continuación se explicará a que elemento corresponde cada nombre:

“Box”: Objeto con la forma básica de los elementos a clasificar. Es necesario crear este elemento para poder duplicarlo mediante a la programación ya que mediante el código no se puede crear directamente un elemento, pero si copiar.

“BoxDummy”: Punto en el espacio situado en el centro del objeto “Box” el cual servirá de referencia para que el robot industrial se dirija a este punto exacto a la hora de manipular los diferentes elementos.

“releasePos”: Punto en el espacio utilizado como referencia para soltar un objeto de color verde.

“releasePos1”: Punto en el espacio utilizado como referencia para soltar un objeto de color azul.

“releasePos2”: Punto en el espacio utilizado como referencia para soltar un objeto de color azul.

“releasePos3”: Punto en el espacio utilizado como referencia para soltar un objeto de color azul.

“releasePos4”: Punto en el espacio utilizado como referencia para soltar un objeto de color rojo.

“releasePos5”: Punto en el espacio utilizado como referencia para soltar un objeto de color rojo.

“releasePos6”: Punto en el espacio utilizado como referencia para soltar un objeto de color rojo.

“releasePos7”: Punto en el espacio utilizado como referencia para soltar un objeto de color rojo.

“ConveyorBelt”: Cinta transportadora la cual será la primera fase de la simulación. Sobre esta cinta será donde se creen los elementos duplicados del objeto “Box”. También será está cinta la encargada de parar cuando el sensor situado en su parte trasera detecte un objeto válido.

“Sensor_cintadistribucion”: Es el sensor situado en la parte posterior del elemento “ConveyorBelt”. Será el encargado de desechar los elementos no apropiados y de detectar los correctos. Este sensor es la simulación conjunta del sensor de posición y el sensor de color.

“cinta_azul”: Cinta transportadora sobre al cual se colocarán las piezas de color azul. Está cinta tendrá situados en su parte delantera y trasera un sensor.

“Sensordelantero_cintaazul”: Sensor dispuesto en la parte delantera de la “cinta_azul”. Este sensor se encargará de activar la cinta transportadora cuando detecte un objeto.

“Sensortrasero_cintaazul”: Sensor situado en la parte posterior de la “cinta_azul”. Dicho sensor eliminará un objeto cuando este entre en su rango de detección. Esta acción simulará que cada pieza ha llegado a su lugar correspondiente.

“cinta_roja”: Cinta transportadora sobre al cual se colocarán las piezas de color roja. Está cinta tendrá situados en su parte delantera y trasera un sensor.

“Sensordelantero_cintaroja”: Sensor dispuesto en la parte delantera de la “cinta_roja”. Este sensor se encargará de activar la cinta transportadora cuando detecte un objeto.

“Sensortrasero_cintaroja”: Sensor situado en la parte posterior de la “cinta_roja”. Dicho sensor eliminará un objeto cuando este entre en su rango de detección. Esta acción simulará que cada pieza ha llegado a su lugar correspondiente.

“cinta_verde”: Cinta transportadora sobre al cual se colocarán las piezas de color verde. Está cinta tendrá situados en su parte delantera y trasera un sensor.

“Sensordelantero_cintaverde”: Sensor dispuesto en la parte delantera de la “cinta_verde”. Este sensor se encargará de activar la cinta transportadora cuando detecte un objeto.

"Sensortrasero_cintaverde": Sensor situado en la parte posterior de la "cinta_verde". Dicho sensor eliminará un objeto cuando este entre en su rango de detección. Esta acción simulará que cada pieza ha llegado a su lugar correspondiente.

"LBR_iiwa_14_R820": Robot industrial encargado de manipular las diferentes piezas consideradas adecuadas. Moverá las piezas desde la "ConveyorBelt" hasta "cinta_roja", "cinta_verde" o "cinta_azul" dependiendo de las características del elemento.

"ROBOTIQ_85": Pinza acoplada en la extremidad final del robot manipulador la cual simulará el agarre de las diferentes piezas.

"Vision_sensor": Sensor el cual se encargará de obtener la información de la posición y orientación de las piezas distribuidas por la cinta transportadora "ConveyorBelt".

2.4.2. Descripción de los *scripts*

Una vez conocida la nomenclatura de los diferentes elementos que conforman la escena del proyecto se procede a explicar el código realizado.

Los 6 *scripts* que conforman el proyecto están asociados a los elementos "ConveyorBelt", "Vision_sensor", "LBR_iiwa_14_R820", "cinta_azul", "cinta_verde" y "cinta_roja". Cada uno de estos se encargará de una parte del proceso.

El *script* asociado a "ConveyorBelt" es el encargado de la creación de las piezas y su transporte mediante la cinta transportadora. Este es el *script* con mayor funcionalidad del proyecto. Es el encargado de la detección y clasificación de los objetos apropiados y de desechar los elementos restantes.

El script asociado al elemento "Vision_sensor" tiene la tarea de obtener los datos de posición y orientación de las piezas que se deslizan por la cinta transportadora "ConveyorBelt".

El *script* del robot industrial "LBR_iiwa_14_R820" se ocupa del movimiento de dicho robot. Mediante trayectorias establecidas previamente sumadas a la detección de las piezas realizadas por los diferentes sensores el robot es capaz de transportarlas al lugar adecuado a través de varios parámetros predefinidos.

Por último, los 3 *scripts* restantes son muy parecidos ya que tienen la misma funcionalidad pero implementada a diferentes objetos. Cada uno de estos scripts se encargará de manejar su correspondiente cinta.

2.4.2.1. Lista de funciones de cada script:

- *Script "*ConveyorBelt":

sysCall_init()

sysCall_cleanup()

syscall_actuation()

syscall_sensing()

removeFirstObject()

insertBox()

createPath()

updatePickupPath()

- *Script* “LBR_iiwa_14_R820”:

Syscall_threadmain()

- *Script* “cinta_azul”/”cinta_roja”/”cinta_verde”:

sysCall_init()

sysCall_actuation()

sysCall_sensing()

addObject()

removeObject()

- Script “Vision_sensor”:

sysCall_init()

sysCall_vision()

2.4.2.2. Lista de funciones propias de CoppeliaSim

Las funciones propias de CoppeliaSim más importantes utilizadas en este proyecto son las siguientes:

sim.getObjectHandle(“nombre_objeto”): devuelve el *handle* correspondiente de un determinado objeto.

sim.getScriptHandle(“nombre_*script*”): devuelve el *handle* de un script. Un *script* no tiene un nombre asignado directamente; sin embargo, el *script* hereda el nombre de su objeto asociado, si tiene uno.

