Desarrollo de módulo de comunicación en Blender para la interoperación con plataforma robótica

Trabajo Fin de Título
Grado en Ingeniería Informática (2011)

Autor:
Alberto Ramos Sánchez
Tutores:
Alexis Quesada Arencibia
Gabriel De Blasio García

Julio 2020

Índice

Índice de figuras

Índice de código

1. Introducción

Los sistemas de posicionamiento en interiores (en inglés *Indoor Positioning System — IPS*) tienen como objetivo la localización de objetos y personas en lugares cerrados, donde las tecnologías actuales de geolocalización como el GPS no tienen alcance. Para hacer posible este sistema de posicionamiento, se requiere de una infraestructura con la que generar mapas de edificios y donde localizar los objetos de su interior [1], [2].

A grandes rasgos, el funcionamiento de un IPS consiste en la recolección de información de señales generadas por emisores (que de ahora en adelante llamaremos *beacons*), estratégicamente colocados en el escenario en el que se está trabajando. Estas señales pueden ser de distinto tipo según el proyecto: WiFi, *Bluetooth Low Energy* (BLE), ultrasonido, etc. Las señales recogidas por los receptores de un dispositivo es tratada por la tecnología IPS mediante algún algoritmo de posicionamiento, extrayendo de las propiedades de las señales información que permite calcular la posición real del dispositivo. Cada conjunto de propiedades de una señal recibida asociada a una posición y orientación es conocida como señal de radiofrecuencia o huella.

El abanico de aplicaciones de esta tecnología es muy variado: desde el uso turístico, en grandes edificios como aeropuertos, metro, centros comerciales; el uso militar, para operaciones de rescate u operaciones tácticas; la automatización en sistemas de producción, construcción; localización de ingresados en hospitales y demás centros clínicos [3].

Esta es una de las líneas de investigación desarrolladas en el Instituto Universitario de Ciencias y Tecnologías Cibernética. El objetivo de este proyecto es crear un sistema de navegación aplicable al ámbito del transporte. Uno de los algoritmos aplicados en la construcción de este sistema de navegación es el Análisis de Escena basado en señales de radiofrecuencia. Su funcionamiento consta de dos fases: de calibrado, que se basa en el estudio de los escenarios para crear una base de datos que almacene un conjunto de señales de radiofrecuencia de referencia; y otra de prueba, en la que se trata de estimar la posición de un dispositivo utilizando las señales de radiofrecuencia de referencia más cercanas.

En investigaciones previas se ha demostrado que la existencia de una plataforma robótica que realice la toma de datos durante la fase de calibrado, mejora tanto la rapidez con la que se recolectan las señales, como la precisión del posicionamiento [4], [5]. Con esta intención fue construida la plataforma robótica *RoboMap* (Figura 1), encargada de recolectar señales provenientes de beacons *BLE* y ultrasónicos, y procesarlas para orientarse por el escenario. *RoboMap* es especialmente útil en escenarios con una alta complejidad, tanto por el tamaño o por la forma. Adicionalmente, permite el estudio del entorno donde se implantaría este sistema de navegación y de la influencia de ciertos parámetros de las señales en el posicionamiento.

Figura 1: Plataforma robótica *RoboMap*.

Dentro de este proyecto, es conveniente poseer una herramienta capaz de automatizar las tareas de control de la plataforma robótica. Esta herramienta debe ser capaz de comunicarse con la plataforma *RoboMap* para enviarle rutas, y recibir la posición y orientación de la plataforma en cada momento (denominada a lo largo del documento como pose). La información tanto de la plataforma como de los beacons deben ser mostrados en un escenario, creado por el usuario en un editor 3D, donde se dispongan de herramientas para diseñar el entorno.

Con la finalidad de crear esta herramienta, en este trabajo se ha desarrollado un editor, construido como un *plugin* sobre la la suite de edición 3D *Blender* [6].

Por necesidades de simplificación y modularización del código, se han creado cuatro *Addons* (término por el que se le conocen a los *plugins* dentro de *Blender*): *archibuilder*, un editor de escenarios; *robotcontrol*, donde se encuentra el módulo de comunicación con la plataforma robótica; *file manager*, que incluye las operaciones necesarias para el manejo de ficheros (para la exportación de escenarios); y *utilities*, que implementa los requisitos adicionales para el correcto funcionamiento de los tres *plugins* anteriores.

Para que el usuario pueda interactuar correctamente a través de la interfaz gráfica con la plataforma robótica, es necesario que exista una representación lo más fiel posible a la realidad del escenario donde se encuentra la plataforma. Con el objetivo de que el usuario pueda crear espacios tridimensionales, se han incluido en la aplicación varias funcionalidades relacionadas con el diseño de edificaciones (creación de habitaciones, techos, paredes), colocación de objetos y creación de *beacons*, indicando su tipo y características. Todas estas funciones son incluidas en el *Addon archibuilder*. Además, estos escenarios una vez creados pueden ser exportados al formato propio de *Blender .blend*, incluido en el *Addon filemanager*.

Dentro de las utilidades de comunicación con *Robomap*, el usuario puede crear una plataforma, que representará al robot en la interfaz. Con este conse-

guimos presentar en el escenario creado por el usuario la pose de la plataforma en el mundo real.

Entre las utilidades de comunicación, se dispone un editor de rutas, con el cual el usuario puede programar las poses que debe adoptar *RoboMap*. Las rutas son creadas dentro del escenario virtual diseñado por el usuario, donde se comprueba que puedan ser realizadas sin peligro de que ningún obstáculo lo impida. Por lo tanto, es responsabilidad del usuario crear lo más fielmente posible a la realidad las dimensiones del escenario, ayudándose de las herramientas disponibles en la aplicación. Para conseguirlo, como complemento a las herramientas aportadas en *Archibuilder*, se recomienda el uso del *Addon Measureit* para la medición de distancias. Con *Measureit* podemos medir con mayor precisión distancias y ángulos, y también sirve de apoyo para ciertas funcionalidades de nuestro editor [7].

2. Estado del arte

Las aplicaciones gráficas 3D tienen una complejidad añadida en la interacción computador-persona, ya que los periféricos comunes que se utilizan junto con los ordenadores no permiten una interacción real con elementos tridimensionales. Aún así, durante la historia del desarrollo de gráficos por computador se han ideado diferentes técnicas para permitir la representación de objetos 3D en un *hardware* que originalmente se concibió para la representación y manejo de gráficos en dos dimensiones.

La necesidad de una herramienta que permita la comunicación entre la plataforma robótica con un software con funcionalidades de control y visualización en un entorno tridimensional, nos ha llevado a realizar un estudio de los distintos entornos existentes.

2.1. Herramientas de simulación robótica

Un simulador robótico es un tipo de aplicación que, de forma independiente al dispositivo real, simula su comportamiento e interacción con otros elementos. La simulación facilita el estudio y ejecución del modelo teórico de un sistema, analizando su salida y posibilitando cambios que, de otra forma, serían muy costosos [8].

Dentro del campo de la simulación robótica, existen varias herramientas destacables.

2.1.1. Gazebo

Gazebo [9] es un simulador *open-source* de robots 3D dentro de entornos complejos. Esta aplicación comenzó a desarrollarse en 2002 como un proyecto en la Universidad del Sur de California del Dr. Andrew Howard y el estudiante Nate Koenig.

Gazebo permite el diseño de robots con una alta fidelidad, mediante la incorporación de sensores y actuadores simulados al modelo. Este simulador facilita el testeo de algoritmos de robótica e incluso es compatible con ROS.

Robot Operating System (ROS) [10] es un *framework* para el desarrollo de software para el control de robots. Aunque no se considera un sistema operativo, provee servicios típicos de un sistema operativo, como abstracción de hardware, control de dispositivos a bajo nivel, comunicación entre procesos, etc. El diseño del software está estructurado en un grafo de procesos. Cada uno de los nodos del grafo proporciona servicios a otros nodos a través del paso de mensajes entre procesos. Esto permite la modularización y adaptabilidad del software, y facilita la colaboración entre proyectos.

Actualmente, se encuentra disponible para *Ubuntu* y *Debian*. La incorporación de modelos y sensores se realiza mediante el lenguaje de marcado *xml RDFormat*, y el comportamiento de los robots es programado en C++.

2.1.2. Webots

La plataforma de escritorio *Webots* [11] provee herramientas para el modelado, programación y simulación de dispositivos robóticos. Es *open-source* y está disponible para *Windows*, *Linux* y *macOs*, y las simulaciones pueden ser exportadas a una versión web a través de *webGL*.

Los modelos de los robots y el ambiente pueden ser creados a partir del editor que provee la aplicación, mediante la utilización de figuras geométricas, materiales y texturas. En cuanto al comportamiento, puede ser programado en *C++*, *Java*, *Python*, *MATLAB* o *ROS*, a partir de una *API* distribuida junto a la aplicación.

Webots incluye *Open Dynamics Engine*, un motor de física con el que se simula la dinámica de cuerpos rígidos y la detección de colisiones.

El proyecto fue comenzado en 1996 por el Dr. Oliver Michel en la escuela politécnica federal de Lausana (EPFL).

2.1.3. CoppeliaSim

CoppeliaSim [12] se origina de un proyecto previo llamado *V-REP*. Este simulador multiplataforma y *open-source* provee un *framework* con motores gráficos y de física, herramientas para la detección de colisiones, sensores y herramientas de desarrollo de interfaces de usuario.

CoppeliaSim está basado en una arquitectura distribuida, en el cada modelo es controlado independientemente por: un *script*, desarrollado en *C/C++*, *Python*, *Java*, *Lua*, *Matlab* u *Octave*; por un nodo de *ROS* o *BlueZero*; o otras soluciones personalizadas.

2.2. Suites de modelado 3D

Las herramientas que hemos visto previamente están orientadas al diseño y testeo de modelos de sistemas. También existen otras herramientas que no están centradas en el aspecto de la robótica, pero, gracias a que aportan soluciones para la expansión de sus funcionalidades, pueden ser adaptadas a la aplicación que necesitemos.

2.2.1. 3DS Max

3ds Max [13] es una suite de modelado desarrollado por *Autodesk* para la creación de gráficos y animaciones en tres dimensiones. Tiene soporte solamente en *Windows*. Es utilizado habitualmente en televisión, cine y desarrollo de videojuegos.

Esta suite incluye herramientas de modelado y animación, edición de materiales, *shaders* y renderizado. *3ds max* permite la incorporación de *plugins*

mediante *Python* y el lenguaje de scripting *MaxScript*.

2.2.2. Maya

Maya [14] está desarrollado por la empresa *Autodesk*. Esta plataforma es un entorno de modelado orientado a la animación y es muy utilizado en la industria del cine. Su origen viene de la aplicación *Power Animation*, desarrollada por la fusión de las empresas *Alias y Wavefront*, absorbidas por *Silicon Graphics* y, a su vez, esta fue finalmente absorbida por *Autodesk*.

Maya permite la programación de *scripts* utilizando el lenguaje *Maya Embedded Language* (sintaxis similar a *Perl*), y también pueden expandirse las funcionalidades y customización de la interfaz de usuario mediante el uso de paquetes desarrollados en MEL.

2.2.3. Blender

Blender [15] surge en 1998 como un proyecto de Ton Roosendaal de renovación de la herramienta de creación 3D en el estudio de animación *NeoGeo*. Ton crea una nueva compañía en su compañía, *Not a Number (NaN)* para llevar a cabo su proyecto. La idea original era crear una suite de creación 3D multiplataforma, gratuita para el público. Se esperaba conseguir una herramienta con características de los programas profesionales pero al alcance del público general.

La herramienta despertó el interés de usuarios e inversores, por el concepto revolucionario que conllevaba para la época. Desafortunadamente, las capacidades de la empresa no fueron suficientes, y fue necesario una reducción de la compañía. Tiempo después, se lanzó la primera versión de pago, *Blender Publisher*, pero no tuvo el éxito esperado, provocando dificultades económicas en la empresa y su cierre.

En marzo de 2002, debido a la gran comunidad de usuarios que habían colaborado y quienes habían adquirido la versión de pago, Ton decidió crear la fundación *Blender Fundation*, con el objetivo de hacer *Blender* de código abierto. En julio de 2002, la campaña "Liberen a Blender" llevada a cabo por varios voluntarios (entre los que estaban antiguos empleados de NaN), logró recaudar 100000€ para comprar los derechos a los inversores de NaN. Posteriormente, el 13 de octubre, Blender fue liberada a la comunidad bajo la licencia *GNU GPLv2* [16].

La última versión de Blender a fecha actual es la 2.82. Esta suite es una combinación de varias herramientas: modelado 3D y renderizado, simulación de físicas, animación en 3D y 2D, edición de video, etc. Además, permite la programación de *scripts* en *Python* con los que automatizar tareas y customizar la interfaz, añadiendo incluso nuevas funcionalidades con la incorporación de *Addons*.

2.3. Computación gráfica en 3D

Las herramientas de edición y simulación en 3D que se utilizan en la actualidad —entre las que se incluyen las analizadas en el apartado anterior— deben su existencia a la investigación y avances tecnológicos llevados a cabo durante el pasado siglo en el campo de la computación gráfica. Es conocida la computación gráfica en 3D como el conjunto de técnicas capaces de convertir una descripción geométrica o matemática de un objeto en una imagen que simula un objeto real [17].

Tradicionalmente, los gráficos por computador se generaban a partir de descripciones muy detalladas de los objetos, a los que se les aplicaban ciertos cálculos —renderizado— para realizar una representación lo más realista posible. Más tarde, a principios de los años 80, empezaron a surgir técnicas para reducir las descripciones de los objetos (eliminación de superficies ocultas, modelos de reflección, etc).

Uno de los desarrollos más importantes en la computación de gráficos es el dispositivo rasterizador (*raster device*), un dispositivo dedicado al rasterizado de modelos tridimensionales. En este dispositivo, la imagen a mostrar es almacenada en un *buffer*, y esta información es convertida por un controlador de video para ser mostrada en un monitor. La ventaja de este dispositivo es que el proceso que genera la imagen a partir de la descripción del modelo 3D, puede ser separado del encargado del rasterizado, y del controlador de video. Además, facilita la interacción del usuario con el modelo, como podemos ver en la figura 2.

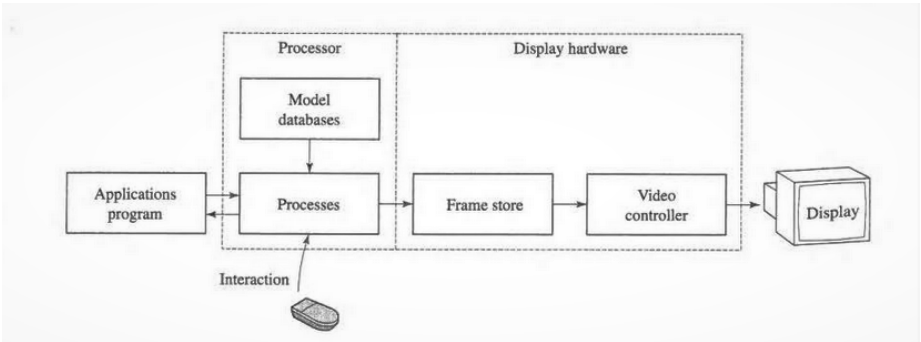


Figura 2: Raster device.
(Extraído de [17])

El *raster device* permitió que se pudieran aplicar cálculos de interacción entre objetos y fuentes de luz, facilitando la visualización de objetos con sombreado, base de los gráficos por computador en tres dimensiones. Uno de los primeros algoritmos diseñados que permitieron el sombreado fue el publicado por *Henri Gouraud* en 1971 [18]. Este algoritmo aplica el método de reflexión de *Phong* para calcular la intensidad de iluminación en cada vértice. Posteriormente, mediante interpolación, se obtiene el valor de cada pixel según las intensidades de cada vértice que forma un polígono del objeto. Con esto, se consigue una representación 2D de un modelo 3D iluminado.

Más tarde, en 1975, este algoritmo fue mejorado por *Bui Tuong Phong* [19]. El algoritmo de sombreado de *Phong* mejora los resultados en reflejos pero aumentando el coste computacional.

Actualmente, las investigaciones en el campo de la computación gráfica están dedicadas a la mejora de los algoritmos aplicados en la computación de gráficos para aprovechar al máximo las capacidades de los avances en el *hardware* [20].

2.4. Análisis de la herramienta escogida

Para este proyecto se necesita una herramienta que permita el diseño de escenarios 3D de una forma sencilla para usuarios novatos y, a la vez, tenga la suficiente capacidad para soportar una comunicación con una plataforma robótica y visualizar en un escenario virtual el estado de la plataforma.

Entre las herramientas de simulación robótica, la mejor opción sería utilizar *Gazebo*, pues ya incluye una herramienta para el diseño de escenarios 3D y, además, al ser compatible con *ROS*, la comunicación con una plataforma robótica y la propia simulación desde la aplicación es posible. Sin embargo, para crear un robot en *Gazebo* con el que realizar una simulación o comunicación se requiere un nivel alto de detalle de su funcionamiento, innecesario para los objetivos de este trabajo.

Con el propósito de conseguir una aplicación que sea lo más accesible para el usuario sin necesidad de entrar en detalles técnicos, se optó por utilizar alguna suite de edición 3D de propósito general, que fuera sencilla de utilizar y, además ofreciera opciones para ser ampliada mediante *plugins*. Entre las suites de edición más conocidas se decidió emplear *Blender*, pues permite la incorporación de *plugins* de una forma muy simple. Además, como los plugins son programados en *Python*, es posible ampliar las capacidades de la aplicación, tanto como lo permita este lenguaje y la estructura de clases de la *API* de *Blender*.

3. Objetivos

Dentro de la línea de investigación de sistemas de posicionamiento de interiores, el objetivo principal de este trabajo consiste en desarrollar un módulo de comunicación con la plataforma robótica *Robomap* dentro de la aplicación que se ha desarrollado durante las prácticas externas. Este módulo permite tanto realizar un seguimiento del posicionamiento de la plataforma en tiempo real como el envío de rutas dentro del escenario virtual modelado por el usuario.

La base sobre la que se desarrolla este TFT es el trabajo realizado durante las prácticas externas. En el transcurso de estas, en un principio se realizó un estudio previo de las opciones disponibles para cumplir la necesidad de tener una herramienta capaz de comunicarse con *Robomap*, y mostrando sus acciones dentro de un entorno simulado. Una vez se seleccionó la herramienta —*Blender*—, se adaptó mediante la incorporación de *plugins*. A la finalización de las

prácticas, las funcionalidades que se tenían desarrolladas eran el módulo de diseño de escenarios y su serialización, y un editor para crear rutas.

En cuanto a este TFT, el objetivo principal era finalizar la función de comunicación con la plataforma. Debido a la coincidencia en el tiempo del desarrollo de este trabajo y la construcción de *RoboMap*, se ha optado por la creación de un servidor que simula, a vista de la aplicación, una comunicación real con la plataforma.

Las funcionalidades que se han incorporado actualmente son: el seguimiento de las poses adoptadas por la plataforma, el envío de rutas creadas mediante un editor de planes de navegación, y una herramienta para diseñar escenarios y plataformas robóticas virtuales (con los que se representa el estado del escenario real durante la ejecución de un plan de navegación, o la simulación de un plan creado).

La comunicación entre la plataforma y la aplicación debe ser robusta, minimizando el retardo y errores. Para ello, se aplica sobre UDP un protocolo de tipos y estructuras de paquetes, comprimidos en formato *msgpack* (descrito en el apartado de desarrollo [9]), con el que se consigue una comunicación eficiente.

4. Competencias

- "*Capacidad para desarrollar y evaluar sistemas interactivos y de presentación de información compleja y su aplicación a la resolución de problemas de diseño de interacción persona computadora.*"
    - Esta competencia se ve cubierta en las funcionalidades incluidas en la aplicación relacionadas con el modelado de escenarios y visualización de la plataforma robótica. Para resolver los requisitos de la aplicación fue necesario el estudio de las posibilidades que brinda *Blender* para poder incluir las funcionalidades necesarias, además del estudio de ciertos aspectos relacionados con interfaces de usuario 3D y computación 3D.

5. Aportaciones

La tecnología de posicionamiento en interiores permite la localización de personas y objetos en lugares donde los sistemas actuales de posicionamiento, como el *GPS* no tienen alcance. Como vimos en la introducción, los sistemas IPS tienen una gran aplicabilidad en múltiples ámbitos.

Relacionando las aportaciones a los objetivos que tiene este trabajo, este contribuye en el proyecto que se esta realizando en la creación de una herramienta que permita la automatización de tareas, proveyendo una interfaz en la que realizar labores de control de la plataforma robótica con la que se trabaja en este proyecto: *RoboMap*.

Blender, al ser una herramienta de código abierto y tener una gran comunidad que lo apoya, tiene una inmensidad de recursos que hacen al editor y a

cualquier proyecto construido sobre él accesible a cualquier usuario principiante con un mínimo de tiempo de aprendizaje.

Además, *Blender* es multiplataforma, por lo que la portabilidad de nuestra aplicación es inmediata a múltiples sistemas operativos: Windows, macOS y GNU/Linux (incluyendo algunas propuestas en versiones previas para Android).

Se ha tratado que la creación de robots y de rutas sea lo más genérica posible, por lo que este *addon* puede ser utilizado para la comunicación con plataformas en otros proyectos similares con alguna modificación leve. Además, la creación de escenarios es completamente independiente, por lo que este *addon* puede separarse del módulo de comunicación y aplicarse en otros trabajos.

6. Metodología y planificación

Para la planificación del desarrollo de este trabajo se ha planteado una metodología ágil, basándose en entregas continuas, revisadas para su validación, donde se comprobó que estas cumplen los requisitos.

En cuanto al desarrollo del trabajo, se ha divido en 4 fases: estudio previo, desarrollo de las funcionalidades, evaluación del resultado y preparación de la documentación.

En una primera fase, se ha requerido un estudio previo de la herramienta *Blender*, la comunicación UDP y los módulos que posee *Python* para la comunicación. Además, se ha requerido el estudio del diseño del software para realizar la comunicación de una forma eficiente y poder adaptar *Blender* a los requisitos propuestos para este trabajo. El diseño del software se encuentra detallado en el apartado 9.

Posteriormente, tanto el desarrollo como la validación se realizó de forma conjunta, evaluando cada una de las iteraciones, tal como se plantea en una metodología ágil.

Inicialmente, en el documento *TFT01*, se propuso la siguiente planificación temporal:

- Estudio previo: 20 horas.
- Diseño y desarrollo: 180 horas
- Validación y prueba: 40 horas.
- Documentación: 60 horas.

Finalmente, la planificación realizada para cada tarea fue:

- Estudio previo: 20 horas.
- Desarrollo y validación: 240 horas.

- Preparación de la memoria: 70 horas.

Debido al aumento de plazo para la entrega de la memoria del TFT los tiempos de tanto el desarrollo como la preparación de la memoria se incrementaron, dedicando la mayor parte del tiempo extra a comprobar las funcionalidades implementadas y corregir posibles errores o mejorarlas.

También hay que tener en cuenta que todas las funcionalidades no fueron implementadas durante el desarrollo de este TFT. Las relacionadas con la creación de escenarios y de rutas fueron en su mayoría implementadas antes de la propuesta de este trabajo, durante las prácticas externas.

7. Herramientas utilizadas

En este trabajo han sido utilizadas varias herramientas software, tanto en la parte de desarrollo como en la parte de documentación.

7.1. Blender

Anteriormente, en el estado del arte (2.2.3), hemos visto los orígenes del proyecto *Blender*. Para el desarrollo de esta aplicación se ha decidido, por las posibilidades que ofrece, utilizar esta suite de edición 3D, específicamente la versión 2.82a (2.82.7).

Más que un simple editor de modelos 3D, *Blender* es un conjunto de herramientas dedicadas al diseño gráfico: modelado 3D, animación, dibujo en 2D, edición de vídeo, etc. Su facilidad de uso (mejorada notablemente en las últimas versiones), la multitud de recursos y la gran comunidad que apoya a este proyecto, lo han convertido en la mejor opción para usuarios principiantes.

El renderizado en *Blender* se consigue gracias a la incorporación de varios motores gráficos. Aunque en el pasado existieron otros motores, en la versión actual se utilizan tres: *cycles*, *freestyle* y *EEVEE*.

Cycles es el motor básico de la aplicación. Basado en el trazado de rayos de luz, este motor, calcula la interacción de la iluminación con los materiales de los objetos para generar imágenes realísticas [21], [22].

Freestyle es otro motor gráfico, no centrado en el fotorealísmo —Non-Photorealistic-Render (NPR)—. Este motor utiliza la geometría de las figuras creadas y la información almacenada en el *z-buffer*, para generar una imagen que simula el efecto de un dibujo realizado a mano alzada [23].

EEVEE es un motor que se aplica en el renderizado de imagenes a tiempo real. Construido sobre *OpenGL*, está centrado en conseguir una interacción lo más fluida, consiguiendo efectos realistas. En comparación a *Cycles*, el tiempo de cómputo es mucho menor, pero con el inconveniente de que su resultado es menos realista (aunque es imperceptible en la mayoría de situaciones). Además,

en el renderizado de figuras complejas, el límite de caras poligonales que *EEVEE* puede procesar es mucho menor [24], [25].

Adicionalmente, tenemos el motor *Workbench*, que está optimizado para realizar renderizados rápidos en la vista de trabajo del editor (3D Viewport). Ya que nuestra aplicación no genera imágenes finales y tampoco hace uso de la previsualización del renderizado, este será el motor utilizado [26].

Blender se encuentra desarrollada en *C++* y *Python*, utilizando la *API OpenGL*. Al ser código abierto, permite a la comunidad la adición de características. Gracias a la posibilidad que ofrece para desarrollar nuevos *Addons* utilizando *Python 3.7.4*, se han desarrollado multitud de herramientas para facilitar tareas al usuario y aumentar las funcionalidades de la suite.

Blender ofrece un entorno de *scripting* con el que automatizar tareas y modificar la aplicación. Para ello, se utiliza el lenguaje *Python 3.7*, que viene distribuido junto a la aplicación, incluyendo todos los módulos propios de *Blender*. En el módulo principal, *bpy*, se encuentran las estructuras necesarias para acceder a los datos de la aplicación: *meshes*, materiales, objetos, escenas, etc. Además, incluye otras clases y un sistema de registro de clases con el que crear nuevas funcionalidades y elementos en la interfaz. Adicionalmente, existen otros módulos de apoyo como: *mathutils*, que incluye funciones para el cálculo de operaciones en el espacio tridimensional; *bmesh*, que da acceso a los datos internos de representación de los *meshes* (vértices, caras, aristas, ...); *bgl*, un *wrapper* de la *API OpenGL*.

Utilizando esta facilidad, se ha desarrollado un conjunto de *plugins* que cumplan los objetivos marcados para este trabajo: implementar un editor en 3D capaz de modelar escenarios y mostrar la posición de la plataforma *Robomap*.

7.1.1. Conceptos básicos en *Blender*

Editores

Ya que *Blender* es un conjunto de herramientas de diseño, dentro de la propia aplicación se incluyen varios editores, cada uno asociado a un modo de vista. Para nuestra aplicación, los únicos que nos interesarán son: el *3D Viewport*, donde funciona nuestra aplicación; *INFO*, donde se muestran los mensajes de información al usuario; y *scripting*, donde se lleva a cabo el desarrollo en *Python*.

El tipo de editor de nuestro espacio de trabajo puede ser seleccionado en la cabecera de la interfaz (ver figura 3).

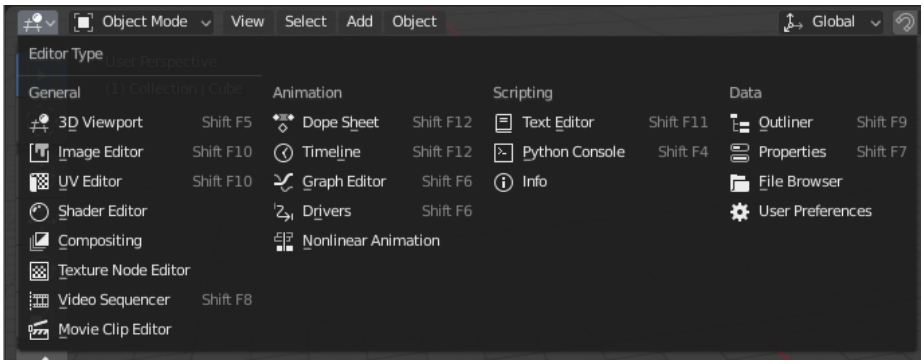


Figura 3: Editores.

Los espacios de trabajos (*workspaces*) son distribuciones de los elementos de la interfaz. Existen algunos por defecto, y podemos crear otros con la opción *Add Workspace*.

Archivo de proyecto: *.blend*

Todo proyecto en *Blender* se encuentra almacenado en un archivo de extensión *.blend*. En él se almacena una base de datos con la información relativa a escenas, objetos, colecciones, materiales, scripts, etc.

Escenas

Una escena es un elemento que nos permite organizar nuestro trabajo. Cada fichero de proyecto contiene al menos una. Dentro de cada escena, se almacenan objetos y materiales, que pueden estar contenidos en una o varias escenas a la vez.

A una escena, además de los objetos, se les asocia una unidad de medida (métrica, imperial o ninguna) y una escala. Por defecto, la escala es 1:1 en el sistema métrico.

Objetos y colecciones

Los diseños gráficos son construidos a partir del uso de objetos. Existen varios tipos según su aplicación: *meshes*, curvas, superficies, texto, luces, cámaras, etc.

Todos los objetos tienen en común una estructura que los forma, dividida en dos partes: un objeto, que mantiene la información sobre posición, rotación y escala; y los datos del objeto, que contiene datos específicos del objeto según su tipo (posición de los vértices y caras para los *meshes*, la intensidad para la iluminación, la distancia focal para las cámaras, etc).

La posición de un objeto es calculada según su origen, un vértice asociado al objeto, el cual está situado por defecto en el centro geométrico de la figura.

Los objetos se organizan en una estructura de árbol, donde uno puede ser padre de otro. Que un objeto sea hijo de otro supone que su posición, escala y rotación es dependiente de la de su padre. Esta relación es conocida en *Blender* como *parenting* [27].

En la figura 4 podemos observar como, después de aplicar una relación entre el objeto *Suzanne* y el *empty* A, el objeto cambia su tamaño y posición, volviéndose relativa a la posición del *empty*. Los cambios que se apliquen al *empty* se aplicarán al objeto hijo, pero no al contrario.

(a) Escena sin *parenting*

(b) Escena con *parenting*

Figura 4: Aplicación de *parenting*.
Extraído de `https://docs.blender.org/manual/en/latest/scene_layout/object/editing/parent.html`

Adicionalmente, los objetos dentro de una escena se encuentran almacenados en colecciones. En cada escena existe una colección especial, llamada colección de escena, que almacena todos los objetos de la escena. Dentro de esta colección pueden incluirse otras colecciones o los propios objetos. La jerarquía de objetos y colecciones en *Blender* puede ser vista como un diagrama de *Venn* (Figura 5), en el que un objeto puede estar contenido en varias colecciones.

El objetivo de las colecciones es organizar los elementos en grupos, para trabajar cómodamente.

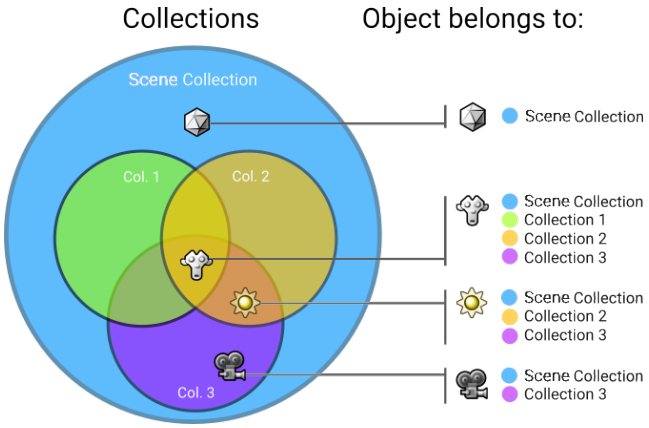


Figura 5: *Colecciones en Blender.*
Extraído de `https://docs.blender.org/manual/en/latest/scene_layout/collections/introduction.html`

En este apartado hemos visto la información necesaria para comprender el funcionamiento básico de *Blender* y de la implementación realizada en este trabajo. Para más detalles, se recomienda visitar el manual oficial [28].

7.2. Herramientas adicionales

- ***Atom***. A pesar de que la propia aplicación *Blender* dispone de un editor de *scripts*, en los casos de aplicaciones con una mayor complejidad, donde se requiere el uso de varios ficheros, es indispensable el uso de un editor externo. Para este trabajo, se optó por utilizar el editor de texto *Atom*. La ejecución se lleva a cabo utilizando uno de los ficheros del programa, tal como se explica en el tutorial de programación de scripts de *Blender* [29].

- ***GitHub***. Tanto para la entrega como para el control y almacenamiento de los cambios realizados durante el desarrollo, se ha utilizado el sistema de control de versiones *Git*, sobre la plataforma de desarrollo *Github*.

- **StarUML**. Los diagramas UML y de control de flujo se desarrollaron utilizando la versión gratuita del programa StarUML.

- ***TeXstudio***. La memoria de este trabajo ha sido redactada en *Latex*. *Latex* es un sistema de generación de texto orientado a la creación de documentos escritos. Como editor se utilizó *TeXstudio*.