sim.getObjectPosition(“*handle*_objeto”): devuelve la posición de un objeto.

sim.setObjectPosition(“nombre_objeto”, datos_coordenadas): establece la posición de un objeto.

sim.getObjectOrientation(“*handle*_objeto”): devuelve la orientación de un objeto.

sim.setObjectOrientation(“nombre_objeto”, datos_coordenadas): establece la posición de un objeto.

sim.setScriptVariable("nombre_objeto",*script_handle*, variable): establece o borra una variable de un *script*.

sim.createPath(atributos_numericos): crea una trayectoria.

sim.followPath(objeto_*handle*,*path_handle,* PosicionOrientacion,relativeDistanceOnPath,velocidad, aceleración): mueve un objeto a lo largo de una trayectoria.

sim.setObjectName(objeto_*handle*,nombre_objeto): establece el nombre de un objeto según su *handle*.

sim.readProximitySensor(sensor_*handle*): lee el estado del sensor de proximidad.

sim.getSimulationTime(): devuelve el tiempo de la simulación actual.

sim.setIntegerSignal("nombre_señal",valor_señal): establece el valor de una señal.

sim.clearIntegerSignal("nombre_señal"): elimina una señal.

sim.waitForSignal("nombre_señal"): espera una señal.

sim.wait(número): espera un cierto tiempo.

sim.removeObject("nombre_objeto"): elimina un objeto.

sim.callScriptFunction("nombre_función",nombre_*script*): llama a una función de otro *script*.

sim.copyPasteObjects(objeto_*handle*): copia y pega un objeto junto a todas sus características y cálculos asociados.

sim.setShapeColor(forma_*handle*,nombre_color,*colorComponent, rgbData*): establece el color de una o varias formas.

sim.insertPathCtrlPoints(*path_handle*,opciones_numericas, *startIndex, ptCnt, ptData*): inserta uno o varios puntos de control en una trayectoria.

sim.setLinkDummy(*dummyHandle, linkedDummyHandle*): define o rompe un par de enlaces ficticios.

2.4.3. Explicación código

A continuación se explicará paso por paso el contenido del código creado. Para ello se seguirá el orden cronológico de la simulación con tal de hacer más visual y entendible este análisis. Para hacerlo más sencillo y no tan repetitivo agruparemos las 3 cintas "cinta_verde", "cinta_roja" y "cinta_azul" en una sola llamada "cinta_color" debido a que el código es idéntico en cada uno de estos tres *scripts*.

Al iniciar la simulación primeramente se establecen los parámetros iniciales de la simulación y se obtienen los *handles* de los diferentes elementos. Dichos parámetros son definidos en las funciones sysCall_init() de los *scripts* asociados a los elementos

"ConveyorBelt" [Anexo 1], "cinta_color" [Anexo 12] y "Vision_sensor" [Anexo 8] además de en el inicio del código del *script* perteneciente a "LBR_iiwa_14_R820" [Anexo 10]. Además también se crearán las diferentes trayectorias que deberá recorrer el robot industrial. Dichas trayectorias serán creadas por las 7 funciones createPath() [Anexo 6] situadas en el *script* de "ConveyorBelt".

Una vez iniciada la simulación inmediatamente se empezarán a crear piezas que irán apareciendo encima de la cinta transportadora "ConveyorBelt". La función encargada de esto es insertBox() [Anexo 5], perteneciente al *script* de la cinta nombrada anteriormente.

Dichas piezas se moverán por encima de la cinta gracias a la función sysCall_actuation() [Anexo 2]. Dicha función viene creada por defecto en el *script* de la cinta transportadora.

Las diferentes piezas avanzarán por la cinta hasta que se encuentren con el sensor de posición. Cuando una pieza entra en el rango de este sensor entra en acción la función sysCall_sensing() [Anexo 3], la cual se encarga de varias cosas.

Primero se debe recordar que este sensor simula la acción tanto de un sensor de posición como un sensor de color. Por tanto a parte de detectar un objeto también obtendrá información del color de la pieza.

En el caso en el que la pieza no sea de uno de los 3 colores definidos como correctos dicha pieza se eliminará. Si la pieza es de un color correcto el sensor de visión situado en la parte superior detectará tanto su posición como su orientación a través de la función sysCall_vison() [Anexo 9].

En un caso real sería dicha información la utilizada por el robot para saber dónde tiene que situar la pinza para coger la pieza. En este caso solo es para hacer más realista la simulación ya que el agarre de la pieza se realiza mediante la conexión del *dummy* situado en el centro de cada pieza y uno de los *dummies* pertenecientes al robot.

Una vez detectada la posición, la orientación y el color de la pieza ocurren varias cosas a la vez. Se conoce el color de la pieza y se cuantifica en la señal "signalcolor" obtenida desde la función syscall_sensing()[Anexo 3] del *script* "ConveyorBelt". Dicha señal llega a la función syscall_threadmain() [Anexo 11] del script "LBR_iiwa_14_R820". Dependiendo del color de la pieza la señal tendrá un valor u otro y mediante 3 comandos "*if*" se clasificará estas piezas en las distintas cintas transportadoras. Además se obtienen los *handles* de las diferentes piezas a través de la función removeFirstObject() [Anexo 4].

Con toda la información anterior el robot industrial es capaz de seguir las diferentes trayectorias definidas previamente. Dichas trayectorias se irán siguiendo mediante la función updatePickupPath() [Anexo 7] la cual va actualizando el valor de la trayectoria para que el robot industrial pueda seguirla.

Se han definido 3 formas diferentes de colocar las piezas sobre la cintas transportadoras. En el caso de la cinta azul las piezas están situadas paralelamente de forma vertical. En

la cinta verde tan solo se colocará una pieza de forma vertical. Mientras que en el caso de la cinta roja las piezas son colocadas paralelas 2 a 2 en vertical y horizontal formando un cuadrado. Las tres configuraciones se pueden observar en la siguiente imagen:

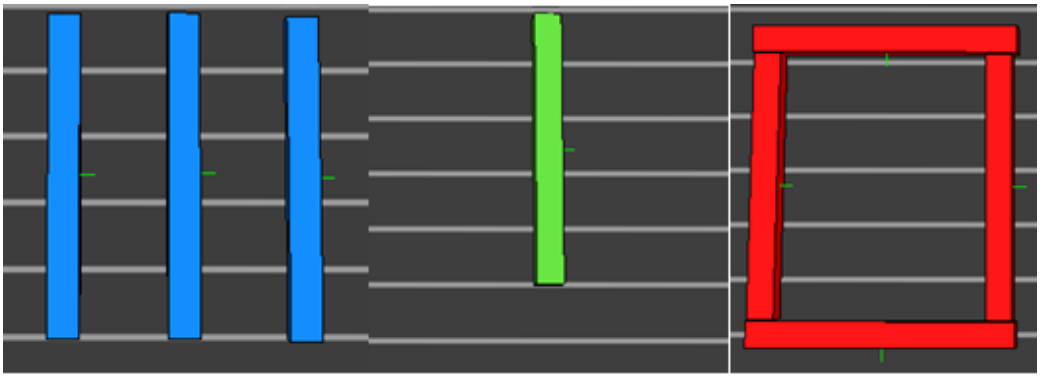

Figura 41. Configuraciones piezas

Cuando las diferentes piezas llegan a su cinta correspondiente entra en acción el *script* "cinta_color".

Mediante los sensores de posición situados en la parte inicial de la cinta se obtiene la información necesaria para el correcto funcionamiento de la función sysCall_sensig() [Anexo 14]. Cuando uno de estos sensores detecta un objeto manda una señal mediante la cual se activa la cinta transportadora correspondiente. El movimiento de cada cinta transportadora es posible gracias a la función sysCall_actuation() [Anexo 13] la cual viene por defecto creada en el *script*.

Por último, cuando la pieza avanza hasta el final de la cinta y es detectado por el sensor posterior de la cinta es eliminado. Esto es posible gracias a las funciones addObjetct() [Anexo 15] y removeObject() [Anexo 16]. La primera de estas funciones sitúa los *handles* de las correspondientes piezas en el final de las tablas de almacenamiento de datos creadas (boxList y boxDummyList). La segunda función se encarga de eliminar estas piezas de las listas anteriormente nombradas.

Con esto llegaríamos al final de la simulación. Dicha simulación puede funcionar infinitamente seguido debido a que no cuenta con ninguna interrupción que pare su funcionamiento.

3. Anexos

Vídeo simulación

https://drive.google.com/file/d/1w9mC_8PR-Bi3KCv8osI7l35nlFNH92E1/view?usp=sharing

Código perteneciente al *script* “ConveyorBelt”:

Anexo 1

```
function sysCall_init()

    -- Parametros
    beltSpeed = 0.4
    T_insert = 5
    insertCoordinate = {-2,-0.5,0.25}
    goodPercentage = 0.9
    goodColor1 = {0.345,0.859,0.192}--verde
    goodColor2 = {0,0.5,1}          --azul
    goodColor3 = {1,0,0}            --rojo

    -- Inicializar variables auxiliares
    T_last_inserted = 0
    deltaTime = 0
    hasStopped = false
    boxList = {}
    boxDummyList = {}
    boolList = {}
    color= 0

    -- Inicializar handles
    box = sim.getObjectHandle("Box")
    boxDummy = sim.getObjectHandle("BoxDummy")
    forwarder = sim.getObjectHandle("ConveyorBelt_forwarder")
    proximity = sim.getObjectHandle("Sensor_cintadistribucion")
    cinta_verde = sim.getScriptHandle("cinta_verde")
    cinta_roja = sim.getScriptHandle("cinta_roja")
    cinta_azul = sim.getScriptHandle("cinta_azul")
    sim.setScriptSimulationParameter(sim.handle_self,"conveyorBeltVelocity",beltSpeed)

    -- Insertar la primera caja durante la inicializacion
    insertBox()
```

```
    -- Obtener handles y posiciones de los dummies
    targetDummy = sim.getObjectHandle("Target")
    idlePos = sim.getObjectPosition(targetDummy,-1)
    idleOrient = sim.getObjectOrientation(targetDummy,-1)

    releasePosHandle = sim.getObjectHandle("releasePos")
    releasePos = sim.getObjectPosition(releasePosHandle,-1)
    releaseOrient = sim.getObjectOrientation(releasePosHandle,-1)

    releasePosHandle1 = sim.getObjectHandle("releasePos1")
    releasePos1 = sim.getObjectPosition(releasePosHandle1,-1)
    releaseOrient1 = sim.getObjectOrientation(releasePosHandle1,-1)

    releasePosHandle2 = sim.getObjectHandle("releasePos2")
    releasePos2 = sim.getObjectPosition(releasePosHandle2,-1)
    releaseOrient2 = sim.getObjectOrientation(releasePosHandle2,-1)

    releasePosHandle3 = sim.getObjectHandle("releasePos3")
    releasePos3 = sim.getObjectPosition(releasePosHandle3,-1)
    releaseOrient3 = sim.getObjectOrientation(releasePosHandle3,-1)

    releasePosHandle4 = sim.getObjectHandle("releasePos4")
    releasePos4 = sim.getObjectPosition(releasePosHandle4,-1)
    releaseOrient4 = sim.getObjectOrientation(releasePosHandle4,-1)

    releasePosHandle5 = sim.getObjectHandle("releasePos5")
    releasePos5 = sim.getObjectPosition(releasePosHandle5,-1)
    releaseOrient5 = sim.getObjectOrientation(releasePosHandle5,-1)

    releasePosHandle6 = sim.getObjectHandle("releasePos6")
    releasePos6 = sim.getObjectPosition(releasePosHandle6,-1)
    releaseOrient6 = sim.getObjectOrientation(releasePosHandle6,-1)

    releasePosHandle7 = sim.getObjectHandle("releasePos7")
    releasePos7 = sim.getObjectPosition(releasePosHandle7,-1)
    releaseOrient7 = sim.getObjectOrientation(releasePosHandle7,-1)
```

```
    -- Obtener el identificador de cada path handle
    releasePath = createPath("releasePath",idlePos,idleOrient,releasePos,releaseOrient)
    releasePath1 = createPath("releasePath1",idlePos,idleOrient,releasePos1,releaseOrient1)
    releasePath2 = createPath("releasePath2",idlePos,idleOrient,releasePos2,releaseOrient2)
    releasePath3 = createPath("releasePath3",idlePos,idleOrient,releasePos3,releaseOrient3)
    releasePath4 = createPath("releasePath4",idlePos,idleOrient,releasePos4,releaseOrient4)
    releasePath5 = createPath("releasePath5",idlePos,idleOrient,releasePos5,releaseOrient5)
    releasePath6 = createPath("releasePath6",idlePos,idleOrient,releasePos6,releaseOrient6)
    releasePath7 = createPath("releasePath7",idlePos,idleOrient,releasePos7,releaseOrient7)

   -- Obtener el script handle del robot
    robotScriptHandle = sim.getScriptHandle("LBR_iiwa_14_R820")
    sim.setScriptVariable("releasePath",robotScriptHandle,releasePath)
    sim.setScriptVariable("releasePath0",robotScriptHandle,releasePath0)
    sim.setScriptVariable("releasePath1",robotScriptHandle,releasePath1)
    sim.setScriptVariable("releasePath2",robotScriptHandle,releasePath2)
    sim.setScriptVariable("releasePath3",robotScriptHandle,releasePath3)
    sim.setScriptVariable("releasePath4",robotScriptHandle,releasePath4)
    sim.setScriptVariable("releasePath5",robotScriptHandle,releasePath5)
    sim.setScriptVariable("releasePath6",robotScriptHandle,releasePath6)
    sim.setScriptVariable("releasePath7",robotScriptHandle,releasePath7)



    -- Crear "Dummy Path" (sera eliminado)
    path = sim.createPath(1);
    sim.setObjectName(path,"pickupPath")
end
```