8. Requisitos

La aplicación que se ha desarrollado para este trabajo tiene dos funciones diferenciadas pero dependientes entre si. La primera consiste en un editor que permite modelar escenarios del mundo real: crear edificios, colocar objetos y balizas. Por otra parte, es necesario un módulo que sea capaz de comunicarse con la plataforma robótica *RoboMap*, para poder incluir en la aplicación ciertas funcionalidades relacionadas con el control de rutas: envío de rutas y seguimiento de la plataforma robótica. Adicionalmente, es necesario una forma de crear las rutas dentro del escenario y poder evaluar si estas van a ser realizadas dentro del escenario modelado.

Los requisitos que se han descubierto para el editor de escenarios son:

- Crear paredes y habitaciones.
- Crear techos.
- Crear obstáculos indicando un área de seguridad para cada uno, el cual no puede ser invadido por el robot.
- Añadir balizas indicando el tipo de baliza y sus propiedades.

Para el módulo de comunicación, se han identificado los siguientes requisitos:

- Crear rutas validando si en estas no se interpondrá ningún obstáculo.
- Añadir robot, indicando su tipo y características. También debe incluir un área de seguridad en el cual ningún objeto debe traspasar.
- Enviar rutas a la plataforma y seguimiento a tiempo real en la aplicación.
- Realizar seguimiento a tiempo real de la posición actual de la plataforma (sin haber enviado ninguna ruta).

9. Desarrollo

Debido a la complejidad que requeriría desarrollar una aplicación desde cero utilizando herramientas de bajo nivel, se decidió adaptar alguna herramienta ya disponible a los requisitos que se habían planteado. Como se ha explicado en apartados anteriores, aunque se han investigado varias opciones, se optó por utilizar *Blender* ya que es una herramienta libre, con una gran documentación disponible para usuarios y desarrolladores y, además, es compatible para cualquier plataforma (*Linux*, *Windows* y *Mac*). *Blender* además posee una *API* en *Python*, la cual posibilita la programación de *scripts* con los que automatizar flujos de trabajo de modelado, y ampliar las funcionalidades de la aplicación mediante la creación de *Addons* (*plugins*).

9.1. Creación de *Addons*

Un *Addon* es lo que se conoce comúnmente como un *plugin*: un software que permite aumentar las capacidades o funcionalidades de una aplicación ya existente. Para este trabajo, se han creado varios *add-ons*, con los que se implementan los distintos requisitos y necesidades adicionales.

Un *addon* en *Blender* está formado por operadores, paneles y propiedades.

Operador y propiedad

Un operador es una clase que incluye el código necesario para ejecutar una acción dentro de la aplicación [30]. Dicho código se encuentra dentro del método `execute` de la clase `bpy.types.Operator`, de la cual heredan los operadores que creamos.

Opcionalmente, podemos sobrescribir los métodos: `invoke`, que se ejecuta antes que el código en `execute`; y el método `draw`, que dibuja los elementos de la ventana flotante que se muestra al lanzar el operador.

Listing 1: Estructura de clase *Operator*.

```
1   import bpy
2   from bpy.types import Operator
3
4   class MyProp(bpy.types.PropertyGroup):
5       prop_integer_value: bpy.props.IntProperty(min=0.0, ↩
            max=10.0, default=5.0)
6       prop_real_value: bpy.props.FloatProperty(min=0.0, ↩
            max=10.0, default=5.0)
7
8   class MockOperator(Operator):
9       bl_idname = "mesh.my_mock_operator"
10      bl_label = 'My mock operator'
11      bl_options = {"REGISTER", "UNDO"}
12
13      def execute(self, context):
```

```
14              """
15              # Código al ejecutar el operador
16              """
17              val = bpy.context.scene.props.prop_value
18              print("Value -> ", val)
19              self.report({'INFO'}, "Value " + str(val)) # ↩
                    Reporta al usuario mensajes de información ↩
                    o de error.
20              return {'FINISHED'} # Indica que finalizó el ↩
                    operador
21
22          def invoke(self, context, event):
23              """
24              # Se ejecuta antes del operador.
25              """
26              wm = context.window_manager
27              return wm.invoke_props_dialog(self) # Mostramos↩
                     ventana
28
29          def draw(self, context):
30              """
31              # Dibuja ventana flotante que se muestra al ↩
                    ejecutar el addon
32              """
33              props = bpy.context.scene.props
34              self.layout.label(text="Texto")
35              self.layout.prop(props, "prop_value", text="↩
                    Value")
36
37      def autoregister():
38          bpy.utils.register_class(MockOperator)
39          bpy.utils.register_class(MyProp)
40          bpy.types.Scene.props = bpy.props.PointerProperty(↩
                type=MyProp)
41
42      def autounregister():
43          del bpy.types.Scene.props
44          bpy.utils.unregister_class(MyProp)
45          bpy.utils.unregister_class(MockOperator)
46
47      if __name__=="__main__":
48          # Solamente para prueba. Esta llamada se realiza en↩
                el fichero \_\_init\_\_.py.
49          autoregister()
```

El operador 1, al ejecutarse, mostrará una ventana de diálogo, donde se pedirán dos valores numéricos. Al pulsarse el botón de *OK* se mostrarán por consola y por la ventana de notificaciones.

Un operador, además de realizar acciones sobre el escenario, puede reportar al usuario mensajes. Para ello utilizamos el método `self.report`, donde el primer parámetro es el tipo de mensaje (error, información, errores específicos) y el segundo la string del mensaje.

Para indicar un nombre y una descripción que aparecerá en la *GUI* como una etiqueta y un *tooltip*, utilizamos las variables `bl_label` y `bl_description`. El identificador `bl_idname` es el utilizado para acceder al método que lanza el operador, y este está asociado a una clase (mesh, wm, world, etc). Todos los operadores registrados se encuentran en `bpy.ops`.

```
1       bl_idname = "mesh.my_mock_operator"
2       bl_label = 'My mock operator'
3       bl_description = "It is a mock operator"
```

Las propiedades son elementos que permiten la creación de formularios, donde el usuario puede introducir datos, seleccionar elementos de listas, etc. Las propiedades pueden ser de varios tipos: numéricas, booleanas (reflejados en la interfaz como botones de *check*), vectores de valores, etc. Todos los tipos de propiedades existentes están recogidos en la documentación [31].

Aunque para el usuario sean vistas como elementos para introducir datos, realmente, las propiedades son extensiones que se realizan sobre el conjunto de datos interno de *Blender*. Estas son asignadas a distintos tipos de clases: escenas, objetos, *meshes*, etc. Esto es importante, pues cuando un usuario cambia el valor de una propiedad mediante un formulario, este mantiene su valor durante toda la ejecución, hasta que vuelva a ser cambiado.

Las propiedades pueden crearse de múltiples modos (dentro de una clase, individualmente, en grupo, etc). En la implementación de esta aplicación, la mayoría de propiedades se crean utilizando la clase `PropertyGroup`, que agrupa distintas propiedades. Para registrarse, se utiliza otra propiedad llamada `PointerProperty`, que se asigna a un tipo de objeto utilizando un nombre no existente entre las propiedades. Podemos ver como se realiza en la figura 1.

Para eliminar el registro de propiedades (`unregister`), simplemente eliminamos la variable a la que se asignó el `PointerProperty` con `del`.

El registro y eliminación de operadores (incluyendo a `PropertyGroup`) se realiza utilizando el sistema de registro de clases `bpy.utils.register\↩ _class(Class)/bpy.utils.unregister_class(Class)`.

Paneles

Con la clase `Panel` [32], [33] podemos editar la interfaz de Blender, añadiendo nuevas funcionalidades. En el apartado anterior hemos visto que podemos tomar valores de un panel flotante que aparece al ejecutar el operador. Ahora, realizaremos la misma funcionalidad pero teniendo un panel fijo en nuestra interfaz. Para ello, creamos un operador, y el panel con el que el usuario introducirá los valores y ejecutará el operador.

Listing 2: Estructura de clase *Panel*.

```
1
2       import bpy
3       from bpy.types import Operator, Panel
4
5       class MyProp(bpy.types.PropertyGroup):
6           prop_value: bpy.props.FloatProperty(min=0.0, max↩
                =10.0, default=5.0)
7           prop_name: bpy.props.StringProperty()
8           prop_bool1: bpy.props.BoolProperty()
9           prop_bool2: bpy.props.BoolProperty()
10          prop_bool3: bpy.props.BoolProperty()
11          prop_bool4: bpy.props.BoolProperty()
12
13      class MockOperator(Operator):
14          bl_idname = "mesh.my_mock_operator"
15          bl_label = 'My mock operator'
16          bl_options = {"REGISTER", "UNDO"}
17
18          def execute(self, context):
19              # C'odigo al ejecutar el operador
20
21              val = bpy.context.scene.props.prop_value
22              print("Value -> ", val)
23              self.report({'INFO'}, "Value " + str(val))
24              return {'FINISHED'} # Indica que finalizo el ↩
                    operador
25
26      class MockPanel(Panel):
27          bl_idname = "panel.MockPanel"
28          bl_label = "Mock Panel" # Nombre en la cabecera del↩
                 panel
29          bl_space_type = "VIEW_3D" # Espacio del programa ↩
                donde se usara el panel
30          bl_region_type = "UI" # Región donde se creara el ↩
                panel
31          bl_category = "Mockpanel" # Crea un panel. Si ↩
                varios paneles tienen la misma categoria se ↩
                crean subpaneles
32
33          def draw(self, context):
34              # Método que dibuja los elementos en la ↩
                    interfaz
35              props = bpy.context.scene.props
36              self.layout.label(text="Mock label")
37              self.layout.prop(props, "prop_value", text="↩
                    Value")
38              self.layout.prop(props, "prop_name", text="Name↩
                    ")
39
40              # Crea una caja dentro del panel
41              box = self.layout.box()
42
43              # Creamos dos filas dentro de la caja
```

```
44              r1 = box.row()
45              r2 = box.row()
46
47              # Anadimos dos propiedades en cada fila
48              r1.prop(props, "prop_bool1", text="Bool1")
49              r1.prop(props, "prop_bool2", text="Bool2")
50              r2.prop(props, "prop_bool3", text="Bool3")
51              r2.prop(props, "prop_bool4", text="Bool4")
52              self.layout.operator("mesh.my_mock_operator") #↩
                    Botón para ejecutar el operador
53
54      def autoregister():
55          bpy.utils.register_class(MockOperator)
56          bpy.utils.register_class(MyProp)
57          bpy.types.Scene.props = bpy.props.PointerProperty(↩
                type=MyProp)
58          bpy.utils.register_class(MockPanel)
59
60      def autounregister():
61          del bpy.types.Scene.props
62          bpy.utils.unregister_class(MyProp)
63          bpy.utils.unregister_class(MockOperator)
64          bpy.utils.unregister_class(MockPanel)
65
66      if __name__=="__main__":
67          # Solamente para prueba. Esta llamada se realiza en↩
                el fichero \_\_init\_\_.py.
68          autoregister()
```

En este panel [2] se muestran varios botones de *check*, *sliders* para introducir distintos tipos de valores y un botón para arrancar el operador.

Para crear este panel sobrescribimos la clase `bpy.types.Panel`. Esta contiene el método `draw`, que al igual que en la clase `Operator`, se encarga de añadir los elementos de la GUI.

Todos los elementos de un panel cuelgan del objeto `self.layout`. En un panel tenemos contenedores, propiedades y operadores. Se añaden utilizando los métodos de cada uno de ellos. Es decir, si queremos añadir un contenedor al panel, creamos una instancia del contenedor utilizando el método que posee `layout`. Y para anidar otro contenedor dentro del existente, utilizaríamos el método que posee el contenedor.

```
1       container = self.layout.box()
2       inner_containter = container.box()
```

La distribución de los elementos dentro de un contenedor viene determinado por su tipo. Por defecto, la distribución de los elementos si no se indica en la documentación, es en columna.

Para añadir botones y elementos de formularios, utilizamos los métodos

`prop` y `operator` que poseen tanto `self.layout` como los contenedores.

Para añadir una propiedad indicamos el grupo al que pertenece (accediendo mediante el objeto al que lo asociamos), y el nombre que le dimos dentro del `PropertyGroup`. Los operadores se añaden indicando el `bl_idname`

```
1     bl_idname = "panel.MockPanel"
2     bl_label = "Mock Panel" # Nombre en la cabecera del ↩
          panel
3     bl_space_type = "VIEW_3D" # Espacio del programa donde ↩
          se usara el panel
4     bl_region_type = "UI" # Región donde se creara el panel
5     bl_category = "Mockpanel" # Crea un panel. Si varios ↩
          paneles tienen la misma categoria, se crean ↩
          subpaneles
```

Al igual que los operadores, los paneles tienen una serie de metadatos. Los más importantes son `bl_label`, `bl_space_type`, `bl_region_type` y `bl_category`. Con `bl_space_type` y `bl_region_type` indicamos el área donde se colocará el panel. Con `bl_category` creamos una pestaña en el panel N (panel lateral derecho. Se abre pulsando la tecla N). Dentro de una pestaña se añadirán todos los paneles que se creen dentro de la misma categoría. Cada uno de ellos tendrá el nombre dado por `bl_label`.

Los paneles son registrados del mismo modo que las demás clases, utilizando la función `bpy.utils.register_class`/`bpy.utils.unregister_class`.

9.1.1. Fichero *__init__.py* y estructura general del proyecto

Hemos visto que tanto los operadores, paneles y otro tipo de clases deben ser registradas para integrarlas a la aplicación *Blender*. Al instalar un *addon*, deben llevarse a cabo todos los registros de clases. Al eliminarlo, se realiza el proceso contrario, eliminándolas.

Para ello, necesitamos una estructura que nos facilite expandir las funcionalidades del programa sin complicar el código. Con este fin, tenemos el fichero `__init__.py`, que es ejecutado por *Blender* cada vez que activamos/desactivamos un *Addon*.

```
1
2     bl_info = {
3     "name": "Example addon",
4     "author": "Alberto Ramos Sánchez",
5     "version": (1, 0, 0),
6     "blender": (2, 80, 0),
7     "location": "View3D > Sidebar Panel > Panel",
8     "description": "Addon example",
9     "category": "Animation",
10    }
11
```

```
12      import os
13      import sys
14      import bpy
15
16      # For testing
17      dir = "D:\\PE\\Desarrollo\\robotcontrol\\"
18      if not dir in sys.path:
19      sys.path.append(dir)
20
21      # TODO: Change to from . import MODULE
22      import panel
23      import operator
24
25      # Remove
26      import importlib
27
28
29      operadores = [operador]
30      paneles = [panel]
31
32      def register():
33          for op in operadores:
34              op.autoregister()
35          for p in paneles:
36              p.autoregister()
37
38      def unregister():
39          for op in operadores:
40              op.autounregister()
41          for p in paneles:
42              p.autounregister()
43
44      if __name__=='__main__':
45          register()
```

El fichero `__init__.py` contiene dos funciones —register y unregister— , que llaman a las funciones `autoregister` y `autounregister` que se encuentran en cada módulo creado. Cuando se arranca un *Addon*, se ejecuta el método `register` en `__init__.py`, que llama a todos los métodos `autoregister`, donde se encuentran realmente las llamadas a `bpy.utils.register_class`.

9.2. Creación de escenarios

El primero de los módulos incorporados en este conjunto de *add-ons* es *archibuilder*. En este se incluyen todas aquellas funcionalidades relacionadas con el diseño de escenarios: creación de edificaciones, posicionamiento de balizas, creación de objetos, etc. Una vez instalado el *add-on*, se encuentran en el panel *archibuilder*, dentro del panel N. (El panel N es el panel derecho, se abre pulsando la tecla N).

Las funcionalidades implementadas son:

- Creación de objetos.

- Creación de paredes.
- Techos.
- Creación de habitaciones.
- Creación de *beacons*.

Creación de objetos

Los obstáculos en un escenario real pueden tomar una gran complejidad que, para nuestra aplicación, no es fundamental que sea representada. El objetivo de crear obstáculos es indicar en que lugares se encuentran áreas a los que la plataforma no debe acercarse: una mesa, armarios o la entrada a algún lugar peligroso para la plataforma, como una escalera.

Mediante la utilización del objeto *Cube* representamos a los obstáculos. También se les ha incluido un área de seguridad, representado por una sombra gris, que es un distancia extra que el usuario puede marcar para que el robot no se acerque. El tamaño del margen de seguridad viene determinado proporcionalmente por el tamaño del objeto (según un porcentaje indicado por el usuario).

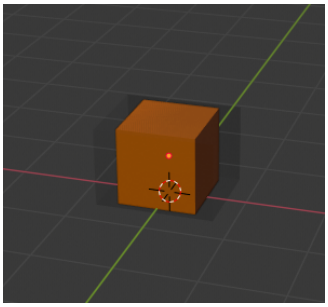


Figura 6: Obstáculo.

Para crear un nuevo obstáculo, se solicita al usuario un nombre para identificar al objeto, unas dimensiones y un margen de seguridad. El objeto se creará en el centro de la escena, por lo que será responsabilidad del usuario colocarlo donde desee.

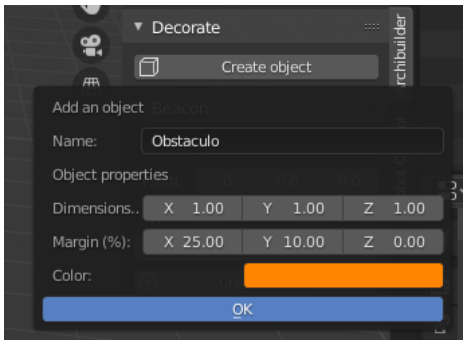


Figura 7: Menú crear obstáculo.

Implementación

Los obstáculos se crean en el centro de coordenadas, mediante la función `bpy.ops.mesh.primitive_cube_add()`. Posteriormente, se coloca a ras de suelo y, luego, se edita sus dimensiones a las solicitadas por el usuario. También le añadimos un material para cambiar el color al solicitado por el usuario (ver anexo: Función *add_obstacle*).

Además, para representar el margen de seguridad, se añade como hijo de la figura ya creada otra con color transparente, editando el material del *mesh*. El tamaño del margen de seguridad se calcula a partir del tamaño del obstáculo. Si la dimensión en un eje es x, y se ha indicado un porcentaje p por ciento, el tamaño en el eje del margen será $m = x + x * p/100$. Como se posiciona la figura de margen de seguridad en el centro, el margen de seguridad, por tanto, será un p por ciento de la dimensión del objeto en cada lado.

La creación de objetos es implementada por el operador `AddObstacleOperator` (`mesh.add_obstacle`).

Creación de paredes

Las edificaciones son representadas en la escena de *Blender* utilizando paredes que limitan su forma y dimensión.

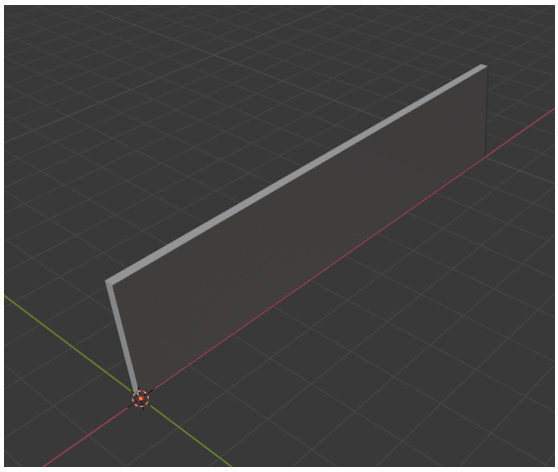


Figura 8: Muro.

La creación de las paredes se realiza solicitando al usuario un punto de inicio y final, y una anchura y altura.

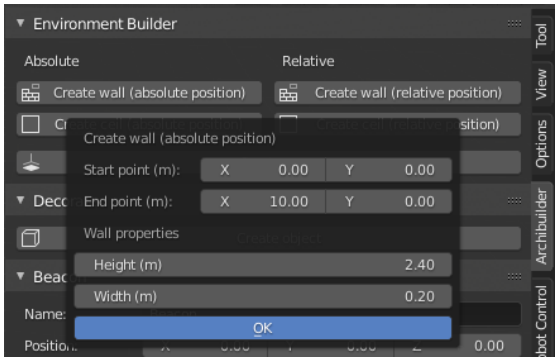


Figura 9: Menú crear muro.

Hay dos modalidades para crear los muros: de forma relativa al cursor 3D, y de forma absoluta.

Si los creamos de forma relativa, se tomará como origen de coordenadas el cursor 3D; en caso contrario, se tomará como origen el centro absoluto de la escena.

El cursor 3D es un vértice especial, que se encuentra en cualquier escena del editor de *Blender*. Es necesario para poder marcar posiciones en el espacio 3D, y tiene múltiples aplicaciones [34].

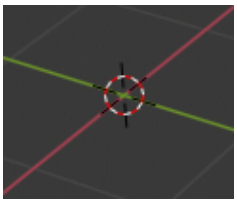


Figura 10: Cursor 3D.

Implementación

La creación de la pared se realiza en el método `create_wall` —utilizado por los operadores `CreateAbsoluteWallOperator` y `CreateRelativeWallOperator`↩ —, que recibe como parámetros el contexto de la escena, y el punto que se toma como origen de la escena (centro de coordenadas).

En este método, tomando los puntos de inicio y final, y las dimensiones, se genera un ortoedro a partir de los vértices calculados, que podemos ver en la figura 11.

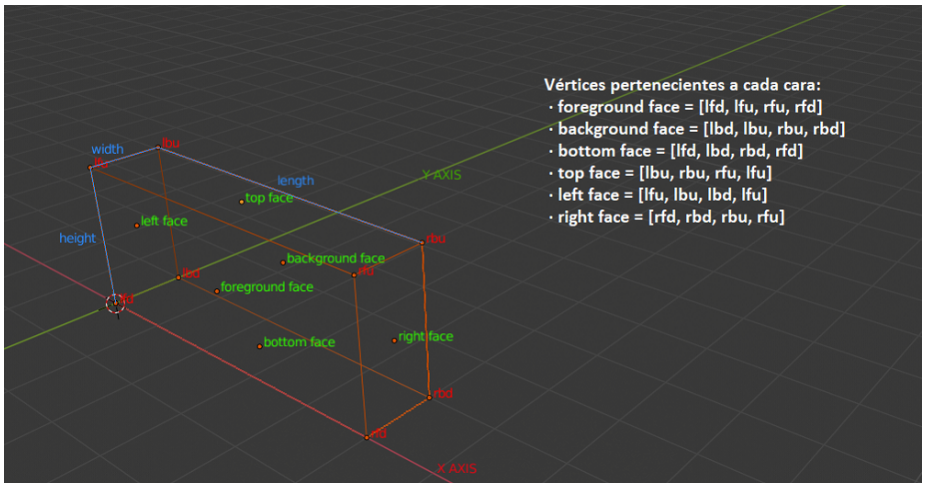


Figura 11: Vértices generados para calcular muros.

Las coordenadas de cada vértice se calcula utilizando las dimensiones de la pared, y la distancia entre el punto de inicio y fin. El punto de inicio es el vértice `lfd`, que se coloca en las coordenadas $(0, 0, 0)$. Los demás vértices se calculan a partir de las dimensiones, teniendo en cuenta que la pared se encuentra paralela al eje x.

Una vez calculado todos los vértices, se crea un nuevo *mesh* indicando vértices, aristas y caras. Además, marcamos como punto de origen de la figura al vértice `lfd` (ver anexo: Creación de paredes introduciendo datos internos del *mesh*).

Una vez creada la pared, situada paralela al eje x, se traslada al punto indicado por el usuario de inicio, y se rota sobre el eje Z θ grados. En ese momento, `lfd` estará situado en el punto de inicio y `rfd` en el punto final. El ángulo θ se calcula a partir de los puntos de inicio y final, como se indica en la figura 12. El ángulo θ es arctan($\frac{d_y - o_y}{d_x - o_x}$), teniendo en cuenta que en el caso límite, cuando $d_x - o_x = 0$, θ será 90^o o 270^o, dependiendo si $d_y - o_y$ es positivo o negativo, respectivamente.

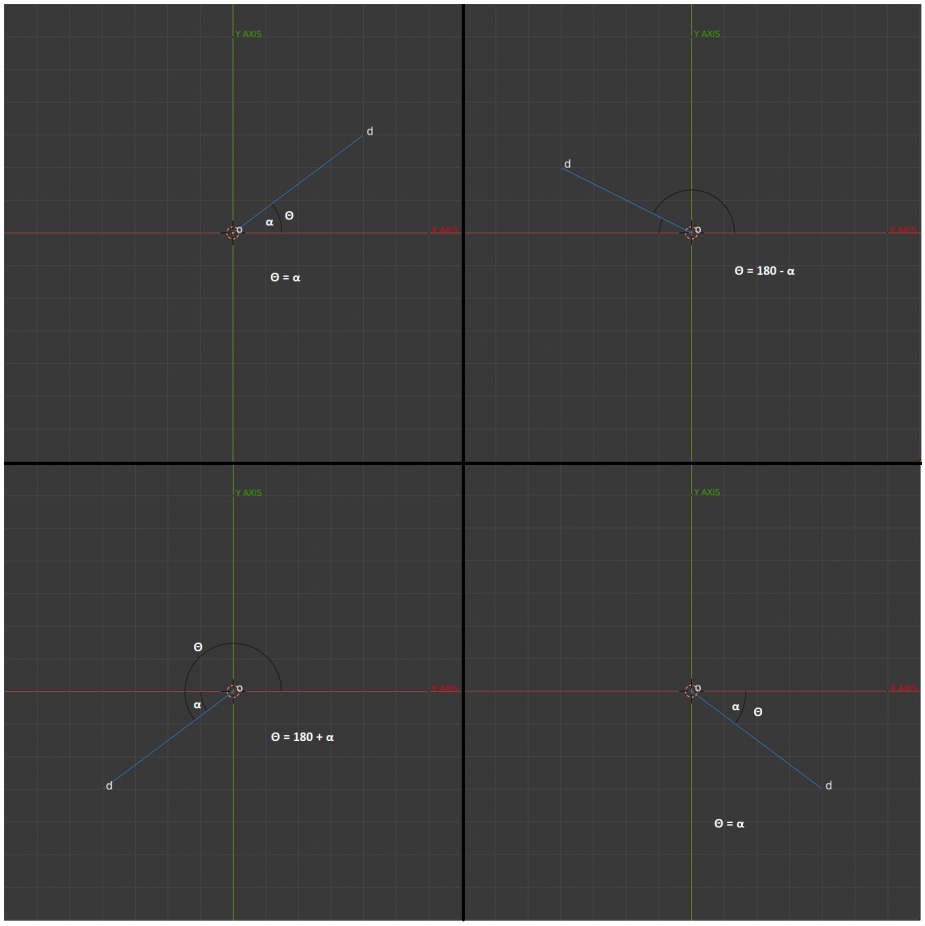


Figura 12: Ángulo θ de rotación del muro.

Para colocar la pared en una posición relativa al cursor 3D (*CreateRelativeWallOperator*), en el momento del cálculo de vértices, sumamos a todos los vértices calculados la posición del cursor, menos en el eje Z, para que quede en el suelo.

Creación de techos

Los techos son creados utilizando el *mesh* primitivo plano. Se le solicita

al usuario la posición donde colocar el plano y las dimensiones (ancho y largo). El origen del plano será la esquina inferior izquierda.

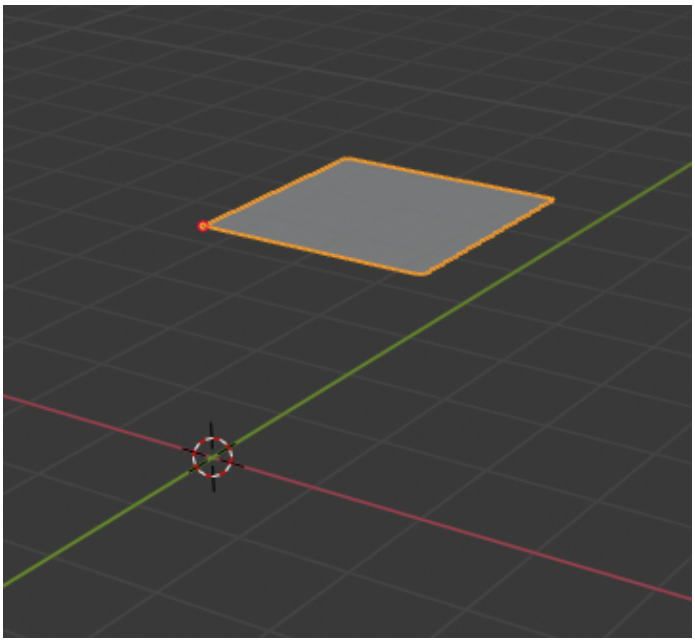


Figura 13: Techo.

Al igual que con las paredes, los techos pueden indicarse su posición relativo al centro de coordenadas global o según la posición del cursor 3D.

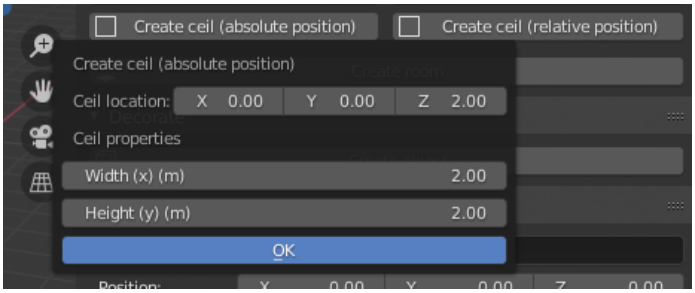


Figura 14: Menú crear techo.

Implementación

La creación de techos está implementada en los operadores `CreateAbsoluteCeilOperator` y `CreateRelativeCeilOperator`.

Para crear un techo utilizamos la siguiente función, con la que creamos un plano en la posición del cursor:

```
1       bpy.ops.mesh.primitive\_plane\_add(location=Vector((↩
            cursor.x + 1, cursor.y + 1, cursor.z + 0.0)))
```

La variable *cursor* es un vector (*mathutils.Vector*) con el que indicamos el punto que se tomará como origen —el origen de coordenadas en el operador `CreateAbsoluteCeilOperator` y el cursor en `CreateRelativeCeilOperator`↩ —.

Una vez creado el plano y desplazado una unidad en el eje x e y (para colocar el vértice inferior izquierdo en la posición del cursor), establecemos el origen del plano a la posición del cursor.

```
1           bpy.context.scene.cursor.location = cursor
2           bpy.ops.object.origin_set(type='ORIGIN_CURSOR')
```

Finalmente, se cambia el tamaño y posición al indicado por el usuario, y se le añade un material para conseguir un efecto de transparencia. (Ver anexo: Creación de techos)

Creación de habitaciones

Para facilitar la creación de edificaciones, se ha introducido el operador `CreateAbsoluteRoomOperator`, con el que creamos habitaciones rectangulares. Posteriormente, pueden ser adaptadas a otras formas combinando la creación de paredes con la creación de habitaciones.

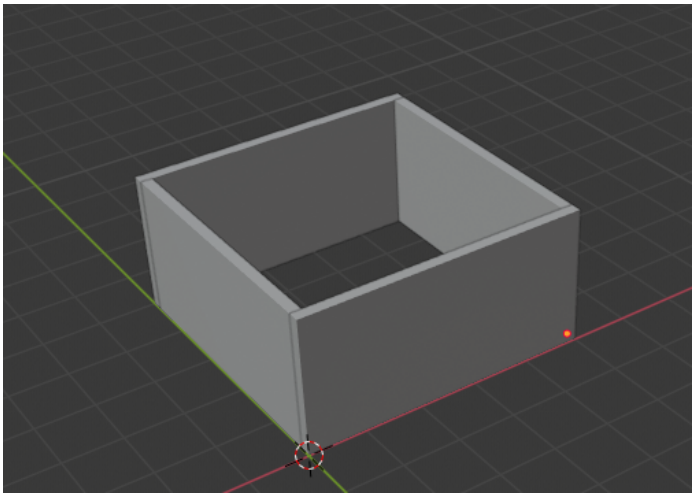


Figura 15: Habitación.

Las habitaciones se crean en un punto indicado por el usuario, tomando como origen el vértice inferior izquierdo. Se pueden indicar las dimensiones en el

eje x e y, la altura de todas las paredes, y la anchura de cada una de las cuatro paredes.