Anexo 2

```
function sysCall_actuation()
    beltVelocity=sim.getScriptSimulationParameter(sim.handle_self,"conveyorBeltVelocity")

    -- Here we "fake" the transportation pads with a single static rectangle that we dynamically reset
    -- at each simulation pass (while not forgetting to set its initial velocity vector) :
    relativeLinearVelocity={beltVelocity,0,0}

    -- Reset the dynamic rectangle from the simulation (it will be removed and added again)
    sim.resetDynamicObject(forwarder)

    -- Compute the absolute velocity vector:
    m=sim.getObjectMatrix(forwarder,-1)
    m[4]=0 -- Make sure the translation component is discarded
    m[8]=0 -- Make sure the translation component is discarded
    m[12]=0 -- Make sure the translation component is discarded
    absoluteLinearVelocity=sim.multiplyVector(m,relativeLinearVelocity)

  -- Now set the initial velocity of the dynamic rectangle:
    sim.setObjectFloatParameter(forwarder,sim.shapefloatparam_init_velocity_x,absoluteLinearVelocity[1])
    sim.setObjectFloatParameter(forwarder,sim.shapefloatparam_init_velocity_y,absoluteLinearVelocity[2])
    sim.setObjectFloatParameter(forwarder,sim.shapefloatparam_init_velocity_z,absoluteLinearVelocity[3])
end
```

Anexo 3

```
function sysCall_sensing()
     -- Lee el sensor de proximidad (0= nada detectado, 1 = objeto detectado)
     local res = sim.readProximitySensor(proximity)
    vision_sensor= sim.getObjectHandle("Vision_sensor")

    sim.handleVisionSensor(vision_sensor)

     -- Comprobar si es posible insertar una caja nueva
     if (sim.getSimulationTime()-T_last_inserted > T_insert) and not hasStopped then
         insertBox()
     end

     -- Si el sensor de proximidad detecta un objeto, para la cinta, para de insertar objetos
     if res == 1 and not hasStopped then
     --vision_sensor= sim.getObjectHandle("Vision_sensor")
     --sim.handleVisionSensor(vision_sensor)

         if boolList[1]==1   then
             color=1 -- verde
             sim.setIntegerSignal("signalcolor",color)
             sim.setScriptSimulationParameter(sim.handle_self,"conveyorBeltVelocity",0)
             deltaTime = sim.getSimulationTime()-T_last_inserted
             hasStopped = true
         -- Generar nuevo pickupPath
             updatePickupPath(boxDummyList[1])
         -- Eliminar el primer objeto y el dummie handle de la tabla
             objs = removeFirstObject()
         -- Poner pickupDummy-handle en el script del robot
             sim.setScriptVariable("pickupDummy",robotScriptHandle,objs[2])
         -- Establecer una se?al para que el robot sepa que el objeto est? disponible
             sim.setIntegerSignal("objectAvailable",1)
         end
```

```
        if boolList[1]==2 then
            color=2 -- azul
            sim.setIntegerSignal("signalcolor",color)
            sim.setScriptSimulationParameter(sim.handle_self,"conveyorBeltVelocity",0)
            deltaTime = sim.getSimulationTime()-T_last_inserted
            hasStopped = true
        -- Generar nuevo pickupPath
            updatePickupPath(boxDummyList[1])
        -- Eliminar el primer objeto y el dummie handle de la tabla
            objs = removeFirstObject()
        -- Poner pickupDummy-handle en el script del robot
            sim.setScriptVariable("pickupDummy",robotScriptHandle,objs[2])
        -- Establecer una se?al para que el robot sepa que el objeto est? disponible
            sim.setIntegerSignal("objectAvailable",1)
        end
        if boolList[1]==3 then
            color=3 -- rojo
            sim.setIntegerSignal("signalcolor",color)
            sim.setScriptSimulationParameter(sim.handle_self,"conveyorBeltVelocity",0)
            deltaTime = sim.getSimulationTime()-T_last_inserted
            hasStopped = true
        -- Generar nuevo pickupPath
            updatePickupPath(boxDummyList[1])
        -- Eliminar el primer objeto y el dummie handle de la tabla
            objs = removeFirstObject()
        -- Poner pickupDummy-handle en el script del robot
            sim.setScriptVariable("pickupDummy",robotScriptHandle,objs[2])
        -- Establecer una se?al para que el robot sepa que el objeto est? disponible
            sim.setIntegerSignal("objectAvailable",1)
        end

        if boolList[1]~=1 or boolList[1]~=2 or boolList[1]~=3 then
           local box = table.remove(boxList,1)
           local boxDummy = table.remove(boxDummyList,1)
           table.remove(boolList,1)
           sim.removeObject(box)
           sim.removeObject(boxDummy)
end
    end
```

```
    -- Si el sensor de proximidad no detecta nada y la cinta esta parada, iniciar la cinta y continuar insertando objetos
    if res == 0 and hasStopped then
        sim.clearIntegerSignal("objectAvailable")
        sim.setScriptSimulationParameter(sim.handle_self,"conveyorBeltVelocity",beltSpeed)
        hasStopped = false
        T_last_inserted = sim.getSimulationTime()-deltaTime
    end
   end
```

Anexo 4

```
function removeFirstObject()
    -- Obtener handles eliminandolos de las tablas
    local box = table.remove(boxList,1)
    local boxDummy = table.remove(boxDummyList,1)
    table.remove(boolList,1)

    --  Anadir handles
    if color==1 then
    sim.callScriptFunction("addObject",cinta_verde,{box,boxDummy})
    end

    if color==2 then
    sim.callScriptFunction("addObject",cinta_azul,{box,boxDummy})
    end

    if color==3 then
    sim.callScriptFunction("addObject",cinta_roja,{box,boxDummy})
    end
    -- Devolver handles
    return {box,boxDummy}
end
```

Anexo 5

```
function insertBox()
    -- Generar numeros aleatorios
    local rand1 = math.random()
    local rand2 = math.random()
    local rand3 = math.random()
    -- Generar perturbaciones aleatorias en la posicion y la orientacion de los objetos insetados
    local dx = (2*rand1-1)*0.05
    local dy = (2*rand2-1)*0.05
    local dphi = (2*rand3-1)*0.5
    local disturbedCoordinates = {0,0,0}
    disturbedCoordinates[1] = insertCoordinate[1]+0.5*dx
    disturbedCoordinates[2] = insertCoordinate[2]+0.5*dy
    disturbedCoordinates[3] = insertCoordinate[3]
    -- Copiar y pegar el elemento Box y BoxDummy
    local insertedObjects = sim.copyPasteObjects({box,boxDummy},0)
    -- Actualizar el tiempo de la insercion de la ultima caja
    T_last_inserted = sim.getSimulationTime()
    -- Mover y rotar
    sim.setObjectPosition(insertedObjects[1],-1,disturbedCoordinates)
    sim.setObjectOrientation(insertedObjects[1],-1,{0,0,dphi})
    -- Almacenar handles de las cajas y dummies
    table.insert(boxList,insertedObjects[1])
    table.insert(boxDummyList,insertedObjects[2])
    local decision = math.random()
    if goodPercentage-0.9 < decision and decision <= goodPercentage-0.8 then
        sim.setShapeColor(insertedObjects[1],nil,sim.colorcomponent_ambient_diffuse,goodColor1)--verde
        table.insert(boolList,1)
    end
    if goodPercentage-0.8 < decision and decision <= goodPercentage-0.45  then
        sim.setShapeColor(insertedObjects[1],nil,sim.colorcomponent_ambient_diffuse,goodColor2)--azul
        table.insert(boolList,2)
    end
    if goodPercentage-0.45 < decision and decision <= goodPercentage then
        sim.setShapeColor(insertedObjects[1],nil,sim.colorcomponent_ambient_diffuse,goodColor3)--rojo
        table.insert(boolList,3)
    end
    if decision > goodPercentage then
        sim.setShapeColor(insertedObjects[1],nil,sim.colorcomponent_ambient_diffuse,{rand1,rand2,rand3})
        table.insert(boolList,0)
    end
end
```

Anexo 6

```
function createPath(name,startPoint,startOrient,endPoint,endOrient)
    -- Crear trayectoria (path)
    local path = sim.createPath(1)

    -- Crear buffer
    local buffer = {startPoint[1],startPoint[2],startPoint[3],startOrient[1],startOrient[2],startOrient[3], 1,0,0,0,0,
                    endPoint[1],endPoint[2],endPoint[3],endOrient[1],endOrient[2],endOrient[3],             1,0,0,0,0}

    -- Insertar 2 puntos de control (comienzo y final)
    sim.insertPathCtrlPoints(path,0,0,2,buffer)

    -- Renombrar el objeto
    sim.setObjectName(path,name)

    -- Devolver el handle de la trayectoria
    return path
end
```

Anexo 7

```
function updatePickupPath(dummy)
    -- Obtener el handle del ultimo pickupPath
    local path = sim.getObjectHandle("pickupPath")
    -- Eliminar la trayectoria
    sim.removeObject(path)
    -- Obtener la posicion del dummy a alcanzar
    local dummyPos = sim.getObjectPosition(dummy,-1)
    -- Obtener la orientacion del dummy a alcanzar
    local dummyOrient = sim.getObjectOrientation(dummy,-1)
    -- Crear una nueva trayectoria
    createPath("pickupPath",idlePos,idleOrient,dummyPos,dummyOrient)

end
```

Código perteneciente al *script* "Vision_sensor":

Anexo 8

```
function sysCall_init()
    local pack={}
end
```

Anexo 9

```
function sysCall_vision(inData)
    simVision.sensorImgToWorkImg(sim.handle_self)
    simVision.edgeDetectionOnWorkImg(sim.handle_self,0.1)
    trigger, pack=simVision.blobDetectionOnWorkImg(sim.handle_self, 0.1, 0, false)
    pack=sim.unpackFloatTable(pack)
    if pack[1]==2 then
    or_posx_posy_rel={pack[10],pack[11],pack[12]}
    or_posx_posy_absoluta={or_posx_posy_rel[1]*(-57.296)-90, or_posx_posy_rel[2]*(-0.2627)-0.6693, or_posx_posy_rel[3]*(-0.2684)-0.3655}
    print(or_posx_posy_absoluta)
    end
    simVision.workImgToSensorImg(sim.handle_self)
end
```

Código perteneciente al *script* "LBR_iiwa_14_R820":

Anexo 10

```
-- Velocidad y aceleracion en la trayectoria
nominalVel = 0.25
nominalAcc = 0.5

-- Obtener los handles de los objetos y los scripts
target = sim.getObjectHandle("Target")
connector = sim.getObjectHandle("Connector")
belt1_script = sim.getScriptHandle("ConveyorBelt")

-- Inicializar variables
pickupDummy = -1
releasePath = -1
releasePath1= -1
releasePath2= -1
releasePath3= -1
releasePath4= -1
releasePath5= -1
releasePath6= -1
releasePath7= -1

lugar=0
lugar1=0
```