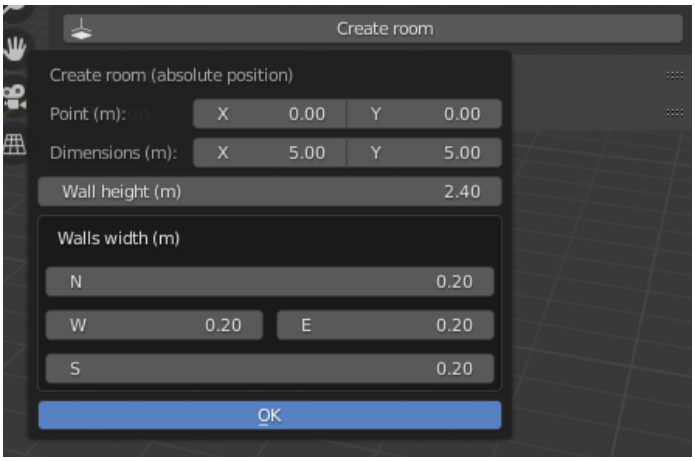


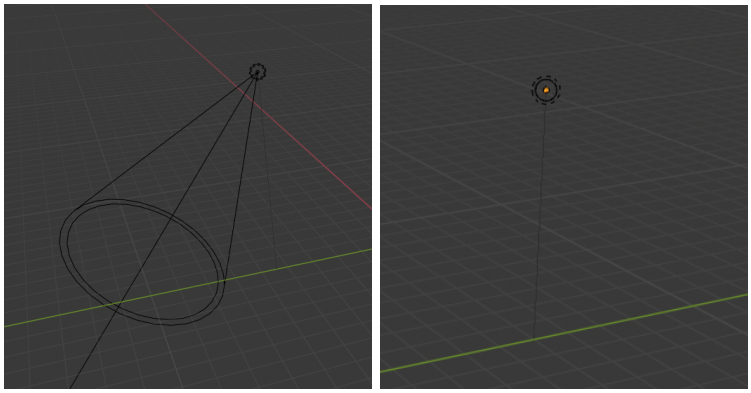
Figura 16: Menú crear habitación.

Implementación

Para esta operación nos apoyamos en el operador `CreateAbsoluteWallOperator`↩ , con el que creamos cuatro paredes, según los datos aportados por el usuario, formando un rectángulo.

Creación de beacons

La representación de los *beacons* se realiza utilizando los objetos de lámparas de iluminación de Blender. Por el momento se han identificado dos tipos de *beacons*: *BLE*, que son omnidireccionales y se representan utilizando la lámpara tipo *Point Light*; y ultrasónicos, que son unidireccionales, y se representan utilizando la lámpara tipo *Spot Light*.



(a) Ultrasónico (b) BLE

Figura 17: Beacon ultrasónico y *BLE*.

Figura 18: Menú crear *beacons*.

Implementación

Los beacons se crean utilizando las funciones `bpy.ops.object.light\_add(type='POINT')` y `bpy.ops.object.light\_add(type='SPOT')`. Según el tipo de beacon, añadimos a cada uno sus propiedades (distancia, tamaño del ángulo del foco, etc.).

Ocultación de objetos

Para mayor comodidad en la selección y visualización, se permite que los techos y los márgenes de seguridad sean ocultados.

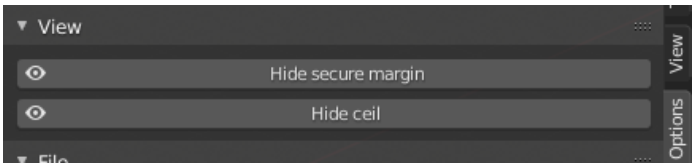


Figura 19: Ocultar techos y márgenes de seguridad.

Esto lo conseguimos mediante el método `hide_set` que poseen los objetos en Blender.

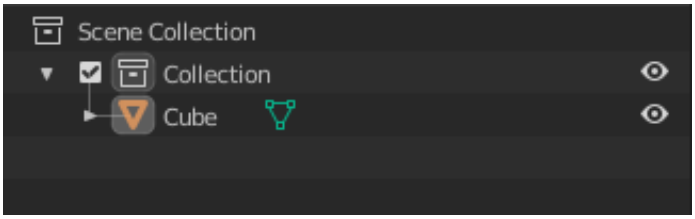


Figura 20: Ocultar objetos.

9.3. Categorización de objetos

Con el objetivo de diferenciar el tipo de los distintos objetos en la aplicación, se ha introducido una propiedad a los objetos, con el que se indica su categoría. La propiedad, creada en la figura 3, se llama `object_type` y se aplica sobre objetos (*meshes*, iluminación, ...).

Listing 3: Propiedad tipo de objeto.

```
1    bpy.types.Object.object_type = bpy.props.EnumProperty(↩
         items=items, default="OTHER")
```

Para acceder al tipo de objeto de un objeto concreto, como con otras propiedades, accedemos de la forma `bpy.data.objects["name"].object_type`.

Las categorías de objetos contempladas son:

- Muros
- Techos
- Obstáculos
- Margenes de obstáculos
- Beacon *bluetooth*
- Beacon ultrasónico
- Robot

- Margen de robot
- Cámara de robot
- Nota de robot
- Cursor geométrico
- Temporal: objetos temporales creados para realizar cálculos, como en la detección de colisiones.
- Otros: todo aquel objeto que no está contemplado en la lista. Es el valor por defecto si no se asigna un tipo a un objeto.

9.4. Serialización de escenarios

El guardado de escenarios no se puede realizar mediante la herramienta de guardado por defecto de *Blender*, pues en la escena de la aplicación, además de los elementos del escenario, se muestran otros elementos que son solamente útiles para un momento concreto de la ejecución pero que no tienen relación con el escenario diseñado (puntos de rutas creadas, robots, etc.).

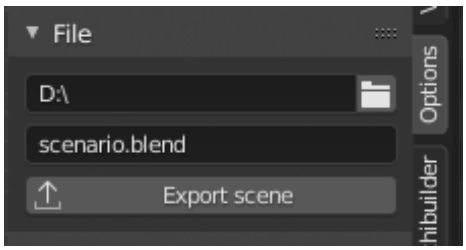


Figura 21: Menú de exportación de escenarios.

Para poder serializar únicamente los escenarios, se ha creado una herramienta que se encuentra en el panel *Options*, con el que se serializa el escenario actual en un fichero *.blend*. Esta herramienta es parte del *addon filemanager*.

La implementación se encuentra en el operador `ExportScenarioOperator`. En este se realiza una copia completa de la escena actual a otra escena mediante la operación `bpy.ops.scene.new(type='FULL_COPY')`. Posteriormente, se eliminan todos aquellos objetos que no pertenecen a una escena: no pertenecen a la categoría muro, techo, *beacons*, obstáculos o márgenes de seguridad.

Finalmente, utilizando la función `bpy.data.libraries.write`, se escribe la escena copiada y limpia en un fichero *.blend*, en la ubicación elegida por el usuario.

La selección de rutas de archivos, al igual que todas las entradas de usuario, se realiza mediante el uso de propiedades [31]. En este caso, se aplica la propiedad `StringProperty`. Al crearla, podemos indicar que sea de subtipo `DIR_PATH`, por lo que en la interfaz, nos aparecerá un selector de directorios,

con el que seleccionar el destino del fichero.

En cuanto al nombre del fichero, utilizamos el subtipo `FILE_NAME` de `StringProperty`↩ . Además, filtramos el nombre utilizando la función `set` que poseen las propiedades. La función `set`, pasada como parámetro opcional en el constructor, se ejecuta al realizarse una actualización en el valor de la propiedad, recibiendo el valor introducido por el usuario. Adicionalmente, existen las funciones: `get`↩ , que devuelve el valor al ser leída la propiedad; y `update`, que se ejecuta al modificarse el valor de la propiedad.

Para importar un escenario, podemos utilizar la herramienta de *Blender* de abrir: *File* > *Open*.

9.5. Creación de robots

Tanto *RoboMap* como otro tipo de plataformas que puedan ser utilizadas en el futuro, son representadas en la aplicación mediante el objeto `Robot`. En concreto, *RoboMap* es representada mediante un cubo con una esfera en la parte superior. Para indicar en que dirección está orientado (la rotación en el eje vertical), se ha añadido una cámara en la parte superior. La dirección a la que observa la cámara indica la orientación a la que está orientada la plataforma. Al utilizar una cámara para indicar la rotación, podemos utilizar la vista de primera persona, pulsando la tecla 0 del teclado numérico, o pulsando el botón en la interfaz a la derecha.

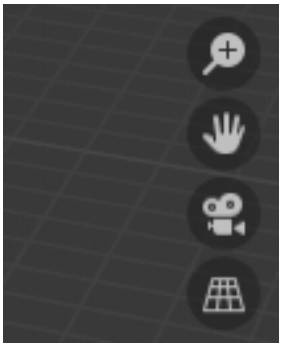


Figura 22: Vista de cámara.

Adicionalmente, igual que ocurre en los obstáculos, las plataformas tienen un margen de seguridad. Este es utilizado en la detección de colisiones para impedir que esta no se acerque a objetos y paredes. El cálculo del tamaño se realiza también del mismo modo, utilizando un porcentaje indicado por el usuario.

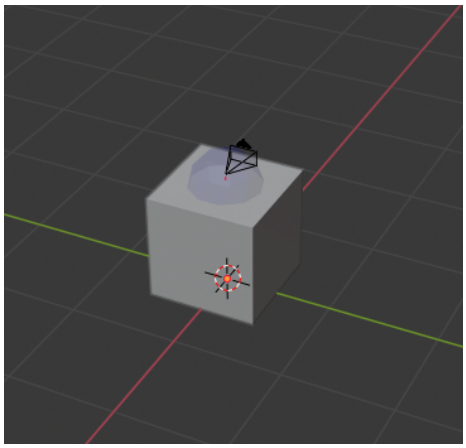


Figura 23: Representación de una plataforma robótica.

Según la implementación actual, se permite tener varios robots en un mismo escenario, añadiéndolos utilizando la operación de añadir robot (Figura 24). Cada uno está identificado por un id y un nombre único. Como veremos, es necesario para que el usuario sepa que robot está seleccionado o que robot desea eliminar.

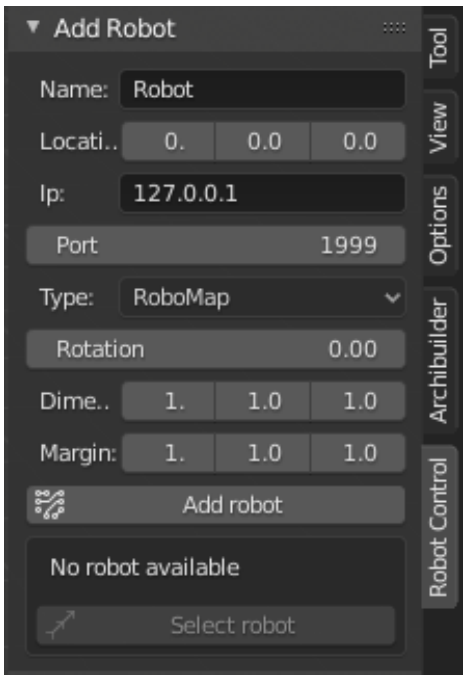


Figura 24: Menú de añadir robot.

Los robots se representan mediante la clase `Robot`, que posee las características principales (pose, ip y puerto, y el tipo de plataforma). De esta clase

heredan las clases que representan a cada tipo de robot (seleccionados en el desplegable *Type*). Cuando añadimos un robot se almacena en el contenedor `RobotSet` (Figura 25). Este contenedor solamente permite robots con id único, pues utilizamos la estructura *set* de *Python*. Sobreescribiendo el método `__hash__` de la clase `Robot`, logramos que se utilice el id como elemento diferenciador entre elementos del *set* (ver anexo: Creación de plataformas robóticas).

Para introducir los datos de un robot a añadir, se han creado dos grupos de propiedades. En `RobotProps`, se encuentran las propiedades generales de un robot (nombre, localización, ip y puerto, y tipo de robot). Como RoboMap solo acepta rotaciones en el eje vertical (eje z), se han introducido como propiedades específicas de *RoboMap* (en el grupo de propiedades `MyRobotProps`), la rotación, la dimensión de la plataforma y sus márgenes de seguridad.

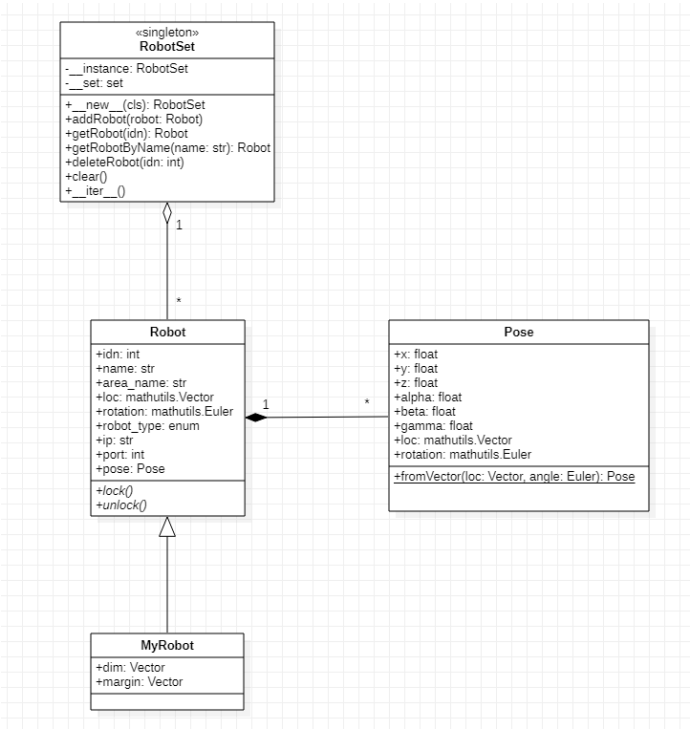


Figura 25: Clase robot.

Aunque podamos tener varias plataformas creadas en la escena, debemos seleccionar una para realizar las operaciones de comunicación que veremos en el apartado 9.7. La selección de robots se realiza con la operación *Select Robot* (Figura 26).

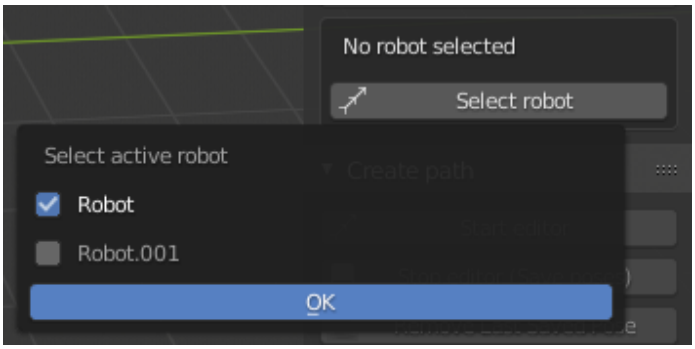


Figura 26: Seleccionar robot.

Las listas de selección en *Blender* se realizan utilizando la propiedad CollectionProperty [31].

Una propiedad colección permite agrupar varios *items* de un tipo de grupo de propiedad. En el ejemplo de la Figura 4, creamos un grupo de propiedades llamado `SceneSettingItem`. Cada instancia de esta clase se almacenará en la colleción `my_settings`.

Listing 4: Ejemplo de *CollectionProperty*.

```
1
2       import bpy
3
4
5       # Assign a collection.
6       class SceneSettingItem(bpy.types.PropertyGroup):
7           name: bpy.props.StringProperty(name="Test Property"↩
                , default="Unknown")
8           value: bpy.props.IntProperty(name="Test Property", ↩
                default=22)
9
10
11      bpy.utils.register_class(SceneSettingItem)
12
13      bpy.types.Scene.my_settings = bpy.props.↩
            CollectionProperty(type=SceneSettingItem)
14
15      my_item = bpy.context.scene.my_settings.add()
16      my_item.name = "Spam"
17      my_item.value = 1000
18
19      my_item = bpy.context.scene.my_settings.add()
20      my_item.name = "Eggs"
21      my_item.value = 30
22
23      for my_item in bpy.context.scene.my_settings:
24          print(my_item.name, my_item.value)
```

En nuestro caso, al necesitar seleccionar un único robot, creamos un grupo de propiedad con el id del robot, un booleano que nos sirva de botón de selección y el nombre del robot (para mostrarlo como etiqueta al usuario).

Listing 5: Selección de robot.

```
1    class RobotItemForSelect(bpy.types.PropertyGroup):
2        name: bpy.props.StringProperty(name="Name", default↩
             ="Unknown")
3        idn: bpy.props.IntProperty(name="Id", default=22)
4        selected: bpy.props.BoolProperty(name="Selected", ↩
             update=selectUpdate)
```

La colección, creada en el método `autoregister` como se indica en la figura 6, se le añade los items al mismo momento que se añaden *robots*, en el operador `AddRobotOperator` (Figura 7).

Listing 6: Colección de robots para seleccionar.

```
1        def autoregister():
2            ...
3            bpy.types.Scene.select_robot_collection = bpy.↩
                 props.CollectionProperty(type=↩
                 RobotItemForSelect)
4            ...
```

Listing 7: Añadir item a la collección de robots para seleccionar.

```
1        # se crea una clase MyRobot en la variable robot
2
3        myitemselect = scene.select_robot_collection.add()
4        myitemselect.name = robot.name
5        myitemselect.idn = robot.idn
```

Como queremos que la selección sea única, modificamos el método `update`↩ de la propiedad `selected` de `RobotItemForSelect`. Para conseguirlo, cuando el valor de la propiedad es `True`, cambiamos a `False` el valor `selected` en los demás `items` a `False`.

Finalmente, el id de la plataforma seleccionada es almacenada en una propiedad global, llamada `prop_robot_id`, accesible de la forma `bpy.context`↩ `.scene.selected_robot_props.prop_robot_id`. El valor de esta propiedad se completa al ejecutar el operador `SelectRobotOperator`. En el método `draw`, mostramos en una ventana emergente todos los robots contenidos en la colección creada. En la función `execute` buscamos el `item` con valor en `selected` a `True` y almacenamos en `prop_robot_id` el id del robot seleccionado. En caso de que no se encuentre ninguno, el valor será -1.

En cuanto a la eliminación de plataformas de la escena, debido a que crea-

mos, además del objeto que lo representa, clases que almacenan información adicional, es necesario una operación para eliminar plataformas de una forma limpia. Se encuentra implementada en el operador `DeleteRobotOperator`↩ , que se encarga de encontrar el objeto de la plataforma que deseamos eliminar (a través del nombre que guardamos en la clase `Robot`), y eliminarla tanto de la escena como del contenedor `RobotSet`. Los nombres de los objetos en *Blender* son únicos, por lo que no existirá ningún conflicto si se busca al robot por el nombre.

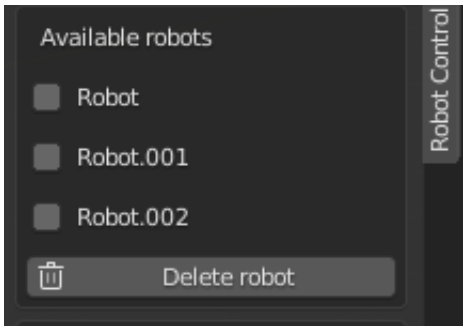


Figura 27: Eliminar robot.

El menú para seleccionar que robots eliminar (Figura 27) fue implementado del mismo modo que con el que se permite seleccionar una plataforma, pero en este caso pueden seleccionarse varios, por lo tanto no se sobrescribió el método *update*.

9.6. Editor de rutas

Unos de los principales objetivos de este trabajo es tener una herramienta con la que enviar rutas a la plataforma *RoboMap*. Estas rutas deben ser creadas dentro del escenario diseñado con el *addon archibuilder*. Con el fin de crear los planes de navegación, se ha creado un editor de rutas.

Una ruta es un conjunto de poses (posiciones y rotaciones) que la plataforma debe adoptar. Entre una pose y otra, la plataforma se mueve en línea recta, rotando en el eje vertical para alcanzar el ángulo solicitado en la pose de llegada.

Adicionalmente, en el momento de la creación de las rutas, se valida que pueda ser realizada, es decir, si algún obstáculo se interpone entre una pose y otra de la ruta.

9.6.1. Creación de rutas

La herramienta de edición de rutas implementada permite almacenar planes que, posteriormente serán enviados a la plataforma robótica. El trabajo de

indicar poses en un entorno 3D puede volverse una tarea compleja para el usuario utilizando periféricos comunes. Marcar una pose con el cursor del sistema operativo provocaría confusión en que punto realmente está eligiendo el usuario.

Una primera aproximación podría ser indicar las coordenadas y rotaciones de forma manual. Sin embargo, debemos descartar esta opción, pues la aplicación debe ser accesible y no crear un trabajo adicional al usuario.

Lo que llamamos cursor geométrico, que vemos en la Figura 28, trata de resolver esta problemática. Se trata de un cono, modificado para dar una apariencia con el que sea identificable. Trasladándolo y rotándolo como cualquier objeto, con las herramientas que posee *Blender* para ello, podemos indicar poses. El vértice del cono representa la dirección hacia donde se orienta la parte frontal de la plataforma al adoptar una pose. Su implementación se encuentra en la clase *GeometricCursor* (Figura 30). Esta clase permite realizar operaciones con el cursor, como consultar la pose actual del objeto que lo representa, dibujar al objeto que lo representa y cambiar programáticamente su pose.

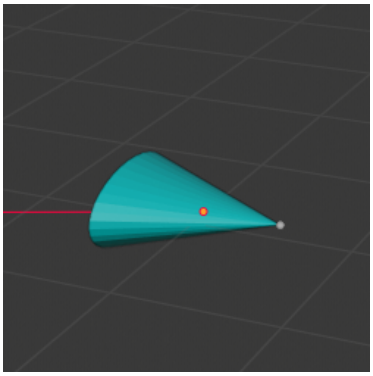
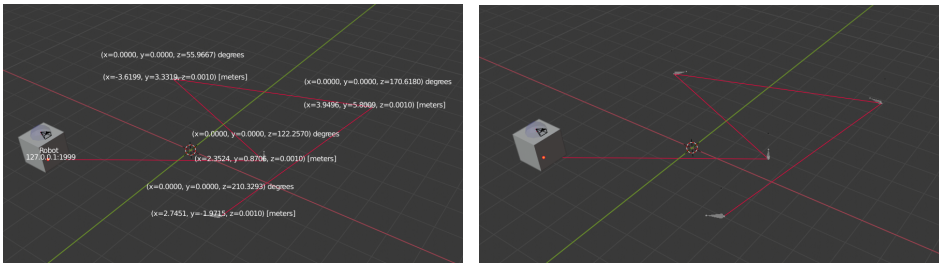


Figura 28: Cursor geométrico.

Una ruta se representa como un conjunto de poses unidas por rectas, que representan la trayectoria que debería tomar la plataforma. Para indicar la rotación en cada pose, utilizamos una flecha, que toma la misma dirección que tenía el cursor cuando marcamos la pose. Con la clase `Action` representamos la trayectoria que realiza el robot en línea recta, entre una pose de inicio y una pose final (Figura 30).



(a) Con anotaciones (b) Sin anotaciones

Figura 29: Plan de navegación.

Para realizar un seguimiento de la posición del cursor geométrico, poseemos un *listener*, que es activado al lanzar el editor de rutas.

El diseño del *listener* está basado en el patrón de diseño *observer*. Por un lado, tenemos a la clase *CursorListener*, que se encarga de detectar cuando se ha movido el cursor geométrico. Para captar cambios en la posición o rotación del cursor geométrico, hacemos uso de los *application handler*, manejadores de eventos producidos en la aplicación [35]. En concreto, utilizamos el *handler* `depsgraph_update_post`, que se ejecuta al producirse un cambio en el grafo de dependencias. A grandes rasgos, el grafo de dependencias es utilizado en *Blender* para almacenar el estado de una escena y realizar evaluaciones cuando se produce algún cambio, reduciendo el coste computacional, pues no almacena objetos [36].

Por otro lado, las clases *Observer*, registradas en *CursorListener* con el método `add_observer`, reciben notificaciones de cambios en el cursor geométrico cada vez que se ejecuta el método `fire`.

El `depsgraph_update_post` es una lista que contiene métodos que se ejecutan cada vez que se produce un evento. Para que nuestra aplicación reaccione a los cambios en el cursor geométrico, cuando arrancamos el editor de rutas con el operador `StartPosesListener` (ver anexo: Operador de inicio del listener), creamos y añadimos el método `cursor_update`, que llama a su vez, al método *fire* de `CursorListener`. Adicionalmente, se notifica a los *observers* que se arrancó el *listener* con `notifyStart`.

Listing 8: Función *cursor_update*.

```
1     def cursor_update(context):
2         """
3         Handler que se lanza cada vez que se produce un ↩
              cambio en el depsgraph
4         """
5         if CursorListener.listener is not None:
6             CursorListener.listener.fire() # Lanzamos ↩
                  listener
```

La implementación actual incluye un único *observer*, `PathDrawer`, encargado de dibujar los trazos de un plan de navegación. En este se almacena el *action* que une la última pose almacenada y la pose actual del cursor geométrico. Cada vez que se produce un cambio en la pose del cursor, `depsgraph_update_post` llamará al método `fire` del listener, que a su vez, notifica con `notifyChange` a todos los *observers*. En nuestro caso notifica a `PathDrawer`, el cual modifica la pose de destino del *action* actual por la nueva pose recibida (método *move* en `Action`). Esto provoca que cambie la representación en la escena de la recta y la flecha que representa la trayectoria.

Mientras se está editando un plan de navegación, los distintos *actions* se almacenan en una lista temporal (`TempPathContainer`). Para guardar una pose, se utiliza el operador `SavePoseOperator`. (Ver anexo: Operador guardar pose)

En el operador de guardado de poses, se toma el `Action` actual almacenado por `PathDrawer`, y se almacena en el contenedor `TempPathContainer`. Además, dibujamos una anotación con información de la nueva pose en la ruta. Posteriormente, en el apartado Utilidades adicionales se detalla en más profundidad como se realizan las anotaciones. En ese momento, se vuelve a crear un *action*, que empieza y acaba en la posición de la última pose guardada.

A través del operador `StopPosesListener`, se cierra el editor de rutas. En este se elimina el método `cursor_update` de `depsgraph_update_post`. También se notifica a los *observers* con el método `notifyStop`.

En `PathDrawer`, los métodos `notifyStart` y `notifyStop`, se utilizan para inicializar y eliminar información que se utiliza durante la creación de escenarios.

En `notifyStart` se selecciona la pose de partida: la última pose de la última ruta que se encuentra almacenada, o la posición del robot en caso de que no se hayan creado rutas. En `notifyStop`, se toman todas las poses en `TempPathContainer`, y se pasan al contenedor `PathContainer`, que recoge todos los *actions* que representan el plan de navegación cargado actualmente. Cuando se envíe a la plataforma el plan, se enviarán todas las poses almacenadas aquí.

Cabe destacar que almacenamos las instancias de la clase `Action` y no solamente las poses porque así podemos mantener en la escena de *Blender* la representación del plan de navegación.

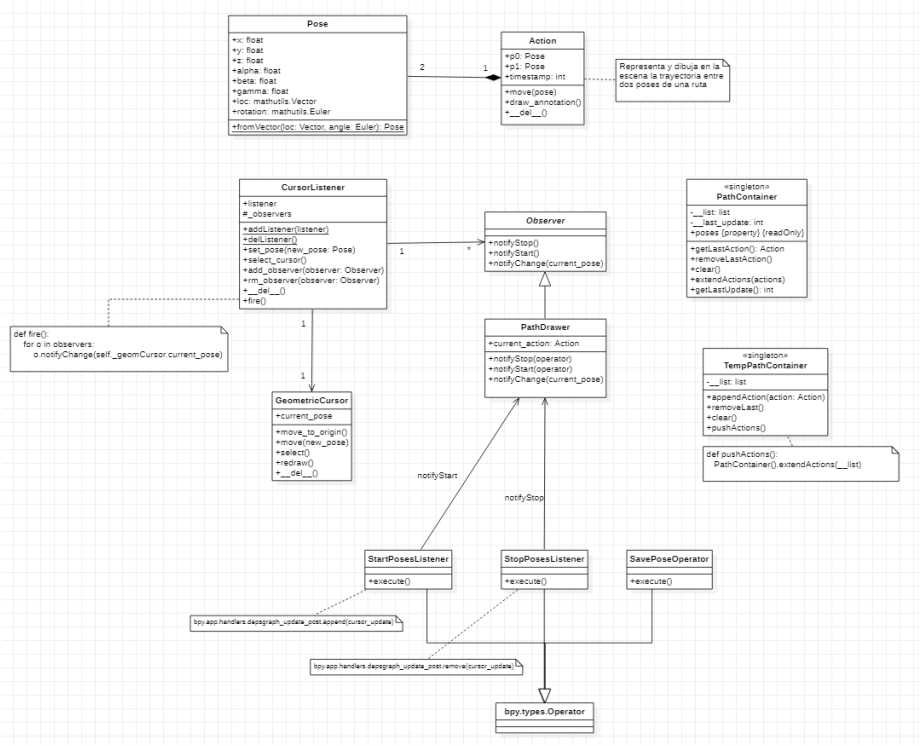
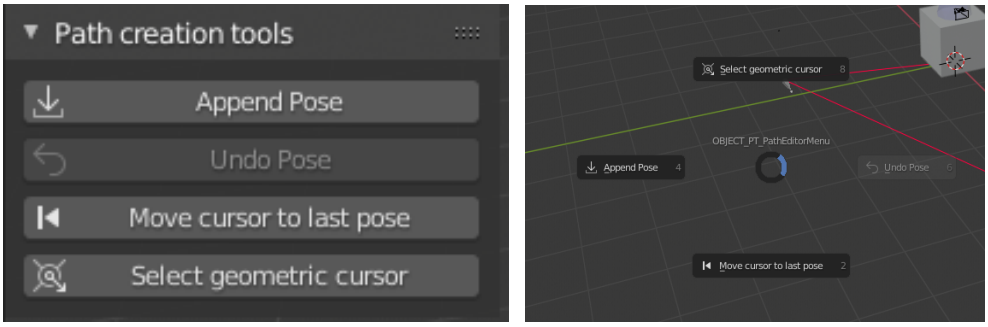


Figura 30: Esquema UML del editor de rutas.

Además de la creación de rutas, se han incluido otras funcionalidades que facilitan el diseño de planes de navegación. Estas herramientas son accesibles de dos modos (Figura 31): mediante un panel estático, situado a la izquierda de la interfaz cuando se arranca el editor de rutas; y un menú radial, que aparece con el atajo de teclado *ctrl+shift+Q*.



(a) Panel estático (b) Pie menu

Figura 31: Menú de herramientas de creación de rutas.

Append pose es el operador que ya hemos visto y que se encarga de añadir poses a un plan de navegación.

Es importante para una interfaz poseer utilidades que permitan al usuario revertir fácilmente errores. Con *undo pose*, podemos deshacer la última pose marcada. Al ejecutarlo, extraemos el último `Action` almacenado en `TempPathContainer`, pasando a ser este el `Action` actual en `PathDrawer` (pudiendo ser modificado nuevamente). El `Action` que se tenía como actual se elimina de memoria, por lo que el trazo que lo representaba desaparecerá de la escena.

Listing 9: Operación de deshacer.

```
1         # Extraemos de la lista el último Action de la lista TempPathContainer
2         last_action = pc.TempPathContainer().removeLast()
3
4         # Eliminamos flecha y linea que representa el último action (método __del__ de Action)
5         del pd.current_action
6
7         # Convertimos el Action extraído en el actual
8         pd.current_action = last_action
9
10        # Eliminamos la anotación que fue añadida al guardar el action
11        pd.current_action.del_annotation()
```

Como solo extraemos instancias de `Action` de `TempPathContainer`, la operación deshacer solamente tiene efecto en las poses que se crearon después de arrancar el editor de rutas.

Es posible que el usuario coloque el cursor en alguna pose, y desee revertir las translaciones aplicadas. Para ello, disponemos de la operación *Move cursor to last pose*, que traslada al cursor geométrico a la pose de inicio del `Action` actual (método `move` de `GeometricCursor`).

No existe ninguna restricción la cual evite que el usuario coloque el cursor geométrico dentro de algún objeto más grande, impidiendo su selección. Con la opción *select geometric cursor*, podemos seleccionar el cursor, deseleccionándose automáticamente todos los demás objetos. La selección de objetos con *bpy* se realiza con el método `select_set` que poseen los objetos en *Blender*. Para deseleccionar todos los objetos de una escena, se utiliza la función `bpy.ops.object.select_all(action='DESELECT')`.

Las herramientas que hemos descrito anteriormente, y que se encuentran en el panel *Path creation tools*, solamente se pueden utilizar mientras el editor de rutas está activo. También se han incluido otras herramientas que pueden ser utilizadas cuando no se están creando rutas nuevas, y que se encuentran en el panel *Create path* (Figura 32).

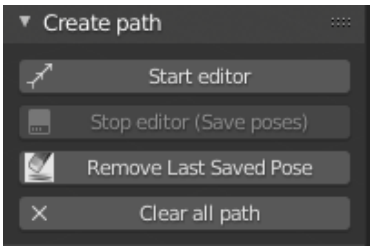


Figura 32: Panel de creación de rutas.

Start editor y *stop editor* son utilizadas para iniciar y cerrar el editor de rutas. Para eliminar la última pose guardada utilizamos *Remove last saved pose*. Con esta opción eliminamos el último *Action* almacenado en `PathContainer`. En el caso que deseemos eliminar todas las poses almacenadas, utilizaremos la operación *Clear all path*.

9.6.2. Validación de rutas

Hemos visto que utilizando el editor de rutas implementado se pueden diseñar y almacenar planes de navegación dentro de un escenario virtual que hayamos construido. Dado que la plataforma robótica no conoce el estado del escenario, es necesario que desde la aplicación se realice una verificación de que en la ruta no existan obstáculos que se interpongan en ninguna trayectoria.

Al añadir una nueva pose a una ruta, se comprueba que ningún objeto esté dentro del área que va a ocupar el robot durante la trayectoria. En la figura 33 podemos ver un área sombreada entre el punto de inicio y fin: el chequeo de colisiones consiste en detectar que no exista ningún objeto dentro.