Anexo 11

```
function sysCall_threadmain()

        while sim.getSimulationState()~=sim.simulation_advancing_abouttostop do

    -- Pausar scripts hasta que llegue la senal de "objectAvailable"
        sim.waitForSignal("objectAvailable")
    -- Obtener alcual pickupPaht-handle
        color=sim.waitForSignal("signalcolor")
        --print(color)
        if color==1 then --verde
        path = sim.getObjectHandle("pickupPath")
    -- Seguir el pickupPath
        sim.followPath(target,path,3,0,nominalVel,nominalAcc)
    -- Esperar 1.5 segundos para imitar el proceso de conexion con la pinza
        sim.setIntegerSignal("gripperClosing",1)
        sim.wait(1.5)
    --  Conectar el "connector" al pickupDummy
        sim.setLinkDummy(connector,pickupDummy)
    -- Establecer tipo de enlace
        sim.setObjectInt32Parameter(connector,sim.dummyintparam_link_type,sim.dummy_linktype_dynamics_loop_closure)
    -- Seguir la trayectoria
        sim.followPath(target,path,3,1,-nominalVel,-nominalAcc)
    -- Seguir la ruta de soltar
        sim.followPath(target,releasePath,3,0,nominalVel,nominalAcc)
    -- Esperar 1 segundo
        sim.setIntegerSignal("gripperClosing",0)
        sim.wait(1)
    -- Desconectar el pickupDummy del "connector"
        sim.setLinkDummy(connector,-1)
    -- Seguir el camino a la posicion de inactividad
        sim.followPath(target,releasePath,3,1,-nominalVel,-nominalAcc)


        end
```

```
        if color ==2 then --azul
        lugar=lugar+1
        if lugar==1 then
        path1 = sim.getObjectHandle("pickupPath")
    -- Seguir el pickupPath
        sim.followPath(target,path1,3,0,nominalVel,nominalAcc)
    -- Esperar 1.5 segundos para imitar el proceso de conexion con la pinza
        sim.setIntegerSignal("gripperClosing",1)
        sim.wait(1.5)
    -- Conectar el "connector" al pickupDummy
        sim.setLinkDummy(connector,pickupDummy)
    -- Establecer tipo de enlace
        sim.setObjectInt32Parameter(connector,sim.dummyintparam_link_type,sim.dummy_linktype_dynamics_loop_closure)
    -- Seguir la trayectoria
        sim.followPath(target,path1,3,1,-nominalVel,-nominalAcc)
    -- Seguir la ruta de soltar
        sim.followPath(target,releasePath1,3,0,nominalVel,nominalAcc)
    -- Esperar 1 segundo
        sim.setIntegerSignal("gripperClosing",0)
        sim.wait(1)
    -- Desconectar el pickupDummy del "connector"
        sim.setLinkDummy(connector,-1)
    -- Seguir el camino a la posicion de inactividad
        sim.followPath(target,releasePath1,3,1,-nominalVel,-nominalAcc)
        end

       if lugar==2 then
       path2 = sim.getObjectHandle("pickupPath")
    -- Seguir el pickupPath
        sim.followPath(target,path2,3,0,nominalVel,nominalAcc)
    -- Esperar 1.5 segundos para imitar el proceso de conexion con la pinza
        sim.setIntegerSignal("gripperClosing",1)
        sim.wait(1.5)
    --  Conectar el "connector" al pickupDummy
        sim.setLinkDummy(connector,pickupDummy)
    -- Establecer tipo de enlace
        sim.setObjectInt32Parameter(connector,sim.dummyintparam_link_type,sim.dummy_linktype_dynamics_loop_closure)
    -- Seguir la trayectoria
        sim.followPath(target,path2,3,1,-nominalVel,-nominalAcc)
```

```
    -- Seguir la ruta de soltar
        sim.followPath(target,releasePath2,3,0,nominalVel,nominalAcc)
    -- Esperar 1 segundo
        sim.setIntegerSignal("gripperClosing",0)
        sim.wait(1)
    -- Desconectar el pickupDummy del "connector"
        sim.setLinkDummy(connector,-1)
    -- Seguir el camino a la posicion de inactividad
        sim.followPath(target,releasePath2,3,1,-nominalVel,-nominalAcc)
        end

        if lugar==3 then
        path3 = sim.getObjectHandle("pickupPath")
    -- Seguir el pickupPath
        sim.followPath(target,path3,3,0,nominalVel,nominalAcc)
    -- Esperar 1.5 segundos para imitar el proceso de conexion con la pinza
        sim.setIntegerSignal("gripperClosing",1)
        sim.wait(1.5)
    -- Conectar el "connector" al pickupDummy
        sim.setLinkDummy(connector,pickupDummy)
    -- Establecer tipo de enlace
        sim.setObjectInt32Parameter(connector,sim.dummyintparam_link_type,sim.dummy_linktype_dynamics_loop_closure)
    -- Seguir la trayectoria
        sim.followPath(target,path3,3,1,-nominalVel,-nominalAcc)
    -- Seguir la ruta de soltar
        sim.followPath(target,releasePath3,3,0,nominalVel,nominalAcc)
    -- Esperar 1 segundos
        sim.setIntegerSignal("gripperClosing",0)
        sim.wait(1)
    -- Desconectar el pickupDummy del "connector"
        sim.setLinkDummy(connector,-1)
    -- Seguir el camino a la posicion de inactividad
        sim.followPath(target,releasePath3,3,1,-nominalVel,-nominalAcc)
        lugar=lugar-3
        end

        end
```

```
        if color==3 then --rojo
        lugar1=lugar1+1
        if lugar1==1 then
        path4 = sim.getObjectHandle("pickupPath")
    -- Seguir el pickupPath
        sim.followPath(target,path4,3,0,nominalVel,nominalAcc)
    -- Esperar 1.5 segundos para imitar el proceso de conexion con la pinza
        sim.setIntegerSignal("gripperClosing",1)
        sim.wait(1.5)
    -- Conectar el "connector" al pickupDummy
        sim.setLinkDummy(connector,pickupDummy)
    -- Establecer tipo de enlace
        sim.setObjectInt32Parameter(connector,sim.dummyintparam_link_type,sim.dummy_linktype_dynamics_loop_closure)
    -- Seguir la trayectoria
        sim.followPath(target,path4,3,1,-nominalVel,-nominalAcc)
    -- Seguir la ruta de soltar
        sim.followPath(target,releasePath4,3,0,nominalVel,nominalAcc)
    -- Esperar 1 segundo
        sim.setIntegerSignal("gripperClosing",0)
        sim.wait(1)
    -- Desconectar el pickupDummy del "connector"
        sim.setLinkDummy(connector,-1)
    -- Seguir el camino a la posicion de inactividad
        sim.followPath(target,releasePath4,3,1,-nominalVel,-nominalAcc)
        end

         if lugar1==2 then
        path7 = sim.getObjectHandle("pickupPath")
    -- Seguir el pickupPath
        sim.followPath(target,path7,3,0,nominalVel,nominalAcc)
    -- Esperar 1.5 segundos para imitar el proceso de conexion con la pinza
        sim.setIntegerSignal("gripperClosing",1)
        sim.wait(1.5)
    -- Conectar el "connector" al pickupDummy
        sim.setLinkDummy(connector,pickupDummy)
    -- Establecer tipo de enlace
        sim.setObjectInt32Parameter(connector,sim.dummyintparam_link_type,sim.dummy_linktype_dynamics_loop_closure)
    -- Seguir la trayectoria
        sim.followPath(target,path7,3,1,-nominalVel,-nominalAcc)
    -- Seguir la ruta de soltar
        sim.followPath(target,releasePath7,3,0,nominalVel,nominalAcc)
```

```
    -- Esperar 1 segundo
        sim.setIntegerSignal("gripperClosing",0)
        sim.wait(1)
    -- Desconectar el pickupDummy del "connector"
        sim.setLinkDummy(connector,-1)
    -- Seguir el camino a la posicion de inactividad
        sim.followPath(target,releasePath7,3,1,-nominalVel,-nominalAcc)

        end

        if lugar1==3 then
        path6 = sim.getObjectHandle("pickupPath")
    -- Seguir el pickupPath
        sim.followPath(target,path6,3,0,nominalVel,nominalAcc)
    -- Esperar 1.5 segundos para imitar el proceso de conexion con la pinza
        sim.setIntegerSignal("gripperClosing",1)
        sim.wait(1)
    -- Conectar el "connector" al pickupDummy
        sim.setLinkDummy(connector,pickupDummy)
    -- Establecer tipo de enlace
        sim.setObjectInt32Parameter(connector,sim.dummyintparam_link_type,sim.dummy_linktype_dynamics_loop_closure)
    -- Seguir la trayectoria
        sim.followPath(target,path6,3,1,-nominalVel,-nominalAcc)
    -- Seguir la ruta de soltar
        sim.followPath(target,releasePath6,3,0,nominalVel,nominalAcc)
    -- Esperar 1 segundo
        sim.setIntegerSignal("gripperClosing",0)
        sim.wait(1)
    -- Desconectar el pickupDummy del "connector"
        sim.setLinkDummy(connector,-1)
    -- Seguir el camino a la posicion de inactividad
        sim.followPath(target,releasePath6,3,1,-nominalVel,-nominalAcc)
        end
        if lugar1==4 then
        path5 = sim.getObjectHandle("pickupPath")
    -- Seguir el pickupPath
        sim.followPath(target,path5,3,0,nominalVel,nominalAcc)
    -- Esperar 1.5 segundos para imitar el proceso de conexion con la pinza
        sim.setIntegerSignal("gripperClosing",1)
```

```
        sim.wait(1.5)
    -- Conectar el "connector" al pickupDummy
        sim.setLinkDummy(connector,pickupDummy)
    -- Establecer tipo de enlace
        sim.setObjectInt32Parameter(connector,sim.dummyintparam_link_type,sim.dummy_linktype_dynamics_loop_closure)
    -- Seguir la trayectoria
        sim.followPath(target,path5,3,1,-nominalVel,-nominalAcc)
    -- Seguir la ruta de soltar
        sim.followPath(target,releasePath5,3,0,nominalVel,nominalAcc)
    -- Esperar 1 segundo
        sim.setIntegerSignal("gripperClosing",0)
        sim.wait(1)
    -- Desconectar el pickupDummy del "connector"
        sim.setLinkDummy(connector,-1)
    -- Seguir el camino a la posicion de inactividad
        sim.followPath(target,releasePath5,3,1,-nominalVel,-nominalAcc)
        lugar1=lugar1-4
        end

        end
end
end
```