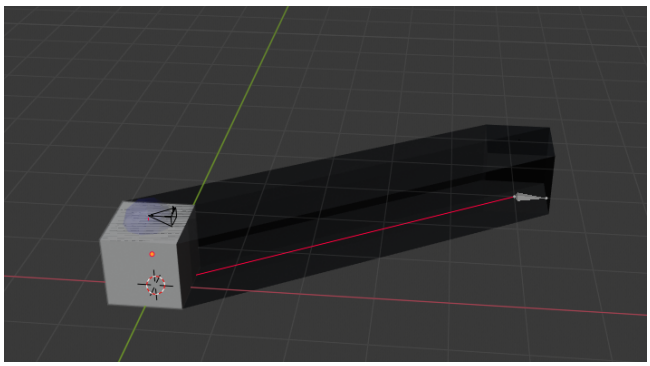


Figura 33: Área de colisión.

Detección de solapamientos entre objetos

La validación de rutas se basa en la detección de solapamientos entre objetos. Consideramos que dos objetos se están solapando cuando alguna arista

de un objeto intersecta al otro objeto, o cuando un objeto se encuentra dentro de otro.

La comprobación de solapamientos se realiza mediante la utilización de la función *overlap* de la clase *BVHTree*.

BVHTree, del módulo `mathutils.bvhtree`, representa una estructura de datos *Bounding Volume Hierarchy (BVH)* [37]. Los árboles BVH son utilizados en cálculos de proximidad, trazado de rayos, solapamiento, etc. A partir de *bounding box* de formas primitivas, son capaces de dividir un objeto en formas simples y crear una estructura de objetos [38]. En la figura 34 podemos observar como, utilizando un *bounding box* en forma de ortoedro, se divide al objeto del avión en conjuntos más simples. El nodo raíz e intermedios representan las operaciones para alcanzar las divisiones del objeto, almacenadas en los nodos hoja.

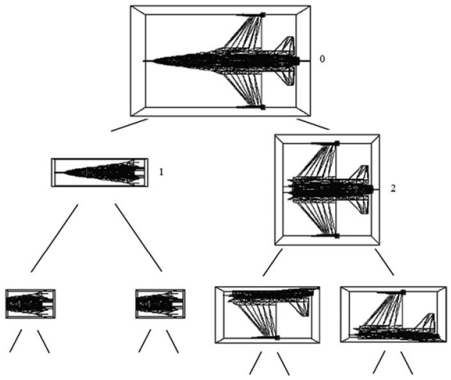


Figura 34: Bounding Volume Hierarchy Tree.
Extraído de `https://www.ibr.cs.tu-bs.de/projects/bvh/`

La construcción de un *BVHTree* de un objeto en *Blender* requiere la conversión del objeto en un *bmesh*. Un *bmesh* [39] almacena y permite acceder a los datos internos de los objetos en *Blender* (vértices, caras, arístas, etc).

Crear un *bmesh* desde un objeto requiere realizar ciertas transformaciones para que este almacene su información en relación a las coordenadas de la escena. Para ello, realizamos una copia del objeto que queremos trasformar a *bmesh* para evitar dañar el objeto original. Posteriormente, aplicamos las transformaciones con `transform_apply` y añadimos los datos del objeto al *bmesh* creado (`bmesh.new()`). Además, aplicamos la matriz de transformación de la escena al *bmesh*, para que se aplique cualquier transformación que se haya realizado. Finalmente, para hacer accesibles los datos del *bmesh*, debemos ejecutar el método `ensure_lookup_table`.

Los *bmesh* son eliminados de memoria automáticamente cuando termina el programa. Sin embargo, como es difícil predecir cuando esto va a ocurrir, es recomendable eliminarlas manualmente con el método *free*.

Listing 10: Función crear *Bmesh*.

```
1         def create_bmesh(obj):
2             tmp = obj.copy()
3             tmp.data = obj.data.copy()
4             bpy.context.scene.collection.objects.link(tmp)
5
6             [o.select_set(False) for o in bpy.data.objects]
7             tmp.select_set(True)
8             bpy.ops.object.transform_apply(location=True, ↩
                  scale=True, rotation=True)
9             tmp.select_set(False)
10
11            bm = bmesh.new()
12            bm.from_mesh(tmp.data)
13            bm.transform(tmp.matrix_world)
14
15            bpy.data.objects.remove(tmp, do_unlink=True)
16            bm.faces.ensure_lookup_table()
17            bm.verts.ensure_lookup_table()
18            return bm
```

La comprobación de solapamientos entre objetos con el método *overlap*, se realiza creando un *BVHTree* para cada objeto. Posteriormente, aplicamos el método en los dos sentidos. Si tenemos dos *bvhtree* A y B, comprobamos que A no solapa a B, y B no solapa a A. Con esto nos aseguramos de que no existan falsos positivos.

Comprobaciones adicionales al chequeo de colisiones

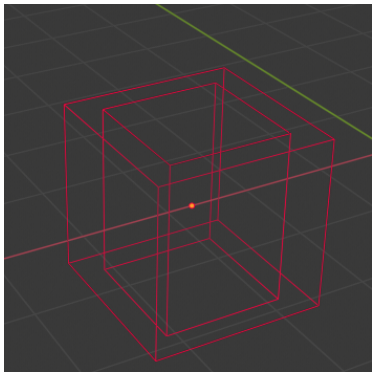
Durante el testeo de la función *overlap* se han detectado algunos casos en los que la función da falsos negativos. Los casos detectados (Figura 35) son causados a que, aparentemente, esta función no considera solapamiento cuando dos caras de dos objetos se tocan.

Para evitar falsos negativos en la validación de rutas, si la función *overlap* resulta ser falsa para dos objetos, comprobamos si existe algún solapamiento entre caras.

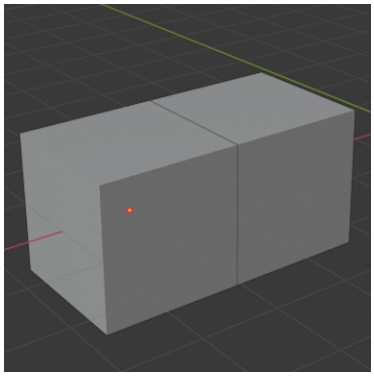
Detección de caras solapadas

La detección de caras solapadas consiste en conocer si dos caras pertenecen al mismo plano y, además, como son finitos, saber si se solapan. (Ver anexo Funciones de chequeo de solapamiento de caras)

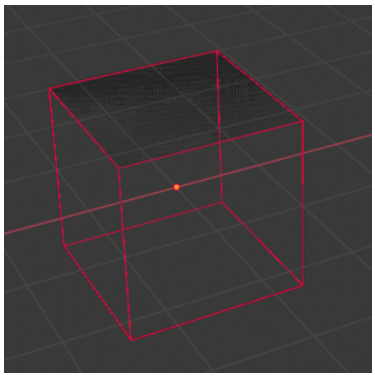
A partir de los *bmesh* generados con la función `create_bmesh` podemos acceder a la información de las caras de un objeto (cuatro vértices del objeto que



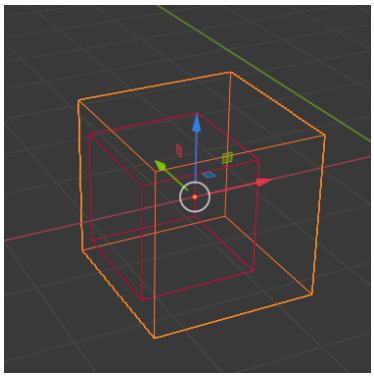
(a) Dos caras de dos objetos solapando



(b) Una cara solapando



(c) Dos objetos del mismo tamaño en la misma posición



(d) Una cara solapando por el interior

Figura 35: Falsos negativos en solapamiento de objetos.

forman la cara y la normal al plano que lo forma). Además, como conocemos la normal $\vec{n}$ y un punto de la cara —tomando cualquier vértice de los 4 que limitan el área de la cara—, podemos calcular la ecuación del plano.

Primero, comprobamos que los planos infinitos sean coplanares. Dos planos son coplanares si sus normales son paralelas y un punto de una pertenece a la otra. Esto lo comprobamos con la función `are_coplanar` (ver anexo H).

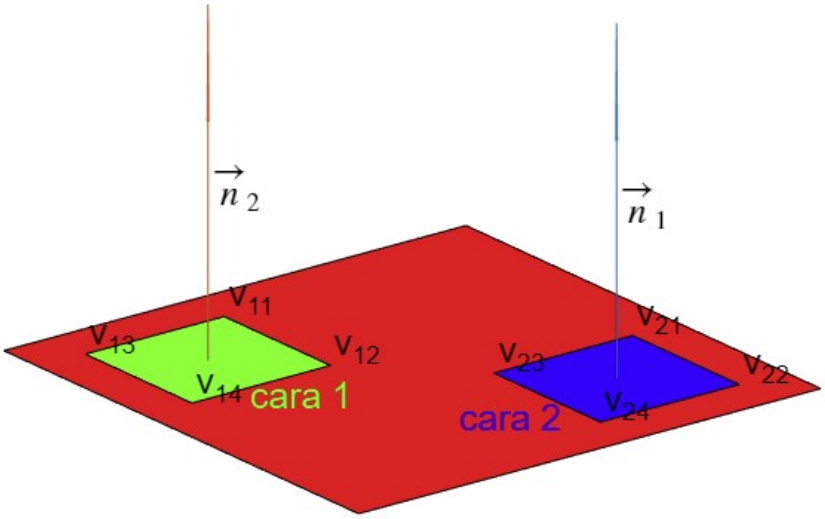


Figura 36: Planos infinitos coplanares.

En caso de que sean coplanares los planos infinitos, debemos comprobar que ambos se solapan. Esto lo hacemos comprobando si alguna arista de una cara está contenida en la otra.

Para comprobar que una arista está contenida en una cara, comprobamos que esto ocurra entre la recta y planos infinitos que los contienen. Un plano contiene a una recta si el vector de la recta y el vector normal del plano son ortogonales (producto escalar a 0) y, además, si algún punto de la recta está en el plano (ver función `infinite_plane_contains_line` en anexo H).

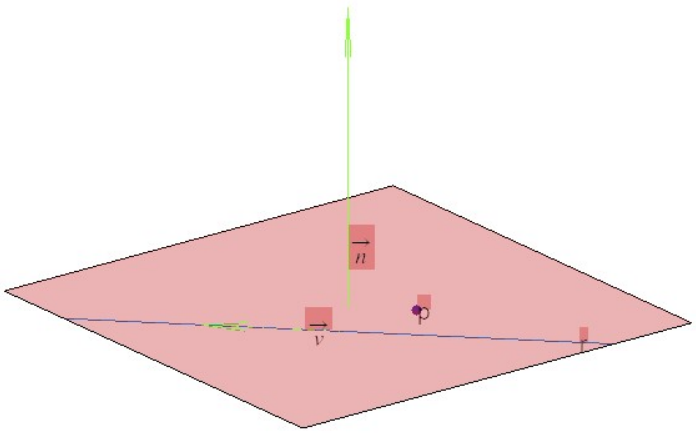


Figura 37: Recta contenida en un plano.

En el caso que esté contenida, comprobamos que esto ocurra entre la arista y la cara (ver función `finite_plane_contains_line` en anexo H).

Una arista está contenida en una cara si, algún punto de la arista se encuentra dentro (ver función `point_inside_finite_plane` en anexo H) o, que alguna arista de la cara corte o coincida con esta arista (ver función `segments_intersect`↩ en anexo H).

Un punto se encuentra dentro de una cara si la distancia entre la reflexión del punto sobre la cara y el punto original se encuentran a una distancia 0. La reflexión de un punto sobre una cara puede calcularse utilizando la función `bmesh.geometry.intersect_face_point` [40].

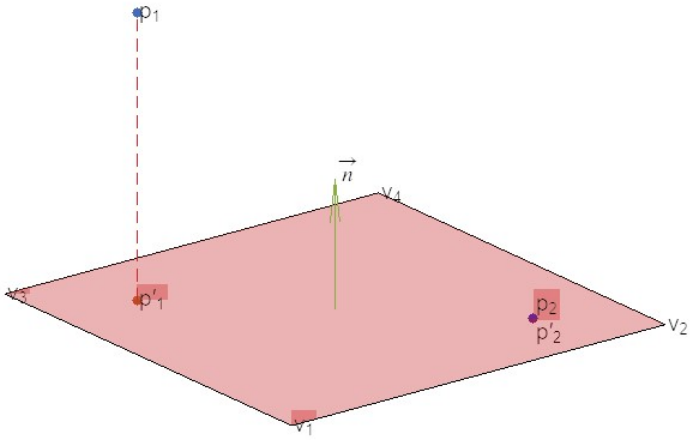
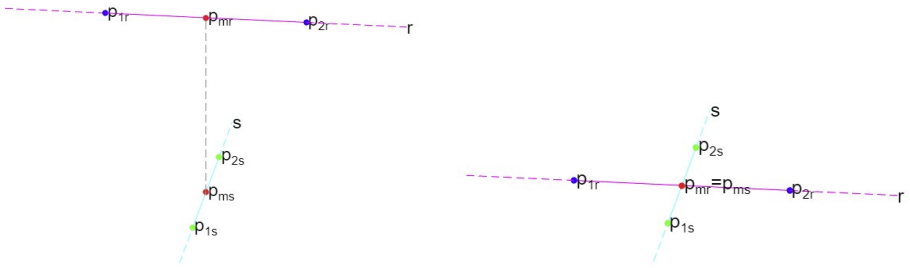


Figura 38: Reflexión de punto en cara.

Para comprobar si dos aristas se cortan o coinciden, utilizamos la función de *Blender* `intersect_line_line` [41], que nos devuelve el punto de cruce de las rectas formadas por dos segmentos dados. Si ese punto de cruce pertenece a ambos, los segmentos se cortan. En el caso de que las aristas sean coincidentes o paralelas, dará el punto más cercano, según la implementación de la función (normalmente una de los vértices de las aristas. Ver funciones `point_in_segment` y `segments_intersect` en anexo H).

Para saber si un punto pertenece a un segmento, calculamos la distancia entre el punto y cada uno de los dos puntos que definen el segmento. La suma debe ser igual a la longitud del segmento. [$dist(p_1, p_2) = dist(p_1, q) + dist(q, p_2)$]



(a) Segmentos que se cruzan (b) Segmentos que intersectan

Figura 39: Intersección de segmentos.

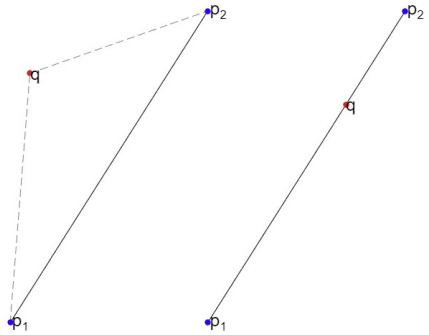


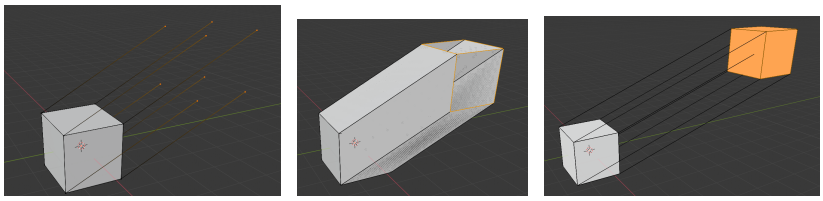
Figura 40: Punto dentro de segmento.

Si ocurre que dos caras de dos objetos coinciden —los planos infinitos son coplanares y además alguna arista de una cara está total o parcialmente contenida en la otra cara—, entonces hemos detectado que dos objetos están coincidiendo por algún lateral (tanto por fuera o por dentro).

Generación del área ocupada por la plataforma en su desplazamiento.

La detección de colisiones en un tramo de una ruta se realiza detectando solapamiento entre los obstáculos de la escena y el área que ocupa el robot durante su desplazamiento (Figura 33). En definitiva, la detección consiste en generar este área, y comprobar si existe algún solapamiento con algún objeto de la escena, utilizando el método anteriormente descrito.

Utilizando la operación *extrude* de *Blender*, podemos calcular el área fácilmente. *Extrude* es una operación que permite duplicar los datos de un objeto (vértices, caras, aristas), desplazándolos a algún punto. Los datos duplicados siguen perteneciendo al objeto, y se unen a los originales. Por ejemplo, si extruimos un vértice y lo desplazamos a un punto, existirá una arista que una el vértice original con el duplicado [42].



(a) Extrude de vértices (b) Extrude de arístas (c) Extrude de caras

Figura 41: Extrude.

La operación de extruír permite seleccionar el punto final hasta donde se extenderá la extrusión. Tomando como objeto para extruir al margen de seguridad de la plataforma, lo copiamos y colocamos en la pose de inicio del tramo que deseamos verificar. Posteriormente extruímos las caras y las arístas, marcando como posición final la posición de la pose de destino del tramo. Como resultado obtenemos un ortoedro que representa el área en el cual ningún objeto puede estar.

Para verificar la trayectoria, comprobamos que ningún objeto se solape con el área generada. Posteriormente, este área se elimina.

Listing 11: Función *generate_area*.

```
1
2       def generate_area(robot_obj, pos0, pos1):
3           """
4           Input:
5           - robot_obj : object that represents robot
6           - loc0 (Vector) : start location of path to check
7           - loc1 (Vector) : end location of path to check
8           returns:
9           - name of mesh created
10          """
```

```
11          offset = (pos1.loc - pos0.loc)
12
13          bpy.ops.object.select_all(action='DESELECT')
14          bpy.context.view_layer.objects.active = robot_obj
15          robot_obj.select_set(True)
16
17          bpy.ops.object.duplicate()
18
19          copy_obj = bpy.context.active_object
20          copy_obj.location = pos0.loc
21
22          bpy.ops.object.editmode_toggle()
23
24          bpy.ops.mesh.extrude_region_move(↩
                MESH_OT_extrude_region={"use_normal_flip":False↩
                , "mirror":False},
25          TRANSFORM_OT_translate={"value":offset[:],
26          "orient_type":'NORMAL',
27          "orient_matrix":((1, 0, 0),
28          (0, 1, 0),
29          (0, 0, 1)),
30          "orient_matrix_type":'NORMAL',
31          "constraint_axis":(False, False, True),
32          "mirror":False, "use_proportional_edit":False,
33          "proportional_edit_falloff":'SMOOTH',
34          "proportional_size":1,
35          "use_proportional_connected":False,
36          "use_proportional_projected":False,
37          "snap":False,
38          "snap_target":'CLOSEST',
39          "snap_point":(0, 0, 0),
40          "snap_align":False,
41          "snap_normal":(0, 0, 0),
42          "gpencil_strokes":False,
43          "cursor_transform":False,
44          "texture_space":False,
45          "remove_on_cancel":False,
46          "release_confirm":False,
47          "use_accurate":False})
48
49
50          bpy.ops.mesh.extrude_edges_move(↩
                MESH_OT_extrude_edges_indiv={"use_normal_flip":↩
                False, "mirror":False},
51          TRANSFORM_OT_translate={"value":(-offset)[:],
52          "orient_type":'GLOBAL',
53          "orient_matrix":((1, 0, 0),
54          (0, 1, 0),
55          (0, 0, 1)),
56          "orient_matrix_type":'GLOBAL',
57          "constraint_axis":(False, False, False),
58          "mirror":False,
59          "use_proportional_edit":False,
60          "proportional_edit_falloff":'SMOOTH',
```

```
61          "proportional_size":1,
62          "use_proportional_connected":False,
63          "use_proportional_projected":False,
64          "snap":False,
65          "snap_target":'CLOSEST',
66          "snap_point":(0, 0, 0),
67          "snap_align":False,
68          "snap_normal":(0, 0, 0),
69          "gpencil_strokes":False,
70          "cursor_transform":False,
71          "texture_space":False,
72          "remove_on_cancel":False,
73          "release_confirm":False,
74          "use_accurate":False})
75
76          bpy.ops.object.editmode_toggle()
77
78          return copy_obj
```

Esta operación la aplicamos cada vez que agregamos una pose en una ruta que estémos creando, entre la pose anterior y la nueva añadida. Solamente se tiene en cuenta la rotación de la pose de inicio de la ruta, no se tienen en cuenta las rotaciones que realice la plataforma mientras se traslada. Además, al ser esta una operación costosa, no se aplica al enviarse una ruta a la plataforma robótica, pues el tiempo de demora aumentaría a medida que aumenta el número de poses. En un futuro podría mejorarse este algoritmo para ser aplicado dinámicamente mientras se ejecuta un plan, y cancelar el plan automáticamente.

9.7. Módulo de comunicación

Este módulo contiene todas las funcionalidades necesarias para realizar una comunicación con una plataforma robótica.

Como se ha indicado anteriormente, actualmente la comunicación con la plataforma robótica es simulada. Es decir, se dispone de un servidor en el equipo local que simula al servidor que debería encontrarse en una plataforma real.

Como protocolo de comunicación se ha decidido utilizar el protocolo de transporte UDP, debido a que en este no se necesita realizar una conexión, y al no tener control de errores, los tiempos de esperas son mucho menores [43]. Por el mismo motivo, se ha necesitado diseñar un conjunto de tipo de paquetes, estructurados de modo que pueda realizarse un detección manual de errores.

Las funcionalidades implementadas en este módulo están relacionadas con el envío de rutas a la plataforma robótica, el control de su ejecución y su seguimiento a tiempo real. Concretamente, las funcionalidades implementadas son:

- Cambio de modo : entre modo plataforma robótica o modo editor. Sirve para conectarse o desconectarse del robot, entendiendo como conectarse al envío de un paquete que indica al servidor que el cliente está escuchando

en el puerto específico (no una conexión real como podría entenderse en otro tipo de protocolos).

- Envío de plan : se envían las poses de un plan a la plataforma robótica.
- Inicio de un plan : se indica a la plataforma que inicie un plan cargado previamente.
- Pausa y reinicio de un plan: se interrumpe la ejecución de un plan y se reanuda desde el último punto en el que se estaba.
- Cancelar plan: se cancela el plan. La plataforma deberá realizar el plan desde el inicio.
- Cambio de velocidad: se puede cambiar la velocidad de la plataforma entre un porcentaje mínimo y máximo.

Adicionalmente, en este módulo se ha incluido un simulador de rutas. Sin realizar ningún tipo de comunicación, se simula el comportamiento que debería tener la plataforma durante la ejecución. En este se incluyen todas las funcionalidades que existen en el módulo de control real.

9.7.1. Paquetes y serializado

Para realizar la comunicación se han diseñado un conjunto de paquetes, con los que enviar órdenes al servidor, recibir información y realizar un seguimiento de paquetes y control de errores.

Podemos diferenciar los paquetes en dos grupos: los que envía y los que recibe el cliente desde el servidor.

El servidor recibe paquetes de cambio de modo, de cambio de velocidad, de creación de ruta y de inicio, pausa, continuar y cancelar plan. El cliente, en cambio, recibe paquetes de seguimiento y de pose alcanzada. En común, se encuentra el paquete de reconocimiento, que es enviado tanto por el cliente como por el servidor.

Paquetes comunes

Todos los paquetes tienen una estructura básica compartida: un identificador (un entero largo) y un tipo de paquete (un entero).

El paquete de reconocimiento es enviado tanto por el servidor como por el cliente, para confirmar que ciertos paquetes han sido recibidos.

La estructura del paquete de reconocimiento es:

- Identificador.
- Tipo de paquete : cuyo valor es 1.

- Paquete reconocido : identificador del paquete el cual se está reconociendo.
- Estado: corrupto (valor 0) o correcto (valor 1).

Paquetes enviados al cliente

Al cliente son enviados dos tipos de paquetes: paquetes de seguimiento y de pose de ruta alcanzada.

Los paquetes de seguimiento son utilizados para que el cliente conozca la pose actual de la plataforma robótica en cada momento. Su estructura es:

- Identificador.
- Tipo de paquete: 3
- Pose actual: x, y, θ.

Los paquetes de pose de ruta alcanzada son enviados por el servidor mientras se ejecuta un plan, cuando la plataforma ha alcanzado una pose marcada en la ruta. Su estructura es la misma que la de los paquetes de seguimiento:

- Identificador.
- Tipo de paquete: 11
- Pose actual: x, y, θ.

Este paquete requiere que el cliente envíe un reconocimiento.

Paquetes enviados al servidor

Al servidor son enviados: paquetes de cambio de modo, paquetes de abrir plan, agregar pose al plan, cerrar plan, iniciar, pausar, continuar y cancelar plan, y cambiar velocidad de la plataforma.

El paquete de cambio de modo permite al servidor conocer cuando el cliente se encuentra escuchando el puerto. Existen dos modos: modo editor y modo *RoboMap*, en el cual se lleva a cabo la comunicación. La estructura de un paquete de modo es:

- Identificador.
- Tipo de paquete: 2
- Modo: 0 en modo editor y 1 en modo *RoboMap*
- Velocidad inicial.

Este paquete requiere que el servidor envíe un reconocimiento.

Para enviar un plan a la plataforma robótica se deben enviar 3 tipos de paquetes: un paquete de abrir plan, con el que se indica que se va a comenzar a enviar poses de un plan al servidor; tantos paquetes como poses existan en la ruta; y un paquete de cerrar plan, con el que se indica que se termino de cargar el plan. La estructura de cada uno de estos paquetes es:

Paquete de abrir plan

- Identificador.
- Tipo de paquete: 4
- Número de poses.

Este paquete requiere que el servidor envíe un reconocimiento.

Paquete de agregar pose

- Identificador.
- Tipo de paquete: 5.
- Pose: x, y, θ.

Este paquete requiere que el servidor envíe un reconocimiento.

Paquete de cerrar plan

- Identificador.
- Tipo de paquete: 6

Este paquete requiere que el servidor envíe un reconocimiento.

El paquete de iniciar plan permite que el cliente indique al servidor que arranque un plan que haya sido cargado previamente. La estructura del paquete es:

- Identificador.
- Tipo de paquete: 7

Este paquete requiere que el servidor envíe un reconocimiento.

El paquete de pausar navegación indica a la plataforma robótica que se pare en la posición que se encuentra, hasta que se le envíe un paquete de continuar plan. En ese momento, la plataforma continúa el plan desde donde se

encuentra. La estructura de estos paquetes es:

Paquete de pausar plan

- Identificador.
- Tipo de paquete: 9

Este paquete requiere que el servidor envíe un reconocimiento.

Paquete de continuar plan

- Identificador.
- Tipo de paquete: 10

Este paquete requiere que el servidor envíe un reconocimiento.

Cuando se le envía al servidor un paquete de cancelar plan, la plataforma robótica se detiene. En ese momento, el plan se descarta, y debe ser vuelto a iniciar desde el comienzo. La estructura del paquete es:

- Identificador.
- Tipo de paquete: 8

Este paquete requiere que el servidor envíe un reconocimiento.

La velocidad de la plataforma es controlada utilizando paquetes de cambio de velocidad. En estos paquetes se indica un porcentaje de velocidad, que es convertido en el servidor a una velocidad según un valor máximo. La estructura de este paquete es:

- Identificador.
- Tipo de paquete: 12
- Porcentaje de velocidad máxima : entre 0 % y 100 %

Este paquete requiere que el servidor envíe un reconocimiento.

Serialización de paquetes

El tamaño de los paquetes es un factor importante en la comunicación, determinando la carga en la red. Para conseguir unos paquetes más pequeños se han estudiado algunas alternativas, eligiéndose finalmente el formato de serialización *msgpack* [44]. *Msgpack* es un formato de serialización disponible para múltiples lenguajes. Según afirman en la página oficial, comparado con *JSON*,

es capaz de reducir el número de bytes necesarios a aproximadamente el 50 %.

En *Python*, la implementación de esta serialización se encuentra en el módulo `msgpack`[45]. Este módulo no se encuentra con la instalación de *Python* distribuida junto a *Blender*, por lo que debe instalarse utilizando el comando `pip`.

Para serializar los paquetes, se generan listas con los valores de cada campo en cada elemento. Utilizando las funciones de `msgpack`, podemos convertir un paquete a una ristra de bytes, y viceversa (Figura 12).

Listing 12: MsgPack Python Serializacion.

```
1         # lista de paquetes
2         packet = [field1, field2, ...]
3
4         # Serializar paquete
5         res = msgpack.packb(packet, use_bin_type=True)
6
7         # Deserializar paquete
8         msgpack.unpackb(res, raw=False)
```

Para llevar a cabo el proceso de serialización (Figura 42) de los paquetes se ha creado un esquema de clases inspirado en el patrón *strategy* [46].

Cada tipo de paquete se encuentra implementado en una clase, que heredan de la clase `Packet` los campos comunes `pid` (identificador de paquete) y `ptype` (tipo de paquete). Para cada tipo de paquete existe un modo de serializar, implementado en las subclases de `Serialization`. Estas clases poseen dos métodos: `pack`, que recoge una clase `Packet` y la convierte a una lista de valores, que se formateará a `msgpack`; y un método `unpack`, que a partir de una lista previamente extraída del paquete en formato `msgpack`, la convierte a una clase `Packet` concreta.

Externamente, gracias a la clase `Serializator`, no es necesario conocer el tipo de paquete a serializar. Esta clase se encarga de reconocer el tipo de paquete y seleccionar la clase `Serialization`, según el valor de tipo de paquete. En *Python*, con un diccionario que almacena el tipo de paquete y la clase `Serialization` necesaria, la aplicación de la serialización se convierten en tres instrucciones: extraer el valor del tipo de paquete recibido, buscar según este valor la clase en el diccionario, y aplicar la serialización.

Listing 13: Serialización y deserialización de paquetes (*Serializator*).

```
1     choose_serialization = {
2         1:  AckPacketMsgPackSerialization,
3         2:  ModePacketMsgPackSerialization,
4         3:  TracePacketMsgPackSerialization,
5         4:  OpenPlanPacketMsgPackSerialization,
6         5:  AddPosePlanPacketMsgPackSerialization,
7         6:  ClosePlanPacketMsgPackSerialization,
8         7:  StartPlanPacketMsgPackSerialization,
9         8:  StopPlanPacketMsgPackSerialization,
```

```
10          9:  PausePlanPacketMsgPackSerialization,
11          10: ResumePlanPacketMsgPackSerialization,
12          11: ReachedPosePacketMsgPackSerialization,
13          12: ChangeSpeedPacketMsgPackSerialization
14      }
15
16      class MsgPackSerializator(st.Serializator):
17
18          @staticmethod
19          def pack(packet: st.Packet) -> bytes:
20              cipher_method = choose_serialization.get(packet↩
                    .ptype) # Devuelve la clase según el valor ↩
                    de tipo de paquete (ptype)
21              if cipher_method is None: # get devuelve None ↩
                    si no existe el valor en la lista
22                  raise "Error: Serialization method not ↩
                        found (pack). Check packet type ↩
                        identification"
23              values = cipher_method.pack(packet) # Se ↩
                    convierte a lista de valores la clase ↩
                    Packet (implementado en el método __iter__ ↩
                    de la clase Packet)
24              return msgpack.packb(values, use_bin_type=True)↩
                     # Se convierte a formato msgpack
25
26          @staticmethod
27          def unpack(byte_packet: bytes) -> st.Packet:
28              values = msgpack.unpackb(byte_packet) # Se ↩
                    extrae la lista del mensaje recibido
29              ptype = values[1]
30              decipher_method = choose_serialization.get(↩
                    ptype) # El segundo valor de la lista ↩
                    indica el tipo. Se busca la clase nesearia
31              if decipher_method is None:
32                  raise "Error: Serialization method not ↩
                        found (unpack). Check packet type ↩
                        identification"
33              return decipher_method.unpack(values) # unpack ↩
                    crea la clase según el tipo de paquete
```

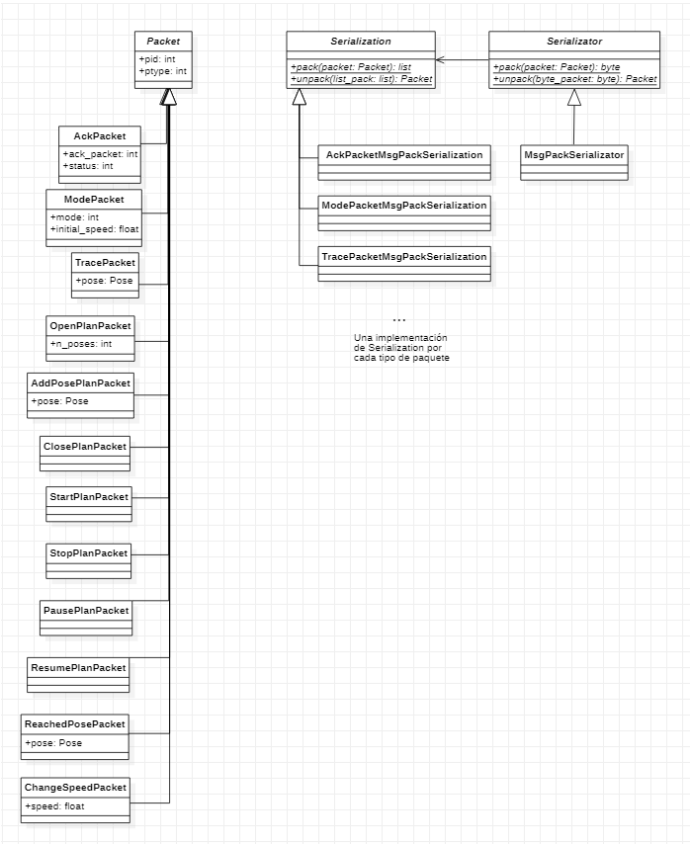


Figura 42: Serialización UML.

9.7.2. Esquema de comunicación

La comunicación UDP en *Python* se realiza utilizando el módulo `socket`. Con este módulo podemos crear un *socket* con el que escuchar el puerto por el que se comunica el servidor, para así enviar y recibir paquetes [47].

El *socket* es creado al ejecutarse la orden de cambio de modo, antes de enviar el paquete de cambio a modo *RoboMap*. Todas las operaciones de comunicación con el servidor, incluida la creación y eliminación del `socket`, se encuentran implementadas en la clase *singleton* `ConnectionHandler` 45. Esta clase sirve de interfaz, ocultando los detalles del envío y recepción de paquetes.

El gestor principal de la comunicación es el operador modal `SocketModalOperator`↩. Un operador modal es similar a los operadores básicos que se han explicado en apartados anteriores, pero con la diferencia de que estos pueden ejecutarse más de una vez. Utilizando el método `modal`, el operador es capaz de captar eventos de la aplicación y ejecutar cierto código que exista en su interior. Igual que otros operadores, también posee el método `execute`, que se ejecuta cuando lanzamos por primera vez al operador. Al retornar el valor `{'RUNNING_MODAL'}` en vez de `{'FINISHED'}`, como hacemos con los operadores que se ejecutan una única vez, provocamos que el operador quede escuchando eventos de la aplicación [48].

Listing 14: Ejemplo de operador modal.

```
1
2       class ModalOperator(bpy.types.Operator):
3           bl_idname = "object.modal_operator"
4           bl_label = "Simple Modal Operator"
5
6           def __init__(self):
7               print("Start")
8
9           def __del__(self):
10              print("End")
11
12          def cancel(self, context):
13              self.report({'INFO'}, "Cancelled")
14
15          def execute(self, context):
16              context.object.location.x = self.value / 100.0
17              return {'FINISHED'}
18
19          def modal(self, context, event):
20              if event.type == 'MOUSEMOVE':  # Apply
21                  self.value = event.mouse_x
22                  self.execute(context)
23              elif event.type == 'LEFTMOUSE':  # Confirm
24                  return {'FINISHED'}
25              elif event.type in {'RIGHTMOUSE', 'ESC'}:  # ↩
                    Cancel
26                  context.object.location.x = self.init_loc_x
27                  return {'CANCELLED'}
28
29              return {'RUNNING_MODAL'}
30
31          def invoke(self, context, event):
32              self.init_loc_x = context.object.location.x
33              self.value = event.mouse_x
34              self.execute(context)
35
36              context.window_manager.modal_handler_add(self)
37              return {'RUNNING_MODAL'}
```

En el ejemplo 14 se puede observar que se utiliza el método `invoke` en vez de `execute` para iniciar el operador modal.

Dentro de un operador modal, tenemos tres opciones después de ejecutar el código: seguir con la ejecución del operador, retornando `{'RUNNING_MODAL'}`; terminar el operador confirmando las acciones, retornando `{'FINISHED'}`; o cancelar el operador retornando `{'CANCELLED'}`. Al cancelar se ejecuta el método `cancel`.

SocketModalOperator (Figura 45) se encarga de la creación y eliminación

del *socket* de comunicación, además de recibir los paquetes desde el servidor y mover a la plataforma a la última pose recibida.

Los paquetes son recibidos independientemente por un hilo, creado al inicializar el operador modal. Este hilo se encarga de ejecutar, dentro de un bucle, el método `ConnectionHandlet.receive_packet`, el cual recibe paquetes, los deserializa, y los almacena en la clase *singleton* `Buffer`. Para evitar conflictos, este hilo será el único que llamará al método `receive_packet`.

En `Buffer` se almacenan todos los paquetes que llegan, y que son posteriormente leídos en otras operaciones. Los paquetes se almacenan según su tipo. Los paquetes de pose alcanzada se almacenan en una lista. De todos los paquetes de seguimiento recibidos, se almacena siempre el último que ha llegado, pues es el único que interesa para dibujar al robot en la posición actual. El resto de paquetes son almacenados en una lista común.

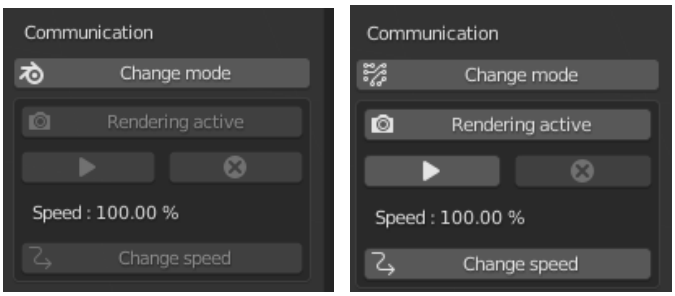
El operador `SocketModalOperator` se arranca al ejecutar el operador `ChangeModeOperator`, que llama al operador modal `bpy.ops.wm.socket_modal('INVOKE_DEFAULT')`. Anteriormente, se comprueba que no se haya lanzado antes, utilizando el flag `SocketModalOperator.closed`.

Listing 15: Operador *ChangeModeOperator*.

```
1         class ChangeModeOperator(bpy.types.Operator):
2             bl_idname = "wm.change_mode"
3             bl_label = "Change mode"
4             bl_description = "Change between robot / editor"
5
6             @classmethod
7             def poll(cls, context):
8                 running_plan = context.scene.com_props.prop_running_nav
9                 selected_robot = context.scene.selected_robot_props.prop_robot_id >= 0
10                changing_mode = SocketModalOperator.switching
11                active_editor = context.scene.is_cursor_active
12                return not active_editor and not running_plan and selected_robot and not changing_mode
13
14            def execute(self, context):
15                if SocketModalOperator.closed:
16                    bpy.ops.wm.socket_modal('INVOKE_DEFAULT')
17                else:
18                    SocketModalOperator.closed = True
19                return {'FINISHED'}
```

Para cambiar de modo, se ha incluido en el panel de control, en el panel

robotcontrol, un pulsador, que además indica en que modo se encuentra actualmente (Figura 43).



(a) En modo editor (b) En modo robomap

Figura 43: Cambio de modo.

Al inicializar el operador SocketModalOperator, se crea el hilo y un timer, que nos servirá como evento para ejecutar cada cierto tiempo el método `modal`.

Dentro del método `modal`, cambiamos la posición del robot, tomando el último paquete de seguimiento almacenado en `Buffer`. En caso que se haya desactivado el renderizado de la posición del robot, no se actualiza la posición del objeto robot. (Figura 44). Adicionalmente, comprobamos que el flag `closed` no haya sido cambiado por `ChangeModeOperator`. Si es el caso, se envía un paquete de modo al servidor y, una vez recibido el reconocimiento (almacenado en *Buffer*), se cierra el hilo.

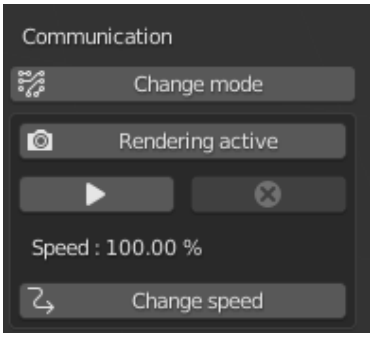


Figura 44: Activar renderizado.

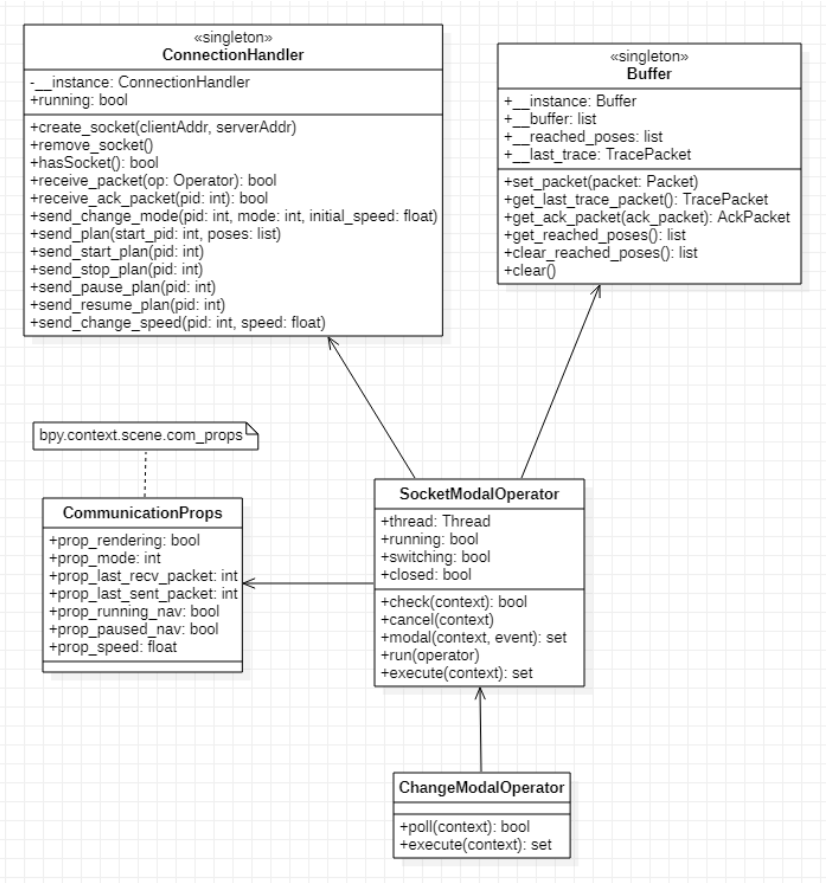


Figura 45: **SocketModalOperator** UML.

9.7.3. Operaciones en la comunicación

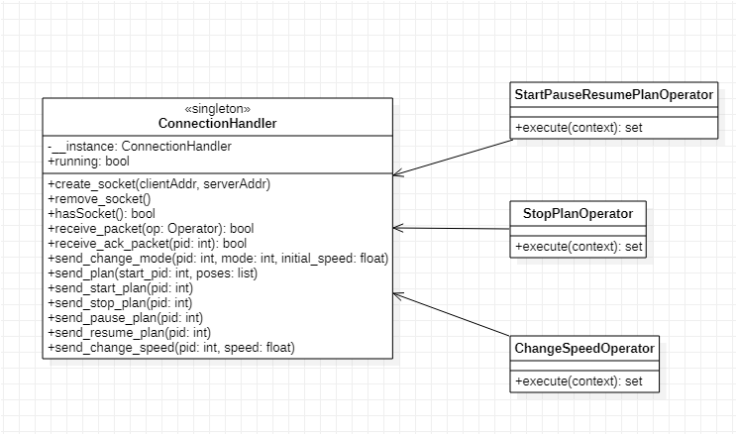
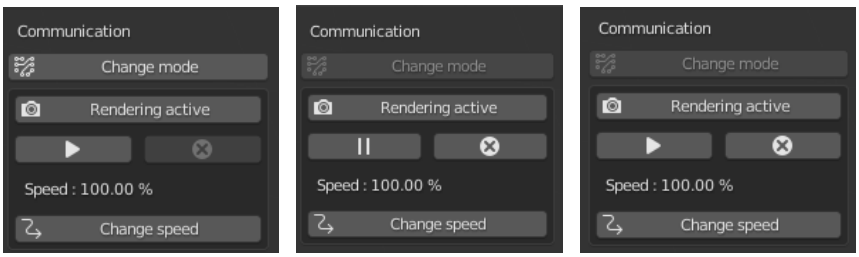


Figura 46: Operaciones de comunicación UML.

Enviar ruta: iniciar, pausar, continuar y cancelar plan

El control de ejecución de rutas se realiza utilizando el panel de control que se habilita al cambiar a modo *RoboMap* (Figura 47).



(a) Iniciar plan (b) Pausar y cancelar plan (c) Continuar y cancelar plan

Figura 47: Iniciar, pausar, continuar y cancelar plan.

El botón de *play* (Figura 47a) es utilizado para enviar planes de navegación creados. Para iniciar un plan, debe existir uno creado. El operador `StartPauseResumePlanOperator` se encarga de controlar el envío, pausa y renaudación de planes.

En la figura 48 podemos ver el control que se lleva en el operador. Para el control de estados de ejecución del plan, se crearon propiedades, que se encuentran en el grupo de propiedades *CommunicationProps*. La propiedad `prop_running_nav` indica que se ha iniciado un plan y `prop_paused_nav`, que ha sido pausado. La propiedad `prop_last_path_update` almacena una marca de tiempo, que indica la última actualización del plan de navegación.

Listing 16: Propiedades de comunicación.

```
1     class CommunicationProps(bpy.types.PropertyGroup):
2         prop_last_path_update: bpy.props.IntProperty(name="↩
              last_path_update", default=-1, min=-1)
3
4         prop_running_nav: bpy.props.BoolProperty(name="↩
              Running nav", default=False)
5         prop_paused_nav: bpy.props.BoolProperty(name="↩
              Paused nav", default=False)
6
7         prop_speed: bpy.props.FloatProperty(name="Speed", ↩
              default=100.0, min=0.0, max=100.0)
8
9         prop_rendering: bpy.props.BoolProperty(name="↩
              Rendering", default=True)
10
11        prop_mode : bpy.props.IntProperty(name="mode", ↩
              default=0, min=0, max=len(robot_modes_summary)↩
              -1)
```

```
12          prop_last_recv_packet : bpy.props.IntProperty(name=↩
                "last_recv_packet", default=-1, min=-1)
13          prop_last_sent_packet : bpy.props.IntProperty(name=↩
                "last_sent_packet", default=-1, min=-1)
```

Como UDP no tiene control de orden, es posible que se reciban paquetes en distinto orden a como se enviaron. Utilizando las propiedades `prop_last_recv_packet` y `prop_last_sent_packet` llevamos un seguimiento del identificador de los paquetes enviados y recibidos, descartando todo paquete con identificador menor al que tenemos.

En el apartado 9.6.1 se explicó como se almacenan en la clase `PathContainer` las rutas creadas con el editor de rutas. Cada vez que se actualiza el contenido de `PathContainer` con el editor, añadiendo o eliminando `actions`, se actualiza una variable que recoge la marca de tiempo en la que se realizó el cambio. Almacenando en `prop_last_path_update` la marca de tiempo de la última ruta enviada, podemos conocer si se ha actualizado el contenido de `PathContainer`.

Para enviar un plan a la plataforma, se recogen las poses de `PathContainer` y se envían utilizando el método `ConnectionHandler.send_plan`. En este método, se envía un paquete de abrir plan, tantos paquetes de agregar pose a la ruta como poses se quieran enviar, y un paquete de cerrar plan. Se envían rutas a la plataforma en dos casos: cuando no existe un plan en ejecución (`prop_running_nav` es falso) o, cuando hay un plan pausado y se ha actualizado (`prop_running_nav` es verdadero, `prop_paused_nav` es verdadero y el valor de `prop_last_path_update` es menor al valor actual en `PathContainer`).

El pausado de un plan se lleva a cabo enviando a la plataforma un paquete de pausa, con el método `ConnectionHandler.send_pause_plan`. Un plan puede ser pausado si un plan se está ejecutando (`prop_running_nav` es verdadero).

Cuando un plan está pausado, para reanudarlo, se envía con `ConnectionHandler.send_resume_plan` un paquete de reanudar plan. Un plan puede reanudarse cuando se está ejecutando un plan, se encuentra pausado y, además, no se ha actualizado. (`prop_running_nav` es verdadero, `prop_paused_nav` es verdadero y el valor de `prop_last_path_update` es igual al valor actual en `PathContainer`).

Tanto cuando un plan está pausado como no, pero en ejecución, puede cancelarse. Al cancelarse, se envía un paquete de cancelar plan a la plataforma, con el método `ConnectionHandler.send_stop_plan`.

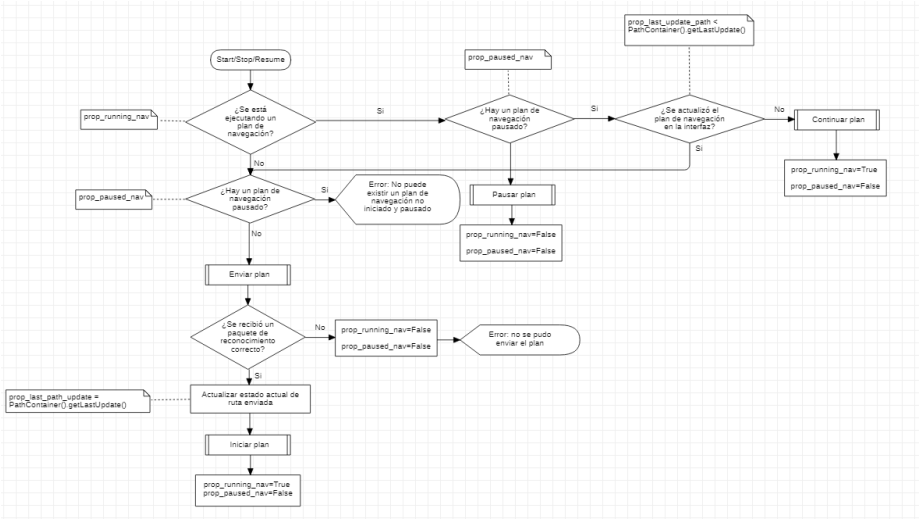


Figura 48: Botón *play* (control de flujo).

Cambiar velocidad

La velocidad de la plataforma es establecida en dos momentos: cuando se cambia de modo y cuando se solicita un cambio de velocidad. Cuando se cambia de modo, dentro del paquete de modo, se incluye un campo que permite indicarle a la plataforma la velocidad inicial que debe tomar. Esto permite al usuario tener un mayor control en la velocidad, evitando que, al iniciar un plan, exista una velocidad demasiado alta o demasiado baja a la que se requiere.

Al cambiar de modo editor a modo plataforma robótica, se abre un diálogo emergente, que solicita una velocidad inicial, que será la que se envíe en el paquete de modo. En cambio, cuando se cambia de modo plataforma a modo editor, no se solicita ninguna velocidad, aunque, como el paquete de modo requiere este campo, se envía la última almacenada.

Durante la ejecución de un plan de navegación, también puede cambiarse la velocidad de la plataforma, con la opción que hay en la interfaz para ello (Figura 49).

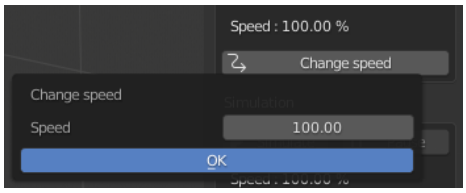


Figura 49: Cambio de velocidad.

El valor de la velocidad deseada (no la real que tomará la plataforma) tiene un valor entre el 0 % y el 100 % de la velocidad máxima real de la plataforma.

Cuando el usuario decide cambiar la velocidad con el *slider*, se almacena el valor en la propiedad `prop_speed` (Figura 16). Cuando el usuario pulsa el botón *OK*, se termina de ejecutar el operador, que se lanza al pulsar en *change speed*. En ese momento, se envía un paquete de cambio de velocidad, con el método `ConnectionHandler.send_change_speed`. Si se recibe un reconocimiento de que se cambió correctamente la velocidad, entonces se actualiza en la interfaz el *label* que muestra la velocidad actual.

El operador que se utiliza para cambiar la velocidad es `ChangeSpeedOperator`. En este, con el método `invoke`, se abre la ventana de diálogo que solicita la velocidad, utilizando como entrada una propiedad interna a la clase llamada `update_speed`, para así mantener la velocidad antes de la actualización. Posteriormente, cuando el usuario cambia la velocidad con el *slider*, se envía el paquete y se comprueba que la operación fue correcta. Si es así, se actualiza la velocidad en `prop_speed`.

Listing 17: Operador cambio de velocidad.

```
1     class ChangeSpeedOperator(bpy.types.Operator):
2         bl_idname = "wm.change_speed"
3         bl_label = "Change speed"
4         bl_description = "Send speed"
5
6         update_speed: bpy.props.FloatProperty(name="Speed",↩
              min=0.0, max=100.0, default=0.0)
7
8         @classmethod
9         def poll(cls, context):
10            return SocketModalOperator.running
11
12        def invoke(self, context, event):
13            self.update_speed = bpy.context.scene.com_props↩
                  .prop_speed
14            wm = context.window_manager
15            return wm.invoke_props_dialog(self)
16
17        def execute(self, context):
18            self.report({'INFO'}, str(self.update_speed))
19
20            older_speed = context.scene.com_props.↩
                  prop_speed
21            curre_speed = self.update_speed
22            context.scene.com_props.prop_last_sent_packet ↩
                  += 1
23            pid = context.scene.com_props.↩
                  prop_last_sent_packet
24            if not cnh.ConnectionHandler().↩
                  send_change_speed(pid, curre_speed):
25                self.report({"ERROR", "Speed can not be ↩
                      changed"})
26            else:
27                context.scene.com_props.prop_speed = ↩
```

```
                    curre_speed
28              return {'FINISHED'}
```

La razón por la que se implementó con un botón intermedio, y no incluyendo el *slider* directamente en el panel, es porque de ese modo, cada vez que se produce un cambio en el *slider* se enviaría un paquete de cambio de velocidad, saturando la comunicación innecesariamente. Las propiedades, como vimos en el apartado introductorio 9.1, para detectar actualizaciones solamente poseen los métodos `set` y `update`, que se ejecutan cada vez que se produce un cambio en el valor de la propiedad.

9.8. Salida segura de la aplicación

Mientras el servidor se encuentra en modo plataforma robótica, es necesario que el cliente se encuentre activo, pues es el único modo con el que se tiene el control de su comportamiento y estado. Por lo tanto, es crítico que cuando el usuario quiera salir de la aplicación, se informe a la plataforma de que el cliente no va a estar disponible.

Los operadores en Blender poseen el método `cancel`, que se ejecuta en dos casos: cuando el usuario cancela la ejecución de un operador o cuando se cierra la aplicación. Normalmente, este método se implementa únicamente en los operadores modales, pues son los más susceptibles de estar en ejecución cuando el usuario cierre la aplicación.

En el caso de este *addon*, para evitar un cierre brusco de la comunicación, se ha sobrescrito en el operador `SocketModalOperator` el método `cancel`. En este comprobamos que no se este ejecutando ningún plan y que no se encuentre en modo plataforma robótica. Si es el caso, se cancela cualquier plan activo y se cambia a modo editor.

9.9. Simulación

En ocasiones es necesario visualizar como se comportará la plataforma robótica durante la ejecución de un plan de navegación. Para ello, se ha creado un simulador de rutas. Sus funcionalidades son las mismas que las que se encuentran en el panel de control de la plataforma robótica, que se han explicado en los anteriores apartados: inicio de un plan, pausa, continuación, cancelación y cambio de velocidad. Sin embargo, no se realiza ningún tipo de comunicación con la plataforma, el comportamiento es completamente simulado.

Hay que tener en cuenta que la simulación es incompatible con la comunicación con la plataforma, por lo que se debe estar en modo editor para poder simular un plan. Esto se debe a que, mientras se encuentra en modo plataforma robótica, la posición del objeto robot es actualizada continuamente, por lo que interferiría en la simulación.

Para poder simular un plan, primeramente debe existir algún plan creado. Una vez exista un plan, se habilitarán las herramientas en el panel de simulación. 50

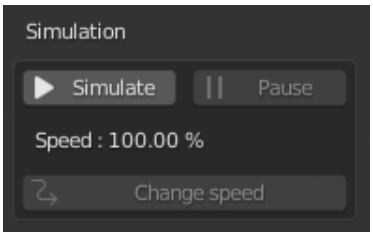


Figura 50: Panel de simulación.

La simulación está implementada utilizando un operador modal, similar al que se usó en la comunicación en el apartado 9.7.2. El operador modal `SimulationOperator`, al inicializarse arranca un *timer*, que sirve como evento para ejecutar el método `modal`. En cada ciclo, se actualiza la pose del objeto robot seleccionado, según el trazo de ruta que se este realizando.

Simulación del movimiento

Para simular el movimiento de la plataforma por la escena, en cada ciclo del *timer*, se actualiza la posición de la plataforma, según la dirección de la trayectoria del tramo de ruta que se está realizando, y un factor de velocidad.

La trayectoria entre dos poses es una línea recta, por lo que podemos calcular la dirección utilizando el punto de inicio y final. Según el porcentaje de velocidad marcado por el usuario y una velocidad máxima, se actualiza la posición en cada ejecución del método `modal` como:

$$P_{t+1}(x,y) = P_t(x,y) + \frac{speed_{percentage} * speed_{max}}{100} * \hat{\mathbf{d}} \tag{1}$$

donde

- P_t es la posición de la plataforma en el instante t
- $\hat{\mathbf{d}}$ es el vector director unitario entre el punto de inicio $P1$ y fin $P2$.
- $\hat{\mathbf{d}} = \frac{\vec{d}}{\|\vec{d}\|}$ donde $\vec{d} = P2 - P1$
- $speed_{percentage}$ es el porcentaje de velocidad introducido por el usuario.
- $speed_{max}$ es una constante que indica la velocidad máxima a la que puede ir la plataforma.

Durante un plan de navegación, las direcciones que debe tomar la plataforma van cambiando. El cambio de dirección se realiza al alcanzarse una pose. Por lo tanto, debemos conocer cuando la plataforma ha llegado a una pose para recalcular la dirección entre la pose alcanzada y la siguiente.

Una primera aproximación puede ser calcular la distancia a la que se encuentra la plataforma a la pose que se desea alcanzar, y cuando la distancia sea menor a un cierto umbral, actualizar la nueva dirección. Sin embargo, como la distancia de los pasos que da la plataforma en cada actualización no es constante, es posible que marquemos un umbral de distancia el cual nunca se alcance porque la plataforma ha sobrepasado la pose.

Podemos ver en la figura 51 como en una sola actualización, la plataforma supera el umbral, impidiendo comprobar si se ha alcanzado la pose.

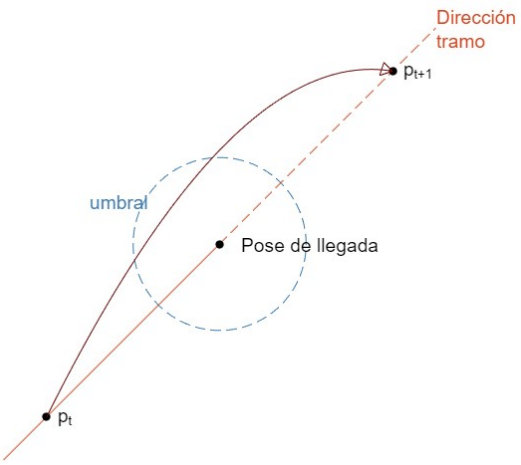


Figura 51: Umbral de pose alcanzada.

Una manera en la que podemos resolver este problema es comparando la dirección del tramo con la dirección a la que está dirigiéndose la plataforma, las cuales deberían tener el mismo sentido mientras no se haya alcanzado o superado la posición final.

Si comparamos el vector entre el punto de inicio y fin ($\hat{\mathbf{d}}$), y el vector entre la posición de la plataforma y el de fin ($\hat{\mathbf{r}}$), la dirección y sentido deberían ser los mismos. En el caso de que se haya alcanzado la pose o se haya superado, los sentidos son opuestos. Como podemos observar en la figura 52, mientras el robot está entre el punto de inicio y fin, los vectores $\hat{\mathbf{r}}$ y $\hat{\mathbf{d}}$ tienen la misma dirección y sentido. Cuando el robot supera la pose final, los sentidos son opuestos. Para comparar si dos vectores tienen el mismo sentido, utilizamos el producto escalar entre vectores.

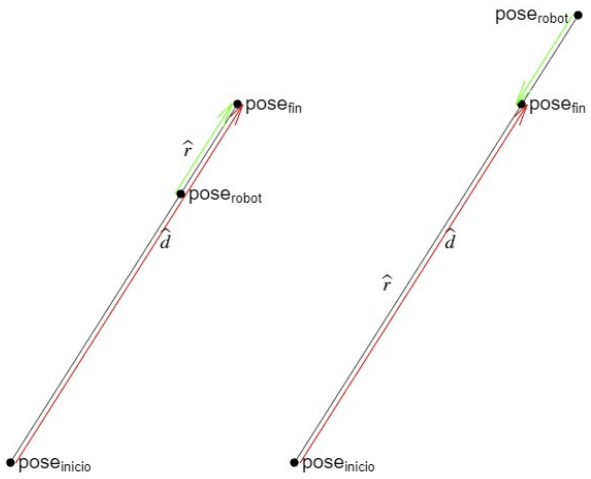


Figura 52: Comparación de vectores.

El producto escalar entre dos vectores es $\vec{a} \cdot \vec{b} = \|a\| \, \|b\| \cos(\theta)$. Cuando los vectores tienen la misma dirección, θ es igual a 0^o, por lo que el valor del coseno será 1. Como los vectores están normalizados, el valor del producto será 1. En el caso de que la dirección sea opuesta, el ángulo de θ es de 180^o, por lo que el valor del coseno, y por tanto, del producto será -1.

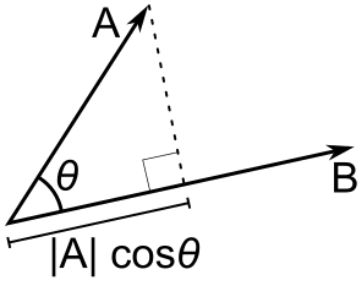


Figura 53: Producto escalar.
Extraído de `https://en.wikipedia.org/wiki/File:Dot_Product.svg`

Realizando esta comprobación cada vez que se actualiza la posición, podemos conocer si se ha sobrepasado o se ha alcanzado justo la pose (el producto será cero ya que el módulo del vector $\hat{\mathbf{r}}$ es cero. Cuando el producto es menor o igual a cero, detectamos que se alcanzó la pose, por lo que se ajusta la pose de la plataforma a la pose exacta que debía de alcanzar y se recalcula la nueva dirección.

Para ajustar la rotación sobre el eje z de la plataforma al indicado en la pose de llegada, se aplica el algoritmo de control proporcional [49]. Al iniciar un tramo de ruta, la plataforma se encuentra rotado en un ángulo, el cual debe modificarse progresivamente para simular la rotación hasta la de la pose final.

Utilizando una constante proporcional K_p y el error entre el ángulo esperado θ_s y el que tiene la plataforma θ_p, se actualiza en cada paso del *timer* el ángulo.

La actualización se realiza de la forma:

$$\theta(t+1)_p = \theta(t)_p + K_p * min\{error, 360^o - error\} \tag{2}$$

donde $K_p \in (0,1)$ y $error = |\theta(t)_p - \theta(t)_s|$

Como estamos calculando el error de una rotación, existen dos ángulos entre θ_s y θ_p, de la cual debemos tomar el mínimo.

Para comprobar que se ha alcanzado la rotación, se comprueba que el error sea mínimo a una tolerancia.

La plataforma robótica, durante el modo edición puede colocarse en cualquier posición que desee el usuario. Por este motivo, el robot, antes de ejecutarse debe colocarse en la posición de inicio. El movimiento en esta trayectoria, aunque no es representada en la interfaz porque no pertenece a la ruta trazada, es también simulada. Por lo tanto, existe un tramo inicial en el que la plataforma se prepara para iniciar el plan trazado. Para simular este comportamiento, se incluye esta trayectoria como si fuera una más de las que tiene que simular, marcando como pose de inicio la pose que tiene el robot en un momento dado.

Aunque realmente la ejecución no es un bucle sino que se utiliza un *timer*, en la figura 54 podemos visualizar un resumen del comportamiento de la simulación al completo. Al inicio se almacena la pose de la plataforma y se consultan las poses de la ruta actualmente activa. Posteriormente se elige la pose a alcanzar, y se realiza el bucle donde se actualiza la pose, hasta que se alcanza. Cuando no existen más poses por alcanzar, se vuelve a posicionar la plataforma en la pose que se almacenó al inicio, y se finaliza la simulación.

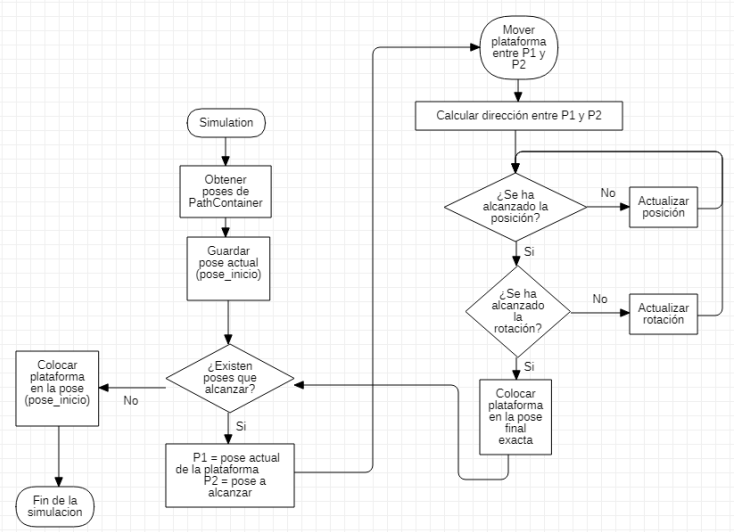


Figura 54: Flujo de la simulación.