Código perteneciente al *script* “Cinta_color”:

Anexo 12

```
function sysCall_init()
    -- Parametros
    beltSpeed = 0.3

    -- Inicializar variables auxiliares
    boxList = {}
    boxDummyList = {}

    -- Obtener los handles de los objetos y scripts
    forwarder=sim.getObjectHandle('customizableConveyor_forwarder_azul')
    textureShape=sim.getObjectHandle('customizableConveyor_tableTop_azul')

    proximity1 = sim.getObjectHandle("Sensordelantero_cintaazul")
    proximity2 = sim.getObjectHandle("Sensortrasero_cintaazul")
end
```

Anexo 13

```
function sysCall_actuation()
    beltVelocity=sim.getScriptSimulationParameter(sim.handle_self,"conveyorBeltVelocity")

    -- We move the texture attached to the conveyor belt to give the impression of movement:
    t=sim.getSimulationTime()
    sim.setObjectFloatParameter(textureShape,sim.shapefloatparam_texture_x,t*beltVelocity)

    -- Here we "fake" the transportation pads with a single static rectangle that we dynamically reset
    -- at each simulation pass (while not forgetting to set its initial velocity vector) :
    relativeLinearVelocity={beltVelocity,0,0}
    -- Reset the dynamic rectangle from the simulation (it will be removed and added again)
    sim.resetDynamicObject(forwarder)
    -- Compute the absolute velocity vector:
    m=sim.getObjectMatrix(forwarder,-1)
    m[4]=0 -- Make sure the translation component is discarded
    m[8]=0 -- Make sure the translation component is discarded
    m[12]=0 -- Make sure the translation component is discarded
    absoluteLinearVelocity=sim.multiplyVector(m,relativeLinearVelocity)
    -- Now set the initial velocity of the dynamic rectangle:
    sim.setObjectFloatParameter(forwarder,sim.shapefloatparam_init_velocity_x,absoluteLinearVelocity[1])
    sim.setObjectFloatParameter(forwarder,sim.shapefloatparam_init_velocity_y,absoluteLinearVelocity[2])
    sim.setObjectFloatParameter(forwarder,sim.shapefloatparam_init_velocity_z,absoluteLinearVelocity[3])
end
```

Anexo 14

```
function sysCall_sensing()
    -- Leer los sensores de proximidad (0 = no deteccion, 1 = objeto detectado)
    local prox1 = sim.readProximitySensor(proximity1)
    local prox2 = sim.readProximitySensor(proximity2)

    -- Arranca la cinta si el objeto es detectado por prox1, para la cinta si no detecta nada
    if prox1==1 then
        sim.setScriptSimulationParameter(sim.handle_self,"conveyorBeltVelocity",beltSpeed)
    else
        sim.setScriptSimulationParameter(sim.handle_self,"conveyorBeltVelocity",0)
    end

    -- Elimina el objeto si es detectado por prox2
    if prox2 == 1 then
        removeObject()
    end
end
```

Anexo 15

```
function addObject(obj)
    -- Inserta box y boxDummy handle al final de las tablas
    table.insert(boxList,obj[1])
    table.insert(boxDummyList,obj[2])
end
```