Operaciones adicionales

Como operaciones adicionales tenemos el pausado y continuación de la ejecución, cancelación y el cambio de velocidad.

El operador `SimulatorOperator` posee un *flag* que nos permite desactivar momentáneamente la actualización de la pose de la plataforma. Con esto, podemos simular, al pulsarse el botón de pausa, que el robot se ha parado. Cuando se vuelve a pulsar el botón de *play*, se desactiva el *flag*, reanudándose la actualización de poses. Esta operación se encuentra implementada en el operador `PauseResumeSimulation`.

La cancelación de planes se encuentra implementada en el operador `SimulatorOperator`↩. Como hemos visto, los operadores modales reaccionan a cualquier evento que se produce en la aplicación. Comprobando que tipo de evento se ha lanzado, podemos asignar la tecla *escape* a la cancelación de una ejecución de un plan de navegación. Al pulsarse *escape*, se vuelve a posicionar la plataforma en la posición en la que estaba antes de iniciar la simulación. Finalmente, se elimina el *timer* y se cierra el operador.

En cuanto al cambio de velocidad, al igual que en la comunicación con la plataforma, tenemos un pulsador que abre una ventana emergente con un *slider*, con el que se establece una velocidad entre 0 % y el 100 % de la velocidad máxima. Este es el porcentaje que se utiliza en la ecuación de actualización de la pose.

9.10. Utilidades adicionales

Muchas de las funcionalidades implementadas en la aplicación necesitan de ciertas modificaciones o funciones adicionales para conseguir una mayor fiabilidad y aportar más datos en la interfaz de cara a facilitar la experiencia al usuario.

Protección y eliminación de objetos

Existen muchos objetos que son añadidos a la escena de *Blender* los cuales no pueden ser eliminados de la manera en la que está orientado el operador original de borrado.

Originalmente, el operador *delete*, solamente elimina los objetos seleccionados, sin tener en cuenta ninguna relación de parentesco. Hemos visto en apartados anteriores que los objetos pueden estructurarse en una jerarquía en forma de árbol, donde la posición y dimensiones de un objeto son relativas a un objeto padre. Sin embargo, cuando seleccionamos un objeto para eliminarlo, no se tiene en cuenta las relaciones, por lo que si eliminamos un objeto con hijos, estos no se eliminarán.

Para los objetivos de esta aplicación y teniendo en cuenta que para crear ciertos objetos utilizamos relaciones de *parenting*, es necesario modificar el operador de borrado de objetos para que pueda aplicarse opcionalmente el borrado

de un objeto y sus dependientes. Además, elimina todo material o *mesh* relacionado.

Existen en la aplicación ciertos objetos que no deberían ser eliminados de forma manual, como los *robots*, que poseen una clase relacionada, o los puntos de ruta. Para evitar que no puedan ser eliminados, se ha introducido a los objetos una propiedad booleana llamada `protected`. Cuando un objeto tiene el valor de `protected` a verdadero, el operador de borrado no actuará, informando al usuario que ese objeto no puede eliminarse. Por defecto, cualquier objeto creado no estará protegido, es obligación del programador indicar cuando un objeto lo está.

Listing 18: Propiedad *protected*.

```
1        bpy.types.Object.protected = BoolProperty(name = '↩
             protected', default = False)
```

Sobrescribir un operador en *Blender* es tan sencillo como registrar uno nuevo con el mismo `bl_idname`. En este caso, debemos crear un operador cuyo id sea `object.delete`. El nuevo operador de borrado está implementado en `DeleteOverrideOperator`. No es necesario crear ningún panel ni atajo para utilizar este operador, pues al tener el mismo `bl_idname` los atajos y menús existentes donde se utilizaba este operador nos sirven para el nuestro. Para eliminar un objeto seleccionado se debe pulsar la tecla *X*.

`DeleteOverrideOperator` se encarga de, teniendo al menos un objeto seleccionado, eliminar al mismo y todos sus hijos.

Para obtener los hijos de un objeto utilizamos la propiedad `children`. Sin embargo, esta propiedad solamente nos devuelve los hijos inmediatos, no todo el árbol. Para obtenerlos todos, se realiza un recorrido el amplitud de todo el árbol, consultando los hijos de cada objeto recuperado en `children` (Figura 19).

Listing 19: Obtener hijos de un objeto.

```
1        def get_children(parent):
2            not_visited = [parent]
3            children = []
4            while len(not_visited) > 0:
5                current_node = not_visited.pop(0)
6                children.extend(current_node.children)
7                not_visited.extend(current_node.children)
8            return children
```

La eliminación de un objeto tiene dos fases: la separación de su unión con la escena y el borrado de los datos. Ambas operaciones se realizan utilizando la lista de objetos `bpy.data.objects`. El método `remove` permite eliminar el objeto, si este existe. Agregando el parámetro `do_unlink=True` se rompe su relación con la escena. En el método `drop` (Figura 20), se elimina, además el

material y *mesh* de un objeto dado.

Listing 20: Método *drop*.

```
    def drop(obj):
        obj_name = obj.name
        mesh = obj.data
        material = obj.active_material

        # Eliminamos objeto
        if obj_name in bpy.data.objects:
            bpy.data.objects.remove(bpy.data.objects[↩
                obj_name], do_unlink=True)

        # Eliminamos el mesh que lo forma
        if mesh is not None:
            mesh_name = mesh.name
        if mesh_name in bpy.data.meshes:
            bpy.data.meshes.remove(bpy.data.meshes[↩
                mesh_name], do_unlink=True)
        if mesh_name in bpy.data.lights:
            bpy.data.lights.remove(bpy.data.lights[↩
                mesh_name], do_unlink=True)

        # Eliminamos su material
        if material is not None:
            material_name = material.name
        if material_name in bpy.data.materials:
            bpy.data.materials.remove(bpy.data.materials[↩
                material_name], do_unlink=True)
        bpy.ops.object.select_all(action='DESELECT')
```

Para eliminar los objetos seleccionados en la escena (dentro de `bpy.↩context.selected_objects`), se obtienen y eliminan todos los hijos. Posteriormente, se elimina el padre. Si tratamos de eliminar un objeto seleccionando tanto el padre como el hijo, es posible que se trate de eliminar un objeto ya eliminado, resultando en una excepción. Por ello, comprobamos que el objeto que vamos a borrar no esté entre los objetos seleccionados. Si es así se ignora, pues igualmente se eliminará en una iteración posterior, cuando se elija como objeto raíz de la búsqueda.

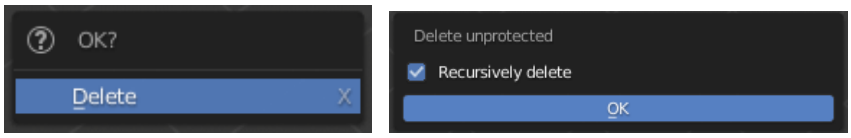
Listing 21: Método eliminar objetos.

```
        def delete(self, context, delete_children=False):
            protected_obj = []
            to_delete = [obj for obj in context.↩
                selected_objects] # map deleted objects : ↩
                avoid an invalid object error when the ↩
                child of a selected object is also selected

            for obj in to_delete:
```

```
6                   if not obj.protected:
7                       if delete_children:
8                           children = get_children(obj)
9                           for child in children:
10                              if child in to_delete: # ↩
                                    indicates selected child ↩
                                    was deleted
11                                  to_delete.remove(child)
12                              drop(child)
13                      drop(obj)
14                  else :
15                      protected_obj.append(obj.name)
16
17              if len(protected_obj) == 1:
18                  self.report({'ERROR'}, (protected_obj[0] +'↩
                        is protected'))
19              elif len(protected_obj) > 1:
20                  self.report({'ERROR'}, (str(protected_obj)↩
                        [1:-1] +' are protected'))
```

Opcionalmente, si el usuario lo indica, se puede desactivar el borrado recursivo, mediante el botón de check que se muestra en la pantalla. Para modificar el panel de confirmación original que tiene *Blender*, se sobrescribió también el método *draw*, mostrando una casilla de selección.



(a) Eliminación original (b) Eliminación recursiva

Figura 55: Ventanas de diálogo de eliminación.

Etiquetas informativas

En una interfaz de usuario impresindible que sea accesible al usuario información acerca de los datos de la aplicación. En esta aplicación existen ciertos datos relacionados con objetos y otros elementos del escenario que no son accesibles de una forma intuitiva. Para hacer disponible esta información se decidió añadir etiquetas informativas a los objetos que lo necesitaran.

Dentro de la API básica de *Blender* no existe un modo sencillo de crear texto dentro de la escena, en una posición concreta. Un modo podría ser utilizando el módulo *blf* [50], sin embargo, este módulo solo permite dibujar texto en la vista HUD (*Head-Up Display*) de la interfaz gráfica.

Para implementar esta funcionalidad, se ha utilizado el *addon MeasureIt* [7]. Específicamente, se ha extraído del código original el método `draw_text`↩ , con el que se puede dibujar texto en la pantalla utilizando *OpenGL* (Figura 22).

Listing 22: Método *draw_text* en addon MeasureIt.

```
1
2         def draw_text(context, name, text, loc, color, hint_space, font, font_align, font_rotation):
3             """
4             draw_text
5             Autor: Antonio Vazquez
6
7             - context: blender context
8             - text: str
9             display text
10            - loc: Vector
11            3D location
12            - color: Vector
13            rgb color (0, 1)
14            - hint_space: int
15
16
17            """
18            if context.area.type == 'VIEW_3D':
19                bpy.ops.object.empty_add(type='PLAIN_AXES')
20                myempty = bpy.data.objects[bpy.context.active_object.name]
21                myempty.location = loc
22                myempty.empty_display_size = 0.01
23                myempty.name = name
24
25                scene = context.scene
26                mainobject = myempty
27                if 'MeasureGenerator' not in mainobject:
28                    mainobject.MeasureGenerator.add()
29                mp = mainobject.MeasureGenerator[0]
30
31                for cont in range(len(mp.measureit_segments)-1, mp.measureit_num):
32                    mp.measureit_segments.add()
33
34                ms = mp.measureit_segments[mp.measureit_num]
35                ms.gltype = 10
36                ms.glpointa = 0
37                ms.glpointb = 0
38                ms.glcolor = color # Vector(0,0,0,0)
39                ms.glspace = hint_space # (-100, 100) def=0.1
40
41                ms.gltxt = text # str
42                ms.glfont_size = font # int
43                ms.glfont_align = font_align # 'L', 'C', 'R' // EnumProperty
44                ms.glfont_rotat = font_rotation # 0-360
45
46                mp.measureit_num += 1
47
```

```
48                  context.area.tag_redraw()
49                  return myempty.name
50              return None
```

Este método crea un objeto *empty* que contiene el texto que hayamos solicitado. Utilizar este método tiene la ventaja de que el texto generado puede ser visualizado desde cualquier punto de vista de la cámara, facilitando la lectura.

Para poder visualizar las notas, debe activarse el módulo *OpenGL*. Para ello, se ha incluido un panel donde activar y ocultar las notas (Figura 88).

El botón de *play* permite alternar entre mostrar u ocultar todas las notas existentes, activando y desactivando *OpenGl*. Para ello, se llama al operador de *measureit* `measureit.runopengl`.

Teniendo *opengl* ejecutándose, se puede ocultar parcialmente las notas. Con el pulsador del fantasma, si lo tenemos deseleccionado, podemos solamente mostrar las notas de los objetos que seleccionemos. Para ello, se cambia el valor de la propiedad booleana de *measureit* llamada `measureit_gl_ghost`.

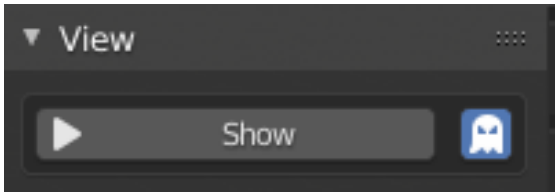


Figura 56: Panel activar notas.

Atajos de teclado

Los atajos de teclado son un método para facilitar la realización de tareas muy repititivas. En *Blender* de incluye un sistema que permite asignar ciertos atajos de teclado u acciones con el ratón a operadores registrados.

El usuario puede asignar atajos de teclado accediendo al panel de preferencias (Edit > Preferences > *KeyMap*). En este menú puede buscarse un operador y asignarle un atajo o, añadir nuevos.

La *API* de *Python* ofrece una forma de asignar por defecto al inicio del programa ciertos atajos a operadores implementados. Para ello, utilizamos el manejador de ventanas (window manager), que ofrece un mapa con el que relacionar los `bl_idname` de los operadores con atajos. Hay que tener en cuenta que muchos atajos de teclado ya se encuentran reservados, por lo que antes de utilizar uno, debemos comprobar que no esté asignado a ningún operador. Para comprobarlo, podemos intentar utilizarlo en la aplicación para observar si se produce algún cambio.

Podemos ver en la figura 23 como asignar un atajo de teclado a un operador. Al eliminar el registro del operador, también debemos eliminar el atajo asignado.

Listing 23: Asignación de atajo a operador.

```
1         keymaps = []
2
3         def autoregister():
4             ...
5
6             wm = bpy.context.window_manager
7             kc = wm.keyconfigs.addon
8             if kc:
9                 km = kc.keymaps.new(name='3D View', ↩
                      space_type='VIEW_3D')
10
11                kmi = km.keymap_items.new("wm.call_menu_pie↩
                      ", type='Q', value='PRESS', ctrl=True, ↩
                      shift=True).properties.name = ↩
                      PathEditorMenu.bl_idname
12                keymaps.append((km, kmi))
13
14        def autounregister():
15            ...
16
17            for km, kmi in keymaps:
18                km.keymap_items.remove(kmi)
19            keymaps.clear()
```

Los atajos de teclado asignados en las funcionalidades de este *addon* son:

- Mostrar anotaciones : Ctrl + Shift + N
- Ocultar márgenes de seguridad : Ctrl + Alt + A
- Ocultar techos : Ctrl + Alt + C
- Seleccionar plataforma robótica: Ctrl + Alt + S
- Cambiar de modo: Ctrl + Shift + M
- Apagar renderizado de la plataforma: Ctrl + Shift + R
- Arrancar, pausar y continuar plan de navegación: Ctrl + Shift + P
- Cancelar plan de navegación: Ctrl + Shift + C
- Editor de rutas:
  - Añadir pose: Ctrl + Shift + A
  - Deshacer: Ctrl + Shift + U
  - Mover cursor a última pose marcada: Ctrl + Shift + M
  - Abrir menú de secciones: Ctrl + Shift + Q

10. Pruebas y resultados

Durante el desarrollo de la aplicación, es necesario realizar pruebas con las que comprobar que ciertas partes críticas del código funcionan correctamente. Las partes que más en detalle se han testeado son: el módulo de chequeo de colisiones y el módulo de comunicación. Las demás partes de la aplicación, como la creación de escenarios o la edición de rutas también se probaron de forma manual, comprobando que en los posibles escenarios de uso, los resultados fueran los esperados. Durante el desarrollo del *addon*, también se siguieron ciertas reglas de diseño de interfaces —aunque se depende del diseño de interfaz de *Blender* y de la configuración final que elija el usuario en su espacio de trabajo—.

10.1. Pruebas en el chequeo de colisiones

El módulo de chequeo de colisiones, en concreto, el método `check_overlap` en el fichero `overlap_check.py` fue probado de forma separada al código del *addon*, realizando pruebas a cada método dependiente por separado.

Listing 24: Función *check_overlap*

```
1  def check_overlap(bm0, bm1):
2      """
3      Check if two bmesh collides
4      """
5      # check overlap
6      # check face_collision
7      if is_overlapping(bm0, bm1):
8          return True
9      if face_overlap(bm0, bm1):
10         return True
11 return False
```

En una primera prueba, solamente se probó el método `is_overlapping`, que comprueba a través del método `overlap` de BVHTree [37] si dos objetos se solapan.

El test consiste en generar dos cubos y alternar entre todas las posibles alternativas, cambiando su posición para comprobar tanto casos donde debería detectarse una colisión como en los que no.

En concreto, se comprobaron siguientes casos:

- Dos cubos A y B de dimensiones (2,2,2) en la posición (1,1,1). Debería detectar una colisión.
- Mover el cubo A hasta (1, 2, 2). Debería detectar una colisión.
- Mover el cubo A hasta (1, 1, 2). Debería detectar una colisión.
- Mover el cubo A hasta (1, 1, 3). Debería detectar una colisión.

- Cambiar las dimensiones de A a (1, 1, 1) y trasladarlo a hasta (1, 1, 2). Debería detectar una colisión.
- Cambiar la posición de A hasta (2,1,2). Debería detectar una colisión.
- Cambiar la posición de A hasta (12,1,2). No debería detectar una colisión.

Tras realizar esta prueba, se comprobó que en los tres primeros casos la detección falla. Esto es debido a que este método no detecta cuando dos caras se solapan o cuando dos vértices o aristas son iguales, o cuando un `mesh` se encuentra dentro de otro.

Para solucionarlo, se implementó la detección de solapamiento de caras, explicado en el apartado de desarrollo 9.6.2.
La pruebas para la detección de solapamiento de caras se realizó método a método y globalmente.

El método `point_in_segment` se testeó en `point_in_segment_test`↩ . Este método recibe dos puntos que definen un segmento y el punto que se desea comprobar si está contenido. Las pruebas que se realizaron son:

- Un punto fuera del segmento y de la recta infinita que lo contiene.
- Un punto fuera del segmento y dentro de la recta infinita que lo contiene.
- Un punto dentro del segmento.
- Un punto igual a uno de los vértices que definen al segmento.
- La reflexión sobre el plano XY de un punto no está contenida en el segmento.

La función `point_inside_infinite_plane` detecta si un punto está en un plano (infinito), a partir de un objeto `Face` y un punto. Este es probado en `point_inside_infinite_plane_test`.

Utilizando la función de un plano conocido, crear un objeto *plane* en *Blender*, para así poder extraer el objeto `Face` con el cual realizar la prueba.

Utilizando la función del plano, se crean puntos que pertenecen al plano. Como sabemos que el objeto `Face` coincide con el plano, estamos seguros que estos puntos pertenecen al plano infinito donde está contenido la cara generada. Posteriormente, con el objeto `Face` se comprueba si estos pertenecen al plano infinito con la función `point_inside_infinite_plane`.

Finalmente, los puntos generados son trasladados una distancia aleatoria distinta de 0 en el eje z. Con esto aseguramos que ningún punto pertenece al plano. Posteriormente, se comprueba que esto es cierto.

A diferencia del anterior método, `point_inside_finite_plane_test`↩ comprueba si un punto está dentro de una cara, utilizando un objeto `Face` y un punto. Para realizar la prueba, al igual que antes, se generan puntos a partir de la función del plano. Para poder conocer fácilmente que un punto pertenece

al plano, se ha utilizado el plano $y = 2$.

La cara está formada por los vértices $(-1{,}0, -1{,}0, 0{,}0), (1{,}0, -1{,}0, 0{,}0)$, $(-1{,}0, 1{,}0, 0{,}0)$ y $(1{,}0, 1{,}0, 0{,}0)$. Por lo tanto, para conocer si un punto generado con la función del plano se encuentra dentro o fuera de la cara, basta con saber si el cumple $1 <= x <= 3 \wedge 1 <= z <= 3 \wedge y = 2$. Posteriormente, los puntos se trasladan aleatoriamente en el eje Y una distancia distinta de 0, y se vuelve a testear la función.

Con `are_coplanar_test` se prueba la función `are_coplanar`, capaz de detectar si dos caras pertenecen al mismo plano. Como parámetros recibe el objeto `Face` de las caras.

La prueba se realiza realizando rotaciones y traslaciones a los planos. Las comprobaciones que se realizaron fueron:

- Dos planos A y B en (0,0,0)
- Trasladar A hasta (1,1,1), comprobar que es falso; y posteriormente trasladar B y comprobar que es verdadero.
- Rotar A 45^o en X, 90^o en Y, y -40^o en Z. Se comprueba que no son coplanares, y posteriormente se rota B.
- Rotar A y B 45^o en X. Ambos son perpendiculares al plano XY. Si se desplaza B en el eje Z 10 metros, los planos deberían seguir siendo coplanares.

La función `infinite_plane_contains_line_test` se encarga de testear `infinite_plane_contains_line`, que comprueba si una recta está contenida en un plano. El plano está definido por la cara que contiene, y la recta por dos puntos que forman un segmento. Las pruebas realizadas para este método son:

- Cuando el segmento está dentro de la cara (dos puntos dentro de la cara), debería ser verdadero.
- Cuando el segmento está parcialmente contenido en la cara (un único punto en la cara), debería ser verdadero.
- Cuando el segmento no está contenido en la cara pero sí en el plano (dos puntos fuera de la cara pero en el plano), debería ser verdadero.
- Cuando la recta intersecta con el plano en un punto dentro de la cara, debería ser falso.
- Cuando la recta intersecta con el plano en un punto fuera de la cara, debería ser falso.
- Cuando la recta es paralela al plano, debería ser falso.

A diferencia de la anterior, `finite_plane_contains_line` se encarga de comprobar si un segmento está dentro de una cara. Para ello, la recta que contiene al segmento debe estar en el plano que contiene a la cara, y al menos una

parte del segmento debe estar en el plano.

Las pruebas realizadas son las mismas que a `infinite_plane_contains_line`↩ , pero en este caso, los resultados deben ser:

- Cuando el segmento está dentro de la cara (dos puntos dentro de la cara), debería ser verdadero.
- Cuando el segmento está parcialmente contenido en la cara (un único punto en la cara), debería ser verdadero.
- Cuando el segmento no está contenido en la cara pero sí en el plano (dos puntos fuera de la cara pero en el plano), debería ser falso.
- Cuando el segmento intersecta con el plano en un punto dentro de la cara, debería ser falso.
- Cuando el segmento intersecta con el plano en un punto fuera de la cara, debería ser falso.
- Cuando el segmento es paralelo al plano, debería ser falso.

Finalmente, para probar la detección de solapamientos de caras entre objetos (en el método `face_overlap_test`), se crean dos cubos, y se mueven en las distintas posiciones posibles en las que la cara de un cubo se superpone a otra. También se prueba cuando un cubo está contenido en otro, solapándose 1, 2 y las 6 caras.

De este modo, queda comprobado que todos los casos que fallaban en el método `overlap` de `BVHTree` quedan cubiertos. Todas las pruebas realizadas en este apartado se incluyen en el Anexo I.

10.2. Pruebas en la comunicación

Las pruebas en la comunicación podemos dividirlas en distintas partes: pruebas en la serialización de paquetes, pruebas en la clase `ConnectionHandler` y pruebas generales de funcionamiento.

Pruebas en la serialización

Inicialmente, para realizar la serialización no se planteó utilizar ningún sistema de compresión de paquetes. El formato de los paquetes consistía en crear cadenas de bytes con los distintos valores, separados por un caracter especial. En un inicio se planteó de este modo para conseguir rápidamente un primer modelo de comunicación funcional con el servidor que nos permitiera realizar pruebas.

Una vez implementada una primera comunicación en la que el servidor solamente informaba de su posición, se cambió la serialización a msgpack. Gracias a la estructura de clases que se creó, simplemente se intercambiaron las clases encargadas de la serialización, sin necesidad de realizar ningún cambio en el

código de la comunicación. Posteriormente se implementaron las demás funcionalidades de la aplicación.

Para probar la serialización, se crearon paquetes de distinto tipo y se probó a serializarlos y, posteriormente a deserializarlos. Para verificar que el resultado es correcto, se comparan los valores de cada campo de paquete.

Durante la serialización y deserialización de cada paquete, se verifica que el número de tipo de paquete es correcto. Cuando se está recibiendo una cadena de bytes del servidor, se comprueba que el número de tipo corresponda con el formato de paquete recibido. En el caso contrario, cuando se desea convertir una instancia de alguna subclase de `Packet` a una cadena de bytes, se comprueba que el tipo de paquete y los valores esperados sean correctos.

Pruebas en la clase ConnectionHandler y generales de comportamiento

Al inicio, la recepción de paquetes en `ConnectionHandler` estaba separada en distintos métodos según en tipo de paquete a recibir. Mientras solo estaba implementado el cambio de modo y el seguimiento de la posición de la plataforma, es decir, solamente se recibían paquetes de seguimiento y de reconocimiento en el cambio de modo, no presentaba problemas. Sin embargo, al implementar las demás funcionalidades, al existir un hilo recibiendo paquetes de seguimiento, si intentamos recibir otros tipos de paquetes desde otras partes del código, era posible que se perdieran.

Para solucionar este problema, se creó la clase `Buffer`, que almacena todos los paquetes recibidos. Ahora, el hilo no solo recibirá paquetes de seguimiento, sino de cualquier tipo, y los insertará en `Buffer`.

Para realizar pruebas en la implementación final de `ConnectionHandler`, se probaron distintos escenarios que podrían ocurrir en la aplicación y se testeó que los paquetes recibidos son correctos.

Los distintos test que se realizaron son:

- Cambio de modo: se debe recibir una confirmación reconociendo el id del paquete enviado.
- Creación de planes: se deben recibir los reconocimientos a cada paquete recibido.
- Ejecución de un plan: se deben recibir como poses alcanzadas las enviadas. Además, las poses de seguimiento deben ser coherentes con la ruta planeada.
- Cambio de velocidad: al cambiar la velocidad, las poses de los paquetes de seguimiento recibido deben reflejar el cambio de velocidad.
- Cancelación de un plan: durante la ejecución de un plan, si se envía un paquete de cancelar, la plataforma debe dejar de moverse.

- Pausa y reanudación de un plan: al pausar un plan, la plataforma debe dejar de moverse. En el caso de que se envíe un paquete de continuar, la plataforma debe reanudar su movimiento.

Adicionalmente, se realizaron pruebas combinando las distintas operaciones. Todas estas pruebas se realizaron a través de la interfaz gráfica.

Control de errores en la comunicación

Por defecto, el módulo `socket` de *Python*, al realizarse una llamada al método de recibir datos `recvfrom`, se bloquea hasta que recibe algún tipo de dato desde el servidor que se indicó al crear el `socket`. Si no recibe nada, quedará bloqueado por siempre.

Este comportamiento es peligroso en nuestra aplicación, pues en *Blender* no podemos permitir que el hilo principal del programa se bloquee, pues congelaría por completo la aplicación. Como es un hilo separado el que realiza esta llamada no hay peligro de que se produzcan errores por esta causa. Sin embargo, tampoco debemos mantener este estado de espera para siempre, pues si el servidor no está disponible, nunca se informaría al usuario.

Para cambiar el comportamiento de bloqueo del módulo, se utiliza la orden `ConnectionHandler.client_socket.settimeout(3)`, que establece un tiempo de espera de 3 segundos. En el caso de que se pase el tiempo y no se reciba nada, se produce una excepción, que capturamos en un `try except`. Como el hilo ejecuta esta orden dentro de un bucle infinito mientras está activo, el comportamiento es el mismo que si no marcáramos un *timeout*. Entonces, para detectar si el servidor no está disponible, se cuenta el número de veces que se ha capturado la excepción de `settimeout`. Si se produce más de 10 veces, se cambia a modo editor, informando al usuario de que el servidor no está disponible.

Esta comprobación también nos sirve cuando el usuario trata de conectarse al servidor y está apagado, o indica mal la ip. En ese caso, se le informará de que no se puede cambiar a modo plataforma robótica.

10.3. Creación de escenarios y editor de rutas

Creación de escenarios y plataformas robóticas

Este módulo fue probado a través de la interfaz gráfica, recreándose los distintos escenarios de uso.

Como la creación de la mayor parte de los objetos de la escena consiste en aplicar modificaciones a objetos básicos y combinarlos, la posibilidad de que se produzca un error es mínima. Las pruebas en este módulo tienen como objetivo comprobar que los objetos producidos son los esperados, y que se realice un

control de parámetros no válidos.

En la creación de paredes, se debe comprobar que la longitud de la pared sean distintas a cero. Si es así, se debe informar al usuario. Por otra parte, debemos chequear que al crear paredes relativas a la posición del cursor 3D, siempre se sitúen a ras de suelo y que la rotación no afecte.

La creación de habitaciones, al estar basados en los operadores de creación de paredes, ya se tiene cubierto que no puedan construirse muros de longitud cero. Sin embargo, si definimos la dimensión de una habitación midiendo por su exterior (incluyendo el ancho de los muros), debemos comprobar que no ocurran solapamiento de paredes. Esto puede ocurrir si la suma de la anchura de dos paredes paralelas no debe superar la longitud de las perpendiculares.

Otra opción, sería cambiar la definición de dimensión de una habitación, siendo las medidas tomadas desde el interior. Como normalmente, la toma de medidas de una habitación es realizada desde su interior, se ha optado por esta opción. Por lo tanto, si indicamos que las dimensiones de una habitación son cero, no se creará, pues no puede existir una habitación de 0 m^2.

En cuanto a la creación de beacons, techos y obstáculos, como la operación simplemente consiste en llamar a la operación de `bpy` que crea los objetos primitivos, las pruebas fueron mínimas.

Las pruebas con la creación de robot se centraron en las funciones de selección y eliminación. De este apartado hay que destacar que, en ciertas ocasiones, la operación de eliminar plataformas robóticas cuando se aplica a demasiadas, puede que no se eliminen todas las seleccionadas, por lo que habrá que pulsar varias veces el botón eliminar.

También se comprobó en este apartado que, al eliminarse una plataforma robótica, esta dejara de estar seleccionada.

Editor de rutas

El editor de rutas fue testeado de forma independiente al módulo de comunicación, ya que fue implementado antes. En este módulo debemos probar que las poses guardadas sean correctas, que se guarden en el contenedor correcto (en la lista temporal cuando se está editando y en la definitiva cuando se guardan las poses) y que la representación en la pantalla sea lo más exacta posible al valor almacenado.

Al encontrarse el editor integrado en la interfaz, solamente podemos realizar pruebas a través de la aplicación. Para conocer el contenido de las listas `TempPathContainer` y `PathContainer`, se han creado un operador, el cual al ejecutarse, muestra en la consola del programa las poses almacenadas. Este operador puede ejecutarse en cualquier momento, utilizando el buscador de operadores (*F3* > *Operator search* > Nombre del operador > *Enter*). Para vi-

sualizar el contenido, abrimos la consola de *Blender* en *Window* > *Toggle System Console*; o, desde la consola que hayamos lanzado *Blender*, si es el caso.

Listing 25: Log de *PathContainer*.

```
1  class PathEditorLog(bpy.types.Operator):
2      bl_idname = "screen.patheditor_log"
3      bl_label = "PathEditor Log"
4      bl_description = "Logger"
5
6      @classmethod
7      def poll(cls, context):
8          return True
9
10     def execute(self, context):
11
12         log = "PathEditor\n"
13         log += "\tPathContainer : " + str(len(pc.↩
               PathContainer())) + "\n"
14         for p in pc.PathContainer().poses:
15         log += "\t\t" + str(p) + "\n"
16         log += "\tTempPathContainer : " + str(len(pc.↩
               TempPathContainer())) + "\n"
17         self.report({'INFO'}, log)
18         return {'FINISHED'}
```

Con este operador, se comprobó que los datos son almacenados tal y como son trazados en el editor.

Al inicio, se utilizaba el objeto *empty single arrow* para representar la rotación en una pose marcada. Sin embargo, como al crearlo, la flecha queda apuntando hacia Z positivo, debemos rotarla 90^o sobre el eje Y, para que su dirección sea el eje X. Al realizar esta transformación, una rotación sobre el eje Z, resultaría en una rotación en el objeto sobre el eje X. Para aplicar el editor de rutas en el plano XY, cambiar rotaciones sobre el eje Z por el eje X no influiría. Sin embargo, si aplicamos el editor de rutas a poses en el espacio tridimensional, como está implementado (aunque se ha bloqueado el marcado de poses en el eje Z y rotaciones sobre X e Y), nos obligaría a llevar un control de todas las transformaciones aplicadas a los objetos.

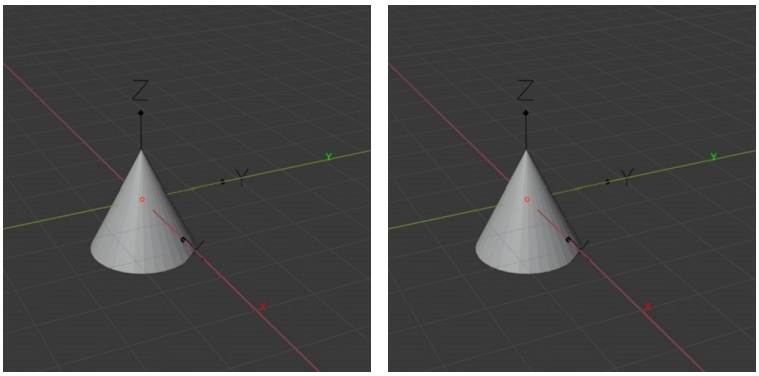


Figura 57: Rotación sobre eje Y.

Este problema podemos solucionarlo aplicando la transformación al objeto, mediante el comando 26. Sin embargo, este comando no puede aplicarse a objetos *empty*, por lo que se decidió cambiar el objeto a un *armature*. Los *armatures* son utilizados para crear esqueletos de personajes y objetos a animar. Como tiene una forma similar a una flecha, podemos utilizarla con este fin.

Listing 26: Aplicar transformaciones.

```
1 bpy.ops.object.transform_apply(location=True, rotation=True↩
      , scale=True)
```

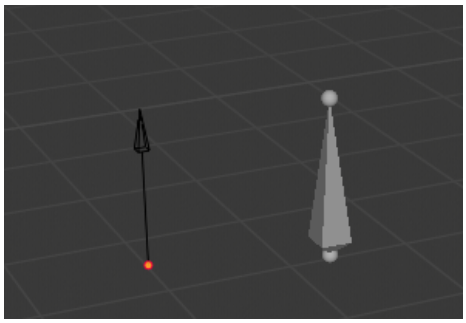


Figura 58: *Single empty arrow* y *armature*.

Otro problema encontrado en la representación de la ruta fue en la creación de la línea que une dos poses. Para crear esta línea, se utiliza el mismo método que se aplicó en la creación de muros. En este caso, creamos una arista con dos vértices: uno en la pose de salida y otro en la pose de llegada. Sin embargo, si tratamos de crear una arista con dos vértices en el mismo punto, la arista se convierte en un único vértice —como ocurre cada vez que añadimos una pose, al crearse la nueva trayectoria—. Por lo tanto, si intentamos mover el segundo vértice mientras se mueve el cursor geométrico, no existirá una línea que une a la pose de inicio con el cursor.

Para solucionar este problema, al crear una arista, se desplaza una de los vértices una distancia mínima que permita al motor de *Blender* diferenciar a los dos vértices (una tolerancia de 0.001).

10.4. Evaluación de la interfaz gráfica

Durante el desarrollo de software, es tan importante comprobar el funcionamiento de los componentes internos como realizar evaluaciones a la interfaz gráfica, pues es la que determina en gran medida la usabilidad de la aplicación.

Aunque en este trabajo se ha dependido en gran medida de las herramientas que ofrece *Blender* y de la distribución de los elementos en su interfaz, se puede trabajar para conseguir que el *addon* cumpla con ciertas reglas, con el objetivo de mejorar la experiencia de usuario.

Para evaluar esta interfaz, se han utilizado las reglas de Schneiderman y Plaisant [51] [52], y las de Jackob Nielsen. [53] [54] Al diseño de ciertos elementos, podemos aplicar una evaluación a partir de los principios de percepción *Gestalt* [55] [56].

Evaluación con reglas Schneiderman y Plaisant

Las reglas de Schneiderman y Plaisant, conocidas comúnmente como las 8 reglas de oro, tienen como objetivo realizar una evaluación heurística de las aplicaciones.

Consistencia

Una interfaz consistente tiene un diseño organizado de modo que se siga un patrón común, en la que para usos similares que haga el usuario se deban seguir los mismos patrones.

El uso de paneles facilita al usuario encontrar rápidamente la funcionalidad que busca, pues estos se dividen en categorías según en la fase de trabajo que se encuentre el usuario. Todas las opciones relacionadas con la comunicación están completamente separadas de la creación de escenarios, pues el usuario durante la comunicación no debería necesitar crear elementos del escenario.

Además, internamente, cada pestaña se divide en subpaneles y otros contenedores, distribuyendo aún más la interfaz.

Otro aspecto en donde podemos percibir la consistencia es el uso de iconos, que tienen en toda la interfaz el mismo significado. Se trata de utilizar los que en todas las aplicaciones tienen un mismo significado, aunque la biblioteca de iconos en *Blender* es limitada.

También se trata de conseguir consistencia en el simulador de ejecución de planes, construyendo un panel exactamente igual al que se utiliza para la comunicación. Todas las acciones en la simulación y en la comunicación real se realizan del mismo modo, siendo más intuitivo para el usuario.

Usabilidad universal

La usabilidad universal está relacionada con lo accesible que es la aplicación para nuevos usuarios.

Este *addon* trata de simplificar las tareas de diseño de escenario a través de funcionalidades que permiten la creación de elementos prediseñados. Esto reduce la necesidad de aprendizaje del usuario novato, que no tiene la obligación de conocer herramientas complejas dentro de *Blender* para empezar a realizar diseños simples de construcciones.

Informativo

Es importante que exista una retroalimentación en la aplicación para las acciones que esté ejecutando el usuario.

Una forma que tenemos de mostrar datos de la aplicación es gracias a la utilización de anotaciones mediante el *addon MeasureIt*.

El progreso de ciertas tareas que se ejecutan durante un largo tiempo (como la ejecución de un plan) es visualizada en la interfaz gracias al movimiento de los elementos de la escena.

Otro proceso que requiere un tiempo largo de ejecución es la exportación de escenarios. La única utilidad que nos facilita *Blender* para mostrar el progreso de una tarea es el cursor de progreso. El cursor de progreso es una especie de contador que aparece en el cursor cuando se está ejecutando una tarea larga. Podemos ver en la figura 59 su aspecto. En ocasiones puede ser demasiado pequeño y el usuario no verlo. Si en un futuro la API de *Blender* da acceso a la barra de progreso que muestra en tareas como el renderizado, sería conveniente cambiarla.

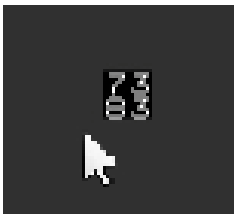


Figura 59: Cursor de progreso.

Flujo y cierre

Tanto como informar del progreso de una tarea, es conveniente informar cuando una tarea ha acabado.

Una forma que tenemos de mostrar mensajes al usuario es a través del método `report` que poseen los operadores. Utilizando este método podemos mostrar tanto mensajes de error como informativos cuando alguna acción iniciada por el usuario ha finalizado.

Prevenir errores

Además de los errores que puedan ocurrir por cuestiones internas del programa, es posible que el usuario realice ciertas acciones erróneas, que provoquen un resultado que no esperaba.

Para evitarlo, en la gran parte de propiedades se establece un formato de entrada. Para ciertos valores numéricos como la velocidad de la plataforma se establece un rango el cuál no puede superarse. En otros casos, también se comprueba el formato de algunas cadenas de caracteres: como la ip de la plataforma, que debe ser correcta; o el nombre de fichero donde se va a exportar un escenario, que debe contener caracteres alfanuméricos y acabar en `.blend`.

Deshacer

Aún si prevenimos errores es posible que el usuario realice una acción que no deseaba. Es esencial que exista una función de deshacer. Así también se posibilita probar distintas opciones.

Hay que tener en cuenta que hay ciertos casos en los que no puede utilizarse la opción deshacer común de la aplicación de *Blender*, pues este solo afecta a los objetos, no eliminando datos de clases que se han creado (como las poses almacenadas en `PathContainer`). Para facilitar al usuario acciones de deshacer, se han facilitado herramientas de eliminación de objetos y de deshacer: como el botón de deshacer y eliminar pose en el editor de rutas, o la opción de eliminar plataformas robóticas.

Sentir el control

El usuario de una aplicación debe sentir que tiene el control del sistema, todas sus acciones provocan algún resultado. Es el programa el que se adapta al usuario y no al revés.

La estructura de operadores que ofrece *Blender* facilita considerablemente cumplir esta regla, pues los operadores no pueden iniciarse si no es con la acción

de un usuario o, al ejecutarse otro operador.

Otro modo en el que el usuario puede tener control de la aplicación es con opciones que faciliten pausar o cancelar acciones iniciadas. Estas pueden variar desde las opciones de control de la plataforma hasta en funciones más simples como las incluidas en el editor de rutas. El usuario puede elegir cuando parar la ejecución de un plan, o incluso puede pausar la ejecución para cambiar la ruta, todo mientras se mantiene la comunicación con la plataforma.

Memoria a corto plazo

La capacidad de memorización del usuario es limitada por lo que facilitar opciones para minimizar la necesidad de memorizar mejorará la experiencia.

Siempre que ha sido posible se han seguido estándares tanto de distribución como de terminología, utilizados en la mayoría de aplicaciones.

Evaluación con reglas de Jackob Nielsen

La reglas de *Jackob Nieslen* comparten muchas definiciones con las de *Schneiderman y Plaisant*. Sin embargo, cabe destacar algunos aspectos.

Visibilidad del estado del sistema

La visibilidad del sistema está relacionada en gran parte con la visualización del progreso de tareas. Aquí también podemos incluir el control de errores que se realiza en la comunicación con el servidor, detectando e informando al usuario cuando se ha perdido la comunicación.

Coincidencia entre el sistema y el mundo real

La coincidencia con el mundo real se puede observar desde dos perspectivas: en la creación e interacción con los escenarios, y en la terminología que se aplica en la interfaz.

El diseño de escenarios trata de, utilizando elementos simples, llegar a representar lo más fielmente posible a la realidad los escenarios del mundo real. Con esto se consigue que el usuario pueda visualizar e interactuar con el escenario imaginando los resultados de la ejecución de un plan.

Control y libertad del usuario

Aunque se faciliten herramientas para la creación de escenarios y para el control de la plataforma robótica, no impide al usuario utilizar las herramientas

propias de *Blender* y otros *addons* disponibles para diseñar o mejorar el escenario.

Consistencia y estándares

La necesidad de aprendizaje en una aplicación puede disminuir en gran medida si se siguen estándares utilizados en otros programas.

Por este motivo, utilizando las posibilidades que nos aporta *Blender*, se ha diseñado la interfaz siguiendo una terminología y utilizando gráficos comunes tanto en *Blender* como en otras aplicaciones similares.

Prevención de errores

La prevención de errores, como se explicó anteriormente, impide al usuario cometer fallos que provocan resultados inesperados.

Reconocimiento más que recordar

La visibilidad de las opciones en la aplicación y el uso de estándares, gráficos e instrucciones accesibles, reducen la necesidad de memorización del usuario, mejorando su experiencia.

Flexibilidad y eficiencia de uso

Con el objetivo de tener más accesible ciertas funciones que pueden ser utilizadas múltiples veces, se han introducido algunos atajos de teclado, para así evitar la necesidad de recorrer todos los paneles de opciones, además de reducir el número de pasos a dar para llegar a una opción. Aunque no sea visible siempre, el panel `menu pie` que se despliega al pulsar *Ctrl + Shift + Q* durante la creación de rutas, también aumenta la accesibilidad de las funciones de agregar y deshacer. El usuario no debe desplazarse al panel izquierdo cada vez que quiere marcar una pose.

Estética y diseño mínimo

Las aplicaciones solamente deben mostrar los datos que son importantes en un momento dado.

Gracias al operador de ocultar anotaciones que ofrece el *addon Measureit*, es posible minimizar la información mostrada en pantalla, permitiendo al usuario seleccionar solamente la que desee ver.

Aunque con las herramientas que ofrece *Blender* es posible desarrollar escenarios realistas, para los objetivos de esta aplicación, es innecesario que el usuario malgaste tiempo en realizar diseños de objetos complejos. El diseño mínimo con ortoedros de los elementos de la escena es suficiente para representar la ejecución de planes de una plataforma robótica.

Reconocer y recuperar errores

Los mensajes de error deben ser informativos, mostrando la causa de los errores que se producen. Además, se deben ofrecer alternativas al usuario para deshacer fallos.

Excepto en errores internos que puedan ocurrir en la aplicación, todos los mensajes de error muestran, en la medida de lo posible la causa.

Documentación y ayuda en línea

Aunque las interfaces de usuario deben ser autoexplicativas, en ocasiones es necesario aportar ayuda al usuario.

Por este motivo, se ha hecho accesible tanto en esta memoria como en el repositorio un manual de ayuda para la instalación y uso del *addon*. El manual es accesible también a través de *Blender* en el enlace de documentación (Ver figura 60).

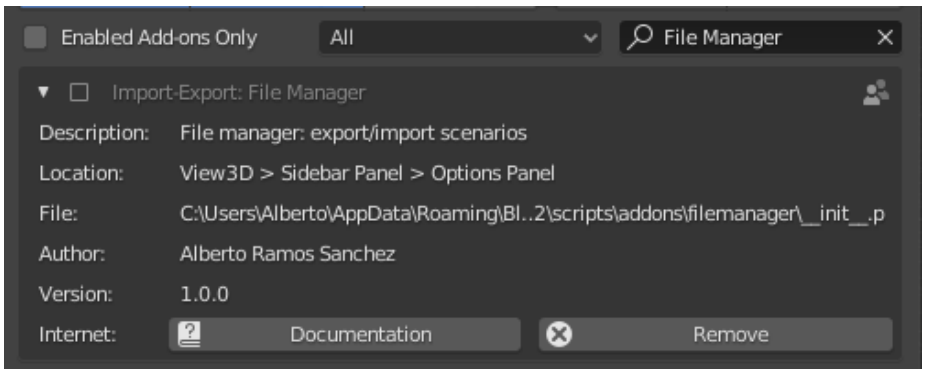


Figura 60: Acceso a documentación.

Principios *Gestalt*

Los principios *Gestalt* son un conjunto de teorías de la percepción visual humana, que afirman que la información percibida por los sentidos pasa por un proceso de reestructuración, originando formas imaginarias (una gestalt), que no son captadas si se trata de analizar.

Ley de la proximidad

Los elementos que se encuentran más cercanos son percibidos como una unidad.

En la aplicación, gracias a la utilización de paneles, subpaneles y contenedores, podemos organizar las distintas funcionalidades por su categoría.

Ley de la semejanza

Sin utilizar la proximidad, es posible relacionar elementos como una unidad utilizando valores similares de color o forma.

Como la API de Blender no permite modificar libremente la interfaz gráfica, la única opción para relacionar funcionalidades es posicionarlas próximamente, utilizando separadores y contenedores. Esta regla es utilizada también en la creación de elementos del escenario: los objetos de la misma categoría, aunque pueden variar ligeramente, tienen características comunes que permiten al usuario conocer su categoría sin necesidad de explicaciones adicionales.

Ley del cierre

Ciertos contenedores permiten crear líneas como separadores. De este modo, podemos separar funcionalidades similares pero que no son utilizables al mismo tiempo. Por ejemplo, las funciones de comunicación y simulación se encuentran en un mismo panel, pero como no podemos utilizarlas al mismo tiempo —para evitar confusiones— las separamos con un contenedor.

Ley de continuidad

Los elementos organizados siguiendo alguna alineación son percibidos por el usuario como un conjunto.

Esta ley se cumple en las ventanas de diálogo donde se muestran formularios para crear objetos, las cuales siguen una distribución común para los mismos tipos de datos: los distintos parámetros se organizan en distintas filas y cuando un parámetro necesita varios valores se distribuyen en una misma fila (como las coordenadas de una posición). Además, la distribución de ciertos elementos indica su significado, como el campo numérico que indica la anchura de cada pared de una habitación. Al colocarlos en forma de cruz, podemos saber a que pared se refiere sin necesidad de leer la etiqueta.

Ley del contraste

Para poder distinguir los objetos de una interfaz es necesario que exista cierto contraste entre el fondo y la figura.

Este problema lo podemos encontrar en la creación de escenarios. Las líneas que representan una trayectoria son aristas, las cuales siempre son de color negro y no hay posibilidad de añadirles un material. Como el fondo por defecto es oscuro, en ocasiones puede ser complicado distinguir la línea.

Para solucionarlo tenemos dos opciones: cambiar el tema de *Blender* a uno claro, o cambiar el color por defecto que da *Blender* a las líneas. En el manual se encuentra explicado como conseguir un mayor contraste en las líneas. A

Ley de la experiencia

Esta ley indica que, además de la percepción hay que tener en cuenta la experiencia previa del usuario. Las interfaces deben ser flexibles a los distintos usuarios y sus hábitos.

Por esto mismo se ha tratado de seguir estándares comunes en aplicaciones similares, siempre que la aplicación lo ha permitido.

Existen otras reglas Gestalt que teorizan acerca de la percepción visual humana, sin embargo, no son aplicables a esta aplicación o al campo de diseño de interfaces.

11. Conclusión y trabajo futuro

11.1. Conclusión

Tras un análisis de todos los objetivos propuestos podemos concluir que, aunque no se haya podido probar la aplicación en un entorno real, se han conseguido cumplir todos los requisitos.

En resumen, actualmente el addon permite:

- Construir escenarios virtuales a partir de objetos prediseñados
- Serializar y cargar escenarios fabricados
- Generar rutas
- Controlar una plataforma robótica comunicándose con un servidor para: enviar rutas y controlar su ejecución.
- Simular la ejecución de un plan de navegación

Aún así, al existir una multitud de plugins gratuitos disponibles, otras funcionalidades adicionales, como la medición, también están cubiertas.

Realizar desde cero un trabajo con una aplicación real ha sido una experiencia provechosa, desde la fase de estudio de las herramientas a utilizar, hasta el diseño y desarrollo del software. Este proyecto ha permitido poner en práctica los conocimientos adquiridos durante la carrera y obtener otros que en su mayoría no fueron enseñados durante la carrera o no fueron puesto en práctica (como ciertos conceptos de computación 3D o la implementación desde cero de una aplicación que actúe como cliente en una comunicación con un servidor).

También hay que destacar la experiencia adquirida en las herramientas utilizadas como *Blender* y la comunicación con el módulo de *UDP* de *Python*; y, aunque no se llegara a utilizar, el descubrimiento de otras herramientas como ROS, que podría ser útil conocerla y aprender a utilizarla en un futuro.

11.2. Trabajo futuro

Claramente, el siguiente paso que debe darse en este proyecto es probar que la aplicación opera correctamente con la plataforma robótica real, y estudiar si la configuración actual es válida para una comunicación con otro equipo dentro de una red (y no en un mismo equipo, como funciona actualmente).

Una vez se haya conseguido un correcto funcionamiento con *RoboMap*, se pueden plantear algunas mejoras o funciones adicionales que serían muy útiles.

Actualmente, las rutas creadas solamente se almacenan durante el tiempo en el que se ejecuta la aplicación. Podría ser útil tener una funcionalidad con la que serializar rutas en ficheros, y poder leerlos y cargarlos en el escenario. También sería conveniente poseer una lista de rutas, en la que seleccionar una en concreto.

En la implementación actual, `PathContainer` almacena la ruta cargada para ejecutar. La implementación de esta lista de rutas necesitaría de una clase que represente a una ruta, almacenando todas las poses, y la cual pueda serializarse. También se necesitaría en la interfaz una lista de rutas, con la que seleccionar que ruta cargar en `PathContainer`. Podría implementarse de forma similar a como se implementó en este trabajo la lista de robots, con la que podemos seleccionarlos o eliminarlos.

Hay que tener en cuenta que `PathContainer` almacena instancias `Action`↩ , los cuales ademas de representar una trayectoria, también la crea en la escena. Por lo tanto, en la lista de rutas no podemos almacenar instancias de `Action`, pues generaría una escena con todas las rutas dibujadas en pantalla.

Si se va a realizar esta implementación, también sería recomendable no realizar el almacenamiento junto con la serialización de escenarios. Aunque una ruta depende de un escenario, el escenario no debería depender de una única ruta.

Por el momento, el editor de rutas solamente realiza trayectorias rectilíneas, si bien la plataforma robótica es capaz de moverse en cualquier dirección, por lo que limitamos su comportamiento. En un futuro podría modificar este editor para que permita crear curvas. Una opción para dibujar curvas podría ser utilizar los tipos de objeto *curve*, que permite la creación de distintas clases de curva en modo editor (el modo editor clásico de *Blender* [57], no confundir con el concepto modo editor que se plantea en el panel de control de comunicación).

Las anotaciones que muestran información de los objetos depende del *addon* *Measureit* que, si bien, es muy útil, convendría eliminar esa dependencia, buscando algún modo de acceder a *openGL*, como lo hace el método `draw_text`. Con esto, podrían personalizarse las etiquetas y, además, añadirla a objetos cuyos parámetros puedan cambiar durante la edición del escenario, como los *beacons*. Las etiquetas de *Measureit* son estáticas, una vez creadas no puede cambiarse el texto.

La validación de rutas se aplica, comparando colisiones entre el área que ocupa la plataforma robótica durante una trayectoria y los objetos de la escena. El tiempo de la validación puede aumentar considerablemente en escenarios con muchos elementos. Una forma de mejorar el chequeo de colisiones sería aplicarlo a un número de objetos limitado, buscando colisiones desde los objetos más cercanos a los más lejanos.

La verificación de ruta es utilizada únicamente durante la edición de rutas. Sin embargo, al ser una operación costosa, no se aplica ni al iniciar un plan de navegación ni durante la ejecución de planes. Como el usuario puede modificar el escenario en cualquier momento, se debería encontrar un método para aplicar la comprobación de colisiones durante la ejecución de los planes.

La plataforma robótica se orienta en el escenario gracias a los *beacons*, por lo que el valor de su posición depende de la posición en los *beacons*. Por el momento, al no existir una comunicación con una plataforma real, pueden utilizarse las mismas coordenadas y rotaciones. Cuando pueda utilizarse este *addon* con *RoboMap*, deberá existir algún sistema para transformar las coordenadas entre el escenario virtual y el real.

Adicionalmente, podría introducirse otras modificaciones, como crear otras formas de obstáculos o permitir crear escenarios con distintas pendientes.

Referencias

[1] Marc Compte (2018, July 9th), *Posicionamiento Indoor.* Extraído de https://www.unigis.es/posicionamiento-indoor/ [Último acceso: 2020 Abril]

[2] Brian Ray (2018, August 16th), *How An Indoor Positioning System Works.* Extraído de https://www.airfinder.com/blog/indoor-positioning-system [Último acceso: 2020 Abril]

[3] *What are some applications of indoor positioning system?* Extraído de https://www.quora.com/What-are-some-applications-of-indoor-positioning-system [Último acceso: 2020 Abril]

[4] Peng Y., Niu X., Tang J., Mao D. y Qian C. (2018, Oct 12nd), *Fast Signals of Opportunity Fingerprint Database Maintenance with Autonomous Unmanned Ground Vehicle for Indoor Positioning.* Extraído de https://www.ncbi.nlm.nih.gov/pubmed/30322016 [Último acceso: 2020 Mayo]

[5] Nastac, D., Lohan, E., Iftimie, F., Arsene, O. y Cramariuc, B. (2018, June 14th-16th), *Automatic Data Acquisition with Robots for Indoor Fingerprinting In Proceedings of the 12th International Conference on Communications.* Extraído de https://www.researchgate.net/publication/327006313_Automatic_Data_Acquisition_with_Robots_for_Indoor_Fingerprinting [Último acceso: 2020 Mayo]

[6] *Blender.* Extraído de https://www.blender.org/about/ [Último acceso: 2020 Mayo]

[7] Vazquez, A., *Measureit Addon.* Extraído de https://docs.blender.org/manual/en/latest/addons/3d_view/measureit.html [Último acceso: 2020 Mayo]

[8] Žlajpah, L. (2008, February 16th), *Simulation in robotics.* Extraído de https://www.sciencedirect.com/science/article/abs/pii/S0378475408001183 [Último acceso: 2020 Mayo]

[9] *Gazebo sim.* Extraído de http://gazebosim.org/ [Último acceso: 2020 Mayo]

[10] *Robotic Operating System.* Extraído de https://www.ros.org/ [Último acceso: 2020 Mayo]

[11] *Webots.* Extraído de https://www.cyberbotics.com/ [Último acceso: 2020 Mayo]

[12] *CoppeliaSim.* Extraído de https://coppeliarobotics.com/ [Último acceso: 2020 Mayo]

[13] *Autodesk 3ds Max.* Extraído de https://www.autodesk.es/products/3ds-max/overview [Último acceso: 2020 Mayo]

[14] *Autodesk Maya.* Extraído de https://www.autodesk.es/products/maya/overview [Último acceso: 2020 Mayo]

[15] *Blender.* Extraído de https://www.blender.org/ [Último acceso: 2020 Mayo]

[16] *Blender's History.* Extraído de https://docs.blender.org/manual/en/2.82/getting_started/about/history.html [Último acceso: 2020 Mayo]

[17] Watt A. (2000), *3D Computer Graphics (3ª ed.). Addison-Wesley* ISBN 0-201-39855-9

[18] Gouraud, Henri (1971, June), *Continuous shading of curved surfaces, IEEE Transactions on Computers.* https://doi.org/10.1109/T-C.1971.223313

[19] Phong B. T. (1975, June), *Illumination for computer generated pictures.* https://doi.org/10.1145/360825.360839

[20] *Nvidia research areas.* Extraído de https://research.nvidia.com/research-area/algorithms [Último acceso: 2020 Mayo]

[21] *Cycles.* Extraído de https://docs.blender.org/manual/en/dev/render/cycles/introduction.html [Último acceso: 2020 Mayo]

[22] *Cycles renderer.* Extraído de https://www.cycles-renderer.org/ [Último acceso: 2020 Mayo]

[23] *Freestyle engine.* Extraído de https://docs.blender.org/manual/en/latest/render/freestyle/introduction.html [Último acceso: 2020 Mayo]

[24] Lampel J. (2019, March 7th), *Cycles vs. Eevee - 15 Limitations of Real Time Rendering in Blender 2.8.* Extraído de https://cgcookie.com/articles/blender-cycles-vs-eevee-15-limitations-of-real-time-rendering

[25] *EEVEE engine.* Extraído de https://docs.blender.org/manual/en/latest/render/eevee/introduction.html [Último acceso: 2020 Junio]

[26] *Workbench engine.* Extraído de https://docs.blender.org/manual/en/latest/render/workbench/introduction.html [Último acceso: 2020 Mayo]

[27] *Object parenting.* Extraído de https://docs.blender.org/manual/en/latest/scene_layout/object/editing/parent.html [Último acceso: 2020 Junio]

[28] *Blender Manual.* Extraído de https://docs.blender.org/manual/en/latest/index.html [Último acceso: 2020 Mayo]

[29] *Use an External Editor.* Extraído de https://docs.blender.org/api/current/info_tips_and_tricks.html#use-an-external-editor [Último acceso: 2020 Abril]

[30] *Operator(bpy_struct).* Extraído de https://docs.blender.org/api/current/bpy.types.Operator.html [Último acceso: 2020 Mayo]

[31] *Property Definitions.* Extraído de https://docs.blender.org/api/current/bpy.props.html [Último acceso: 2020 Mayo]

[32] *Panel(bpy_struct).* Extraído de https://docs.blender.org/api/current/bpy.types.Panel.html [Último acceso: 2020 Mayo]

[33] *Panel UILayout(bpy_struct).* Extraído de https://docs.blender.org/api/current/bpy.types.UILayout.html#bpy.types.UILayout [Último acceso: 2020 Mayo]

[34] *3D Cursor.* Extraído de https://docs.blender.org/manual/en/latest/editors/3dview/3d_cursor.html [Último acceso: 2020 Junio]

[35] *Application Handlers.* Extraído de https://docs.blender.org/api/current/bpy.app.handlers.html [Último acceso: 2020 Junio]

[36] *Depsgraph.* Extraído de https://wiki.blender.org/wiki/Source/Depsgraph [Último acceso: 2020 Junio]

[37] *BVHTree (mathutils.bvhtree).* Extraído de https://docs.blender.org/api/current/mathutils.bvhtree.html [Último acceso: 2020 Junio]

[38] *Bounding Volume Hierarchy.* Extraído de https://www.ibr.cs.tu-bs.de/projects/bvh/. [Último acceso: 2020 Junio]

[39] *BMesh module.* Extraído de https://docs.blender.org/api/current/bmesh.html#. [Último acceso: 2020 Junio]

[40] *BMesh Geometry Utilities (bmesh.geometry).* Extraído de https://docs.blender.org/api/current/bmesh.geometry.html [Último acceso: 2020 Junio]

[41] *Geometry Utilities (mathutils.geometry).* Extraído de https://docs.blender.org/api/current/mathutils.geometry.html [Último acceso: 2020 Junio]

[42] *Extrude.* Extraído de https://docs.blender.org/manual/en/latest/modeling/meshes/editing/mesh/extrude.html [Último acceso: 2020 Junio]

[43] J. Postel (1980, August 28th), *User Datagram Protocol (RFC 768).* Extraído de https://tools.ietf.org/html/rfc768 [Último acceso: 2020 Junio]

[44] *MsgPack.* Extraído de https://msgpack.org/ [Último acceso: 2020 Junio]

[45] *MsgPack PyPi.* Extraído de https://pypi.org/project/msgpack/ [Último acceso: 2020 Junio]

[46] *Strategy Method.* Extraído de https://styde.net/patron-de-diseno-strategy-introduccion/ [Último acceso: 2020 Junio]

[47] *socket — Low-level networking interface.* Extraído de https://docs.python.org/3.7/library/socket.html [Último acceso: 2020 Junio]

[48] *Modal operator.* Extraído de https://docs.blender.org/api/current/bpy.types.Operator.html#modal-execution [Último acceso: 2020 Junio]

[49] *Controlador PID.* Extraído de https://www.picuino.com/es/arduprog/control-pid.html [Último acceso: 2020 Junio]

[50] *Font Drawing (blf).* Extraído de https://docs.blender.org/api/current/blf.html [Último acceso: 2020 Junio]

[51] Shneiderman B., Plaisant C. (2009, December 15th) *Diseño de Interfaces de Usuario. ADDISON WESLEY* ISBN-13 978-8420548036

[52] Santos, A., *8 reglas de oro para un mejor diseño de interfaz.* Extraído de https://webdesign.tutsplus.com/es/articles/8-golden-rules-for-better-interface-design--cms-30886 [Último acceso: 2020 Junio]

[53] Nielsen J., Molich R. (1990, March), *Heuristic evaluation of user interfaces.* https://doi.org/10.1145/97243.97281

[54] Nielsen J. (1994, April 24th), *10 Usability Heuristics for User Interface Design.* Extraído de https://www.nngroup.com/articles/ten-usability-heuristics/ [Último acceso: 2020 Junio]

[55] Soegaard, M., *Gestalt principles of form perception.* Extraído de https://www.interaction-design.org/literature/book/the-glossary-of-human-computer-interaction/gestalt-principles-of-form-perception [Último acceso: 2020 Junio]

[56] Blanco, J. (2016, May 17th), *Leyes Gestalt en el diseño de interfaces digitales.* Extraído de https://blog.interactius.com/leyes-gestalt-en-el-dise%C3%B1o-de-interfaces-digitales-168e82c1475f [Último acceso: 2020 Junio]

[57] *Object Modes.* Extraído de https://docs.blender.org/manual/en/latest/editors/3dview/modes.html [Último acceso: 2020 Junio]

Anexo

Apéndice A Manual de usuario

Nota previa

Los *Add-ons* se encuentran desarrollados sobre la versión 2.82 de Blender. Debido los cambios realizados en la aplicación y la API de *Python*, estos no funcionan en versiones inferiores a la 2.80.

A.1 Instalación

A.1.1 Aplicación Blender

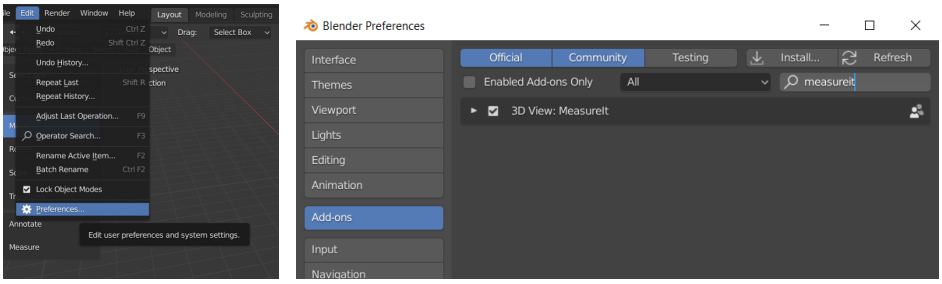
La última versión de Blender se encuentra en formato portable o instalable en la web oficial: `https://www.blender.org/download/`

A.1.2 Instalación de Addons

Una vez tengamos la aplicación Blender instalada en nuestro equipo, es necesario activar e instalar ciertos *plugins* (conocidos como *Addons* en *Blender*) para la utilización de nuestra aplicación.

En primer lugar, debemos activar el *Addon Measureit* que incluye *Blender* pero que está desactivado por defecto.

Para activarlo nos dirigimos a la pestaña preferencias : Edit > Preferences. Posteriormente, en la pestaña *add-ons*, buscamos *Measureit* y comprobamos que el *checklist* esté seleccionado.



(a) Preferencias (b) Activar *Measureit*

Figura 61: Activar *Measureit*.

Measureit es necesario para crear las anotaciones que se muestran sobre los objetos de la aplicación. Además, contiene ciertas utilidades con las que tomar medidas.

Finalmente, debemos instalar los *Addons* propios de nuestra aplicación. En este caso, al no estar distribuidos con la propia aplicación *Blender*, debemos

instalarlos a partir de los 4 ficheros comprimidos en zip: *archibuilder.zip*, *robot-control.zip*, *utilities.zip*, *filemanager.zip*.

La instalación se realiza desde el panel de *Addons* de la ventana de preferencias.

Clicando en *Install...*, seleccionamos el fichero *zip* que deseamos instalar. Una vez instalado, nos debería aparecer automáticamente en el término de búsqueda. Si no es así, lo buscamos y lo activamos, al igual que lo hicimos con *Measureit*.

Estos pasos los repetimos con los 4 ficheros zip que disponemos.

Una vez completados estos pasos, guardamos las preferencias en la opción *Save preferences*.