Anexo 16

```
function removeObject()
    -- Elimina los primeros objetos de las tablas
    sim.removeObject(table.remove(boxList,1))
    sim.removeObject(table.remove(boxDummyList,1))
end
```

4. Bibliografía

https://www.coppeliarobotics.com/helpFiles/en/apiOverview.htm

https://www.coppeliarobotics.com/helpFiles/en/visionPlugin-imageProcessing.htm

https://www.coppeliarobotics.com/helpFiles/en/objectParameterIDs.htm

https://www.coppeliarobotics.com/helpFiles/en/apiConstants.htm#textureMappingModes

https://www.coppeliarobotics.com/helpFiles/en/visionSensorPropertiesDialog.htm

https://coppeliarobotics.com/helpFiles/

https://www.kuka.com/es-es/productos-servicios/sistemas-de-robot/robot-industrial/lbr-iiwa

https://robotiq.com/es/productos/pinza-adaptable-2f85-140

https://www.bannerengineering.com/mx/es/products/sensors/registration-mark-color-and-luminescence-sensors/qcm50-series.html?pageNum=1&sort=4#all

Documento 2. Pliego de Condiciones

Tabla de contenidos

1. Objeto

El objeto de este documento es fijar las condiciones técnicas mínimas que debe cumplir la simulación de un entorno industrial a través del software CoppeliaSim para la clasificación de objetos, especificando las características exactas de la simulación con tal de:

- Garantizar la seguridad de las personas que puedan intervenir en la creación de este.
- Mejorar la calidad del proyecto.
- Descripción precisa de los materiales a utilizar en la ejecución de dicho proyecto.

La correcta utilización de la simulación y el cumplimiento del presente pliego de condiciones garantizan el cumplimiento de las especificaciones fijadas por el promotor. Es responsabilidad del ejecutor final aplicar correctamente la información presentada y hacerlo siempre de acuerdo a las condiciones recogidas en el pliego de condiciones, quedando exento de responsabilidad el proyectista sobre el incumplimiento de lo escrito en este documento.

El proyecto queda totalmente definido, tanto cuantitativamente como cualitativamente, en los siguientes documentos:

- Documento 1. Memoria Técnica
- Documento 2. Pliego de Condiciones
- Documento 3. Presupuesto.

2. Condiciones Generales

Con el presente documento se intenta recopilar las exigencias legales como técnicas que deberán regir la evolución y ejecución del proyecto.

2.1. Pliego de Condiciones Facultativas

2.1.1. Obligaciones y Derechos del Proyectista

El proyectista no se responsabiliza de cualquier error en la simulación causado por la modificación del código por parte del cliente.

El promotor será el encargado de facilitar el material necesario para la realización de la simulación. Tanto una computadora con las características mínimas para poder ejecutar el software como el coste de la licencia del programa CoppeliaSim Pro.

2.1.2. Planificación y Tiempos de ejecución

Se estipula como fecha de comienzo la fecha en la que el proyectista presente los documentos necesarios para justificar la rentabilidad del diseño de la simulación y estos sean aprobados por el promotor.

El proyectista será el encargado de definir el ritmo de programación. Este deberá presentar un ciclograma en el que queden definidos los tiempos marcados y además deberá mantener informado al promotor conforme avance el proyecto.

2.1.3. Aceptación del proyecto por el cliente

Se establece un plazo de un mes durante el cual el proyectista deberá estar al servicio del cliente con tal de dar asistencia y proporcionar un mantenimiento del proyecto. Una vez pasado este plazo la solicitud de cualquier mejora o corrección de error no será considerada parte del proyecto y por tanto conllevará un cargo económico extra.

2.2. Pliego de Condiciones Económicas

El precio del proyecto incluirá tanto el diseño como cualquier material necesario para la realización de este.

El proyectista y el cliente se comprometen a cumplir las cláusulas financieras que conciernen a cada uno de ellos.

2.2.1. Fianzas y anticipos

El cliente deberá cumplir las siguientes condiciones de pago:

- Este abonará, en el momento inicial de aceptación y adjudicación del proyecto, antes de iniciarse el diseño y la programación, una cantidad a modo de fianza de un valor equivalente al 15% del coste total del proyecto.
- Si el proyectista cumple los plazos estipulados en este documento se le abonará un 55% del coste total del proyecto. Los pagos se realizarán semanalmente de una cantidad total del 55% del coste total dividido por las semanas de duración del proyecto estipuladas.
- Una vez entregado el proyecto, el cliente abonará el 30% restante.

En el caso en el que se produzca una demora en el pago, el proyectista tendrá derecho a reclamar una indemnización del 5% del importe retrasado.

En el caso en el que se produzca un retraso en la evolución del proyecto, debido a errores del proyectista, el cliente tendrá derecho a reclamar un 5% del coste total del proyecto por cada día de demora sobre la fecha acordada.

2.2.2. Composición de precios

Los precios aplicados al proyecto están sujetos a las condiciones de los acuerdos aprobados entre el cliente y el proyectista.

2.2.3. Mejoras del proyecto

Si se realiza una modificación del proyecto por parte del cliente, este deberá costear los gastos que causen dicha modificación en el precio total.

2.3. Pliego de Condiciones Legales

Marcas Registradas: El proyectista y el promotor reconoces públicamente las marcas registradas que sean utilizadas durante el desarrollo y ejecución del proyecto.

Licencias: El proyectista y el promotor reconocen tener al día las licencias del software utilizado.

Derechos de autor: Los derechos de autor de este proyecto son los estipulados por la legislación y reglamentación vigente al inicio del desarrollo del proyecto.

Documento 3. Presupuesto

Tabla de contenidos

1. Cuadro de precios elementales

Referencia	Unidades	Denominación	Precio
Licencias			
l1	ud.	CoppeliaSim Pro	750,00 €
Mano de Obra			
h1	h.	Ingeniero encargado del diseño y la programación	60,00 €

2. Cuadro de precios descompuestos

Referencia	Unidades	Denominación	Precio	Cantidad	Total
Licencias					
l1	ud.	CoppeliaSim Pro	750,00 €	1	750,00 €
Total licencias					750,00 €
Mano de Obra					
h1	h.	Ingeniero encargado del diseño y la programación	60,00 €	200	12.000,00 €
Total mano de obra					12.000,00 €
Total Presupuesto Diseño y Programación					12.750,00 €

3. Resumen

Descripción	Total
Diseño y Programación	12.750,00 €
Costes indirectos (10%)	1.275,00 €
IVA (21%)	2.945,25 €
Total Presupuesto Ejecución Proyecto	16.970,25 €