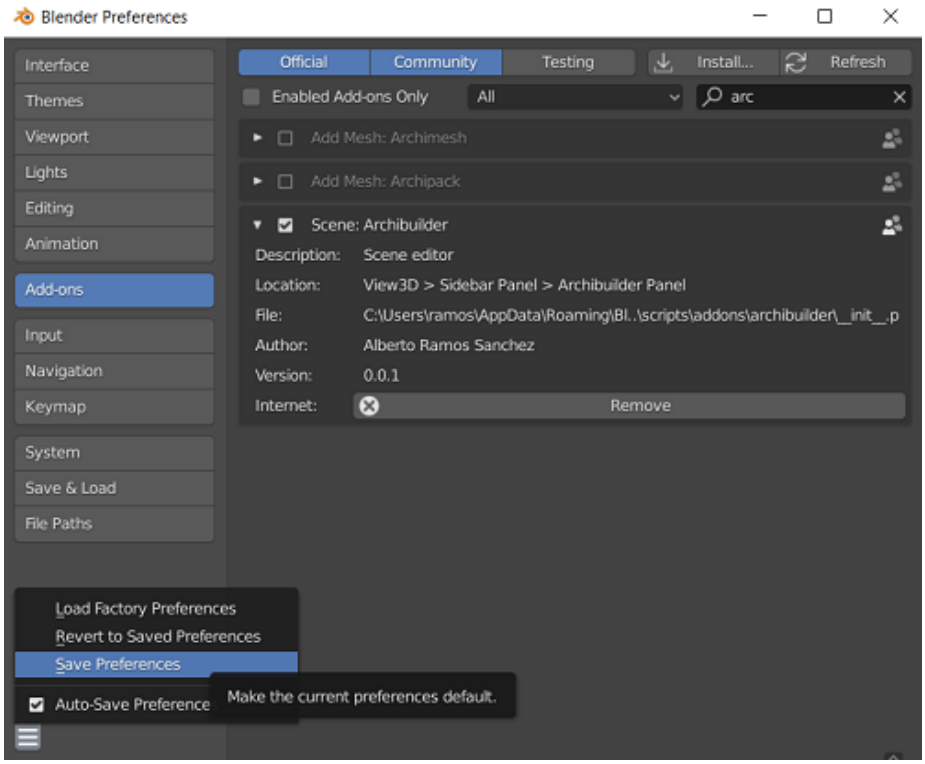


Figura 62: Guardar preferencias.

A.1.3 Instalación de módulos adicionales

El *addon* de comunicación utiliza el módulo *msgpack* de *Python* para la serialización de escenarios. Este módulo no está incluido en la instalación de *Python* distribuida junto a *Blender*, por lo que hay que instalarlo con `pip`.

Para instalarlo, seguimos los siguientes pasos:

1. Nos dirigimos a la carpeta donde tenemos instalado *Blender*.

2. Abrimos una terminal en **ruta-instalación-blender/blender-2.82a-windows64/blender-2.82a-windows64/2.82/python/bin/**

3. Ejecutamos los siguientes comandos:

   ./python.exe -m pip install –upgrade pip

   ./python.exe -m pip install msgpack

A.1.4 Desinstalación de Addons

En el caso que necesitemos desinstalar un *addon*, primero debemos desactivarlo desmarcando el *checklist* en el menú de preferencias. Esto es importante sobre todo con los addons creados para la aplicación, pues con esta acción se eliminan todas las clases registradas en la aplicación. Una vez hecho, la eliminamos con el botón *remove*.

A.1.5 Recomendaciones

A.1.6 Aumentar visibilidad de los elementos del escenario

Por defecto, el tema de *Blender* es oscuro. Esto es un inconveniente para visualizar ciertos elementos, pues el contraste de colores es muy bajo.

Debido a que no puede cambiarse el color de vértices desde la *API* de *Blender*, la única forma que tenemos de solucionar este problema es cambiar la configuración de *Blender*.

Cambio de tema de *Blender*

La primera opción que tenemos es cambiar el color del tema de *Blender*. Para ello, vamos a *Edit* > *Preferences* > *Themes*. En la pestaña superior (Figura 63), elegimos un tema más claro, para así conseguir mayor contraste.

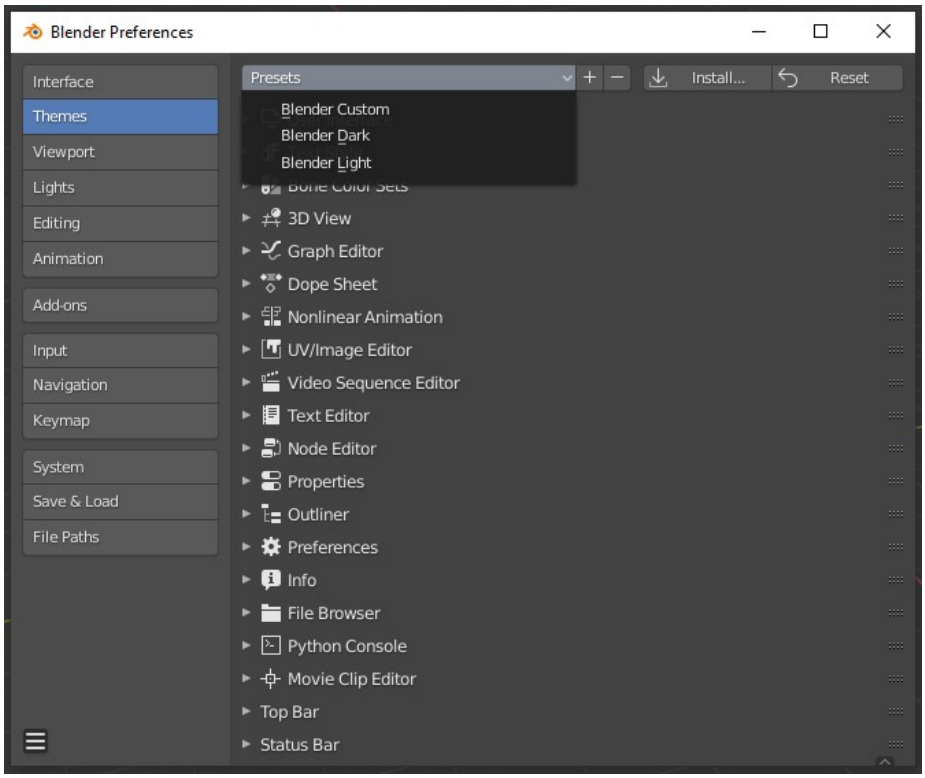


Figura 63: Cambiar tema.

Cambio de color de color de aristas

Otra opción sería cambiar únicamente el color de las aristas. Para ello, en el mismo menú de antes, nos dirigimos a *3D View* y seleccionamos un color para el campo *Wire*. De este modo, las aristas que representan las rutas tendrán el color seleccionado.

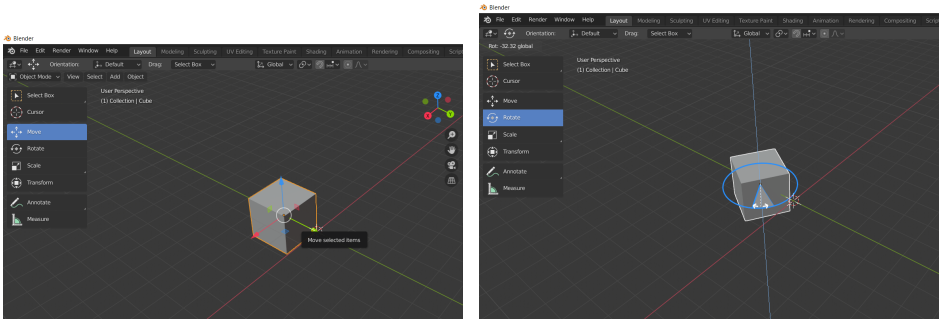
A.2 Introducción a la aplicación *Blender*

Hay varias operaciones que debemos conocer para utilizar la aplicación. Las más básicas, relacionadas con el manejo de objetos 3D son mover y rotar objetos, eliminarlos, etc.

Para mover un objeto, lo seleccionamos clicando sobre el objeto, o arrastrando el cuadro de selección sobre él; y seleccionamos la herramienta de mover. Con las flechas que aparecen sobre el objeto podemos moverlo en los 3 ejes. La rotación se realiza del mismo modo, pero con la herramienta de rotación.

Otra forma de cambiar la posición es, teniendo seleccionado un objeto, pulsando la tecla G y posteriormente el nombre del eje sobre el que deseamos movernos (X, Y, Z). Para la rotación podemos pulsar la tecla R y posteriormente el eje sobre el que se desea rotar. Desplazando el ratón, cambiamos el valor.

Para confirmar la nueva posición, clicamos el botón izquierdo del ratón. Para cancelar, pulsamos el derecho(el objeto volverá a la posición de origen).



(a) Mover objeto (b) Rotar objeto

Figura 64: Posicionamiento de objetos.

El valor de la posición y rotación también podemos modificarlo introduciendo el valor. Para ello nos dirigimos a la pestaña *Item* del *panel N* (lo abrimos pulsando la tecla N).

En ocasiones podremos observar que alguno de los ejes de movimiento y/o rotación aparecen bloqueados. Aún, por cuestiones de la implementación de *Blender*, es posible cambiarlos desde la pestaña *item*, como se destaca en el propio manual de *Blender* :

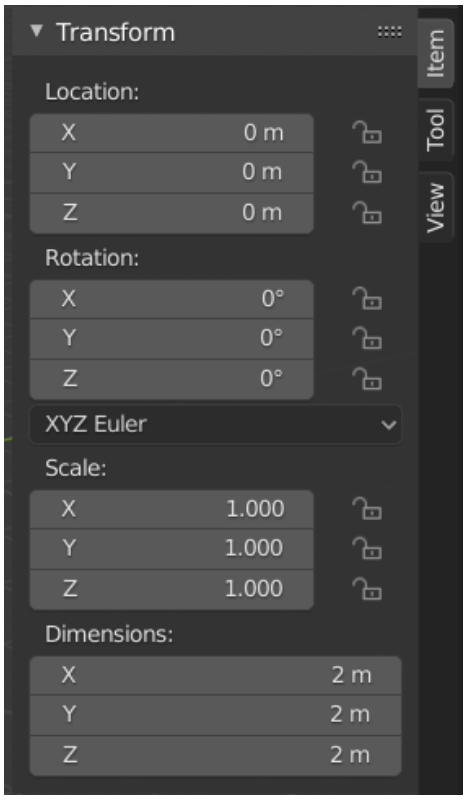
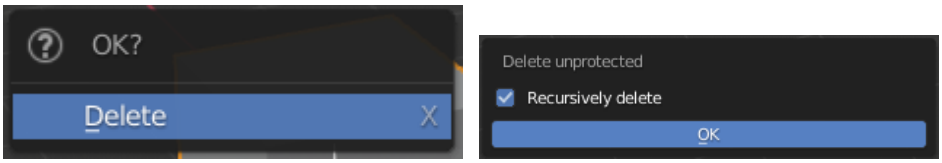


Figura 65: Panel item.

Si queremos eliminar un objeto, teniéndolo seleccionado, pulsamos la tecla *'X'* y, posteriormente, la tecla *'Enter'*.

Vemos que en la ventana de confirmación de eliminación tenemos un *check recursively delete* (Figura 66b). Este se refiere a la modificación realizada sobre la operación de eliminación de objetos propia de *Blender*. Esta modificación permite, cuando se selecciona el botón *check*, eliminar todos los objetos hijos de un objeto padre. Los objetos en *Blender* están organizados en una jerarquía de árbol donde pueden existir objetos padres de otros. Lo más seguro para nuestra aplicación es dejar seleccionada esta opción, como viene por defecto.



(a) Eliminar original (b) Eliminar modificado

Figura 66: Eliminar objetos.

El último elemento básico que debemos conocer es el cursor 3D (Figura 67).

Este permite indicar al usuario una posición en el espacio 3D y es utilizado como origen para la creación de cualquier objeto en *Blender*.

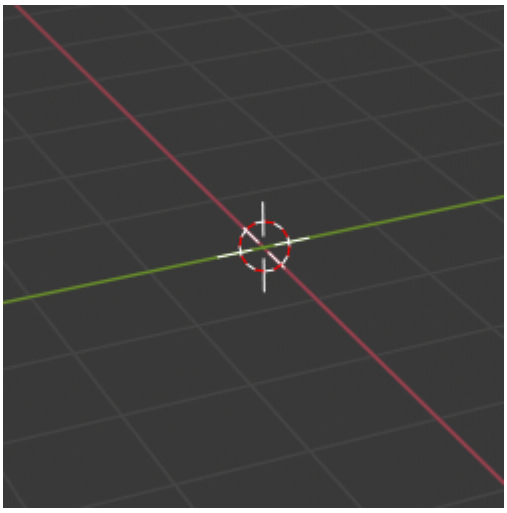
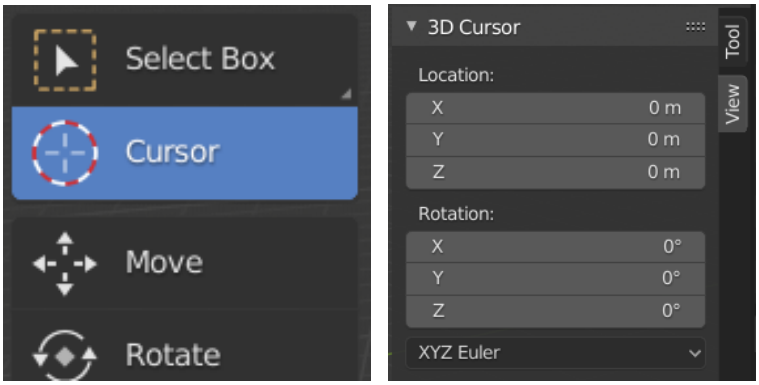


Figura 67: 3D Cursor.

Su posición podemos cambiarla tanto utilizando la herramienta *Cursor* como con el panel *3D Cursor* (Figura 68). Si deseamos volver a colocarlo en el centro, pulsamos las teclas *Shift + C*.



(a) Herramienta cursor 3D (b) Posición del cursor 3D

Figura 68: Posición del cursor 3D.

A.3 Módulo de creación de escenarios

El *Addon Archibuilder* posee todas las funcionalidades que nos permiten modelar escenarios. Estas se encuentran en la pestaña *Archibuilder* (Figura 69).

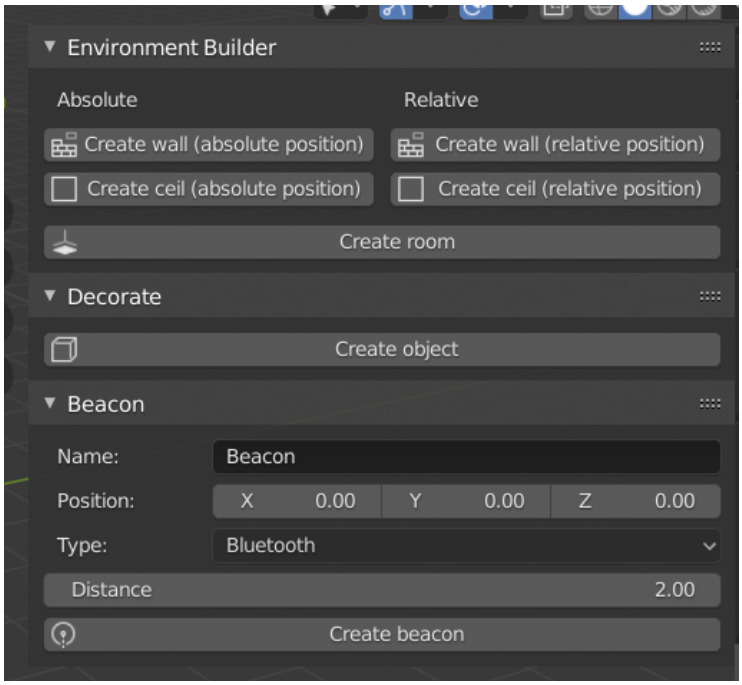
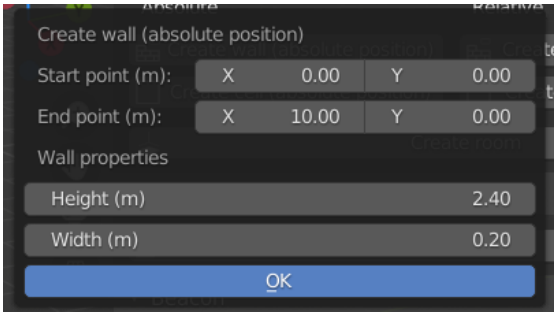
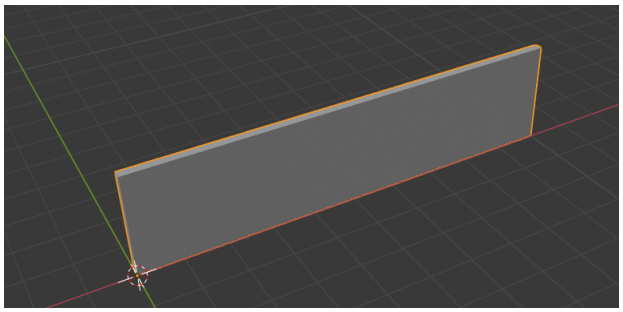


Figura 69: Archibuilder panel.

La creación de muros se realiza indicando dos posiciones, una de inicio y otra de fin. En línea recta entre esas dos posiciones se creará un muro, de la altura y anchura indicada.



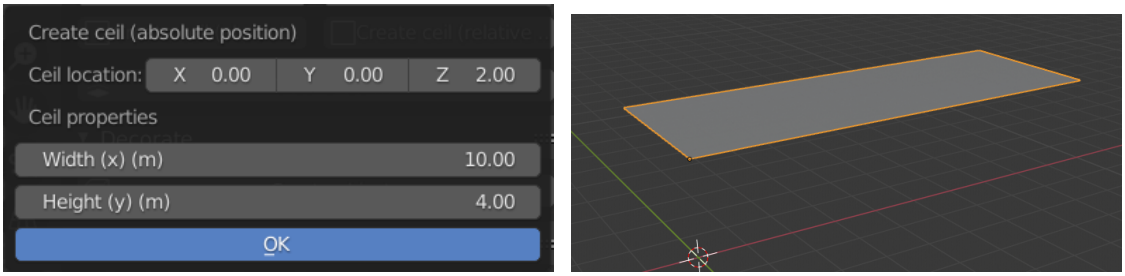
(a) Panel de crear muros



(b) Muro creado

Figura 70: Creación de muros.

En cuanto a la creación de techos, el usuario debe indicar la posición y dimensiones del plano que lo representará.

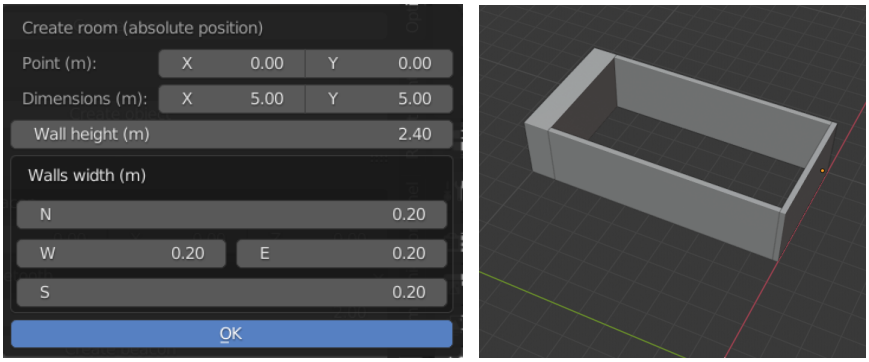


(a) Panel de crear techo (b) Techo creado

Figura 71: Creación de techos.

Tanto para la creación de muros como de techos existen dos opciones: elegir posiciones relativas o absolutas. Si indicamos que las posiciones son absolutas, se tomará como origen el centro del mundo. En el caso de que elijamos posiciones relativas, se tomará como origen al cursor 3D.

Para mayor facilidad, también se ha considerado la opción de crear habitaciones. Las habitaciones son de forma rectangular, tienen como punto de origen el vértice izquierdo inferior; y sus dimensiones son elegidas por el usuario. Además, puede elegir el grosor de cada una de las paredes: norte, sur, este y oeste. (El norte se encuentra en la dirección positiva del eje y).



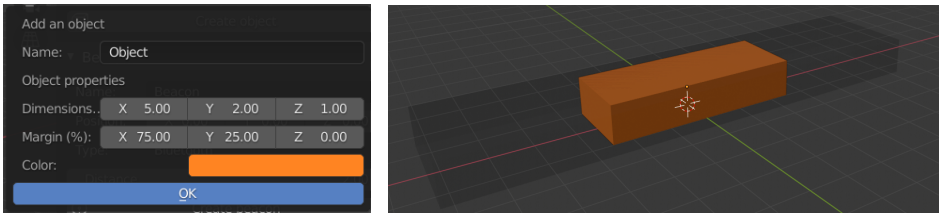
(a) Panel de creación de habitaciones (b) Habitación creada

Figura 72: Creación de habitaciones.

Una vez tenemos el escenario, también podemos añadir objetos dentro de él. Para ello, podemos indicar tanto sus dimensiones en sus 3 ejes como el del área de seguridad. El tamaño del área de seguridad es proporcional al tamaño real del objeto. Es decir, el tamaño del área de seguridad es calculado según el porcentaje elegido por el usuario. El área de seguridad permite que la plataforma robótica no se acerque completamente a los objetos, manteniendo una distancia de seguridad.

El obstáculo es creado en el centro del escenario, por lo que es responsabili-

dad del usuario situarlo en la posición que desee.



(a) Panel de creación de obstáculos (b) Obstáculo creado

Figura 73: Creación de obstáculos.

Finalmente, el último elemento del módulo *Archibuilder* es la creación de *beacons*. Por el momento existen dos tipos: *bluetooth* y ultrasónico. La creación de beacons se realiza utilizando el panel disponible (Figura 74), donde, además de indicar un nombre identificativo y su posición, podemos seleccionar el tipo y las características específicas de cada uno de los tipos de beacon.

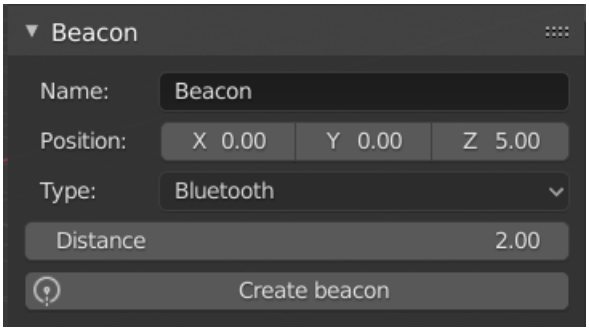
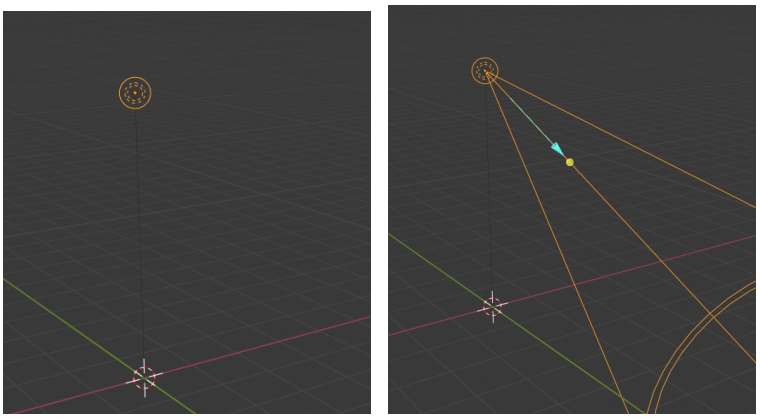


Figura 74: Panel de *beacons*.



(a) Beacon bluetooth (b) Beacon ultrasónico

Figura 75: Creación de *beacons*.

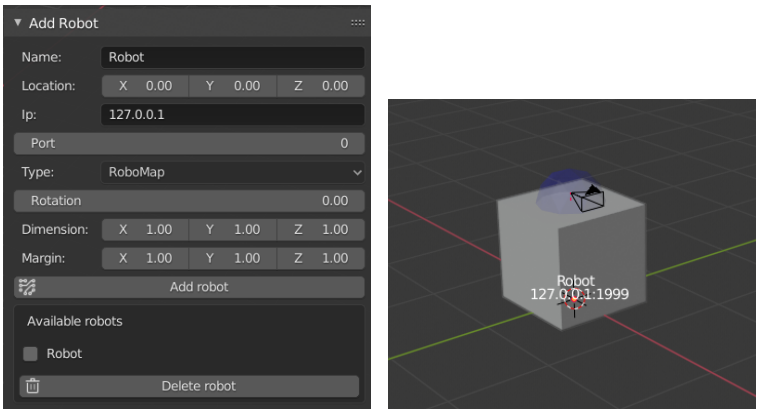
A.4 Módulo de creación de rutas y comunicación

A.4.1 Crear plataformas robóticas

El *Addon RobotControl* posee las funcionalidades relacionadas con la creación de plataformas robóticas, rutas y la comunicación con el servidor de la plataforma robótica real.

El primer paso que debemos hacer para tener una comunicación con la plataforma robótica real es crear un objeto que representa al robot, que posee la ip del servidor con el que se comunicará la aplicación, y será utilizado para visualizar en el escenario modelado las acciones de la plataforma robótica.

En el panel de creación de la plataforma robótica (Figura 76a) tenemos el formulario para añadir el robot. En el se encuentran tanto las características comunes a todos los robots como las propias de cada tipo de plataforma. Por el momento solamente existe un tipo de plataforma (*Robomap*).



(a) Panel de añadir robot. (b) Robot añadido.

Figura 76: Creación de robots.

Para eliminar un robot no podemos utilizar la acción de eliminar propia de *Blender*, sino que debemos hacerlo con el panel disponible para ello (Figura 77).

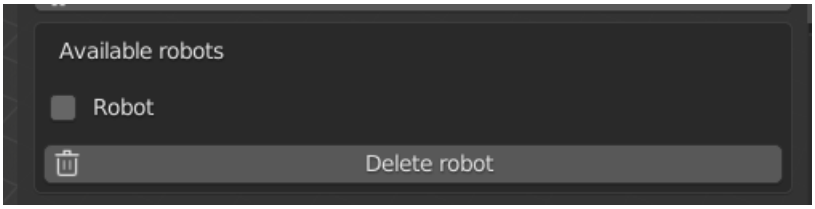


Figura 77: Eliminar robot.

En cualquier momento, puede cambiarse la vista de la plataforma a primera persona. Para ello podemos pulsar el cero en el teclado numérico, o activar la opción *toggle camera view* (Figura 78). Si hay varias plataformas, se activará la cámara del robot seleccionado.

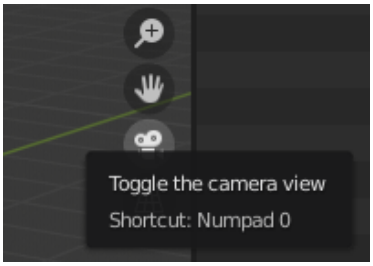


Figura 78: Vista primera persona.

A.4.2 Creación de rutas

Gracias al editor de rutas se pueden diseñar planes de navegación, que pueden ser utilizados de dos formas: enviarlas a la plataforma robótica para que la

ejecute, o simularla.

Para empezar a diseñar un plan de navegación debemos haber creado, al menos, una plataforma robótica y haberla seleccionado. Para seleccionar una plataforma robótica, nos dirigimos al botón *Select robot*, en el panel *robotcontrol*.

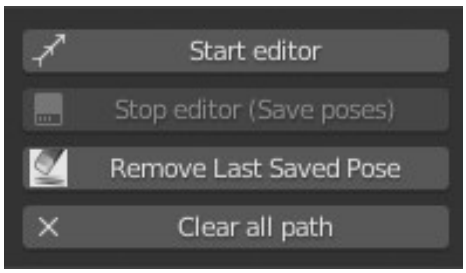


Figura 79: Arrancar editor.

Para arrancar el editor de rutas, pulsamos en la tecla *Start Editor* (Figura 79). Una vez aparezca el cursor geométrico (cono de color cian) lo movemos y rotamos para marcar las distinta poses. El cursor geométrico aparecerá en la posición que se encuentre la plataforma robótica.

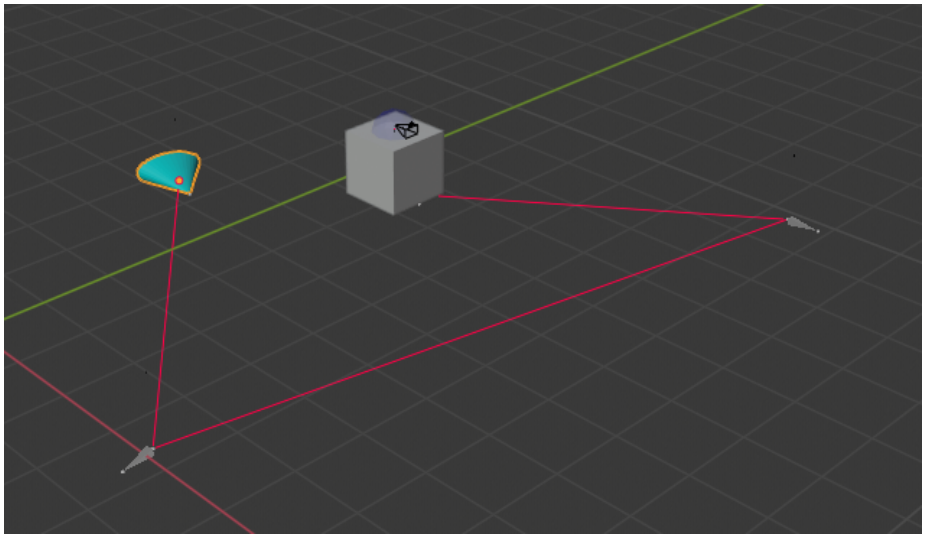
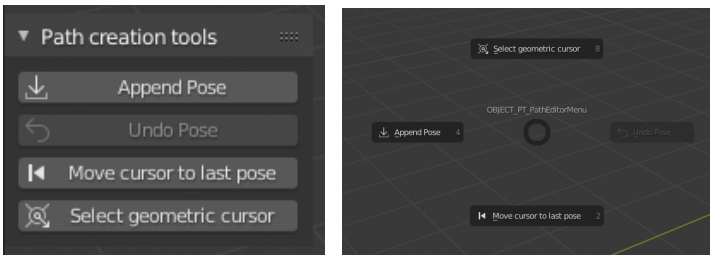


Figura 80: Crear ruta.

Para marcar una pose en la ruta, se utiliza la opción *append pose*. Esta operación aparece en el panel de la izquierda, o en el menú de disco que se abre con el atajo *Ctrl* + *Shift* + *Q* (Figura 81).



(a) *Path creation tools panel.* (b) *Path creation tools pie menu.*

Figura 81: *Path creation tools.*

También se han incluido otras herramientas en este panel: como deshacer la última pose marcada, mover el cursor a la última pose guardada o una opción para seleccionar el objeto cursor geométrico.

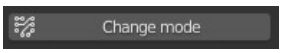
Para cerrar el editor de rutas, se utiliza el botón *stop poses listener*. En ese momento, todas las poses que se hayan creado se almacenarán como poses de la ruta cargada en la escena.

Adicionalmente, mientras se tiene el editor de rutas cerrado, se pueden eliminar las últimas poses marcadas con dos herramientas disponibles: *delete last saved pose* y *clear all path* (Figura 79).

A.4.3 Panel de control

Una vez exista una plataforma robótica seleccionada, se podrá establecer la comunicación. Para comenzar la comunicación, se debe cambiar a modo plataforma robótica. El botón *change mode*, permite intercambiar entre los modos plataforma robótica o modo diseño. Para saber en que modo nos encontramos, el propio pulsador cambiará su icono según el modo actual (Figura 82). Al entrar a modo plataforma robótica, se solicitará una velocidad inicial, que podrá ser cambiada en cualquier momento con el botón *change speed* (Figura 83). El valor de la velocidad es relativo a la velocidad máxima que puede ir la plataforma robótica (0 - 100 % de la velocidad máxima).

(a) Modo diseño.



(b) Modo plataforma robótica.

Figura 82: Cambio de modo.

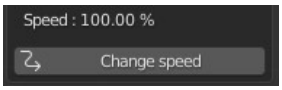


Figura 83: Cambiar velocidad.

Dentro de modo robot, la plataforma robótica se moverá a la posición en la que se encuentra realmente en el escenario real. En este modo también pueden crearse y editarse planes.

Si existe un plan creado, se podrá enviar a la plataforma robótica. Para ello, se pulsa el botón de *Play* que hay en el panel de control. Dentro del mismo panel, se encuentran opciones para pausar y cancelar el plan que se está ejecutando. En caso de que se edite un plan mientras está pausado, se cancelará el que se estaba ejecutando y se enviará el nuevo, por lo que la plataforma comenzará desde el inicio.

Durante la comunicación es posible hacer que no se actualice en la interfaz la posición de la plataforma. Para ello se dispone del botón *rendering active* (Figura 84).

(a) Renderizado activo.

(b) Renderizado inactivo.

Figura 84: Renderizado de la posición de la plataforma robótica.

A.4.4 Panel de simulación

En el modo simulado no se establece ningún tipo de comunicación con el servidor, sino que se imita el comportamiento de la plataforma desde *Blender*.

El panel de control es similar al de control de la plataforma, con la única diferencia que el botón de cancelar ha sido sustituido por el de pausa. En la simulación, para cancelar un plan, se pulsa la tecla *escape* (Figura 85).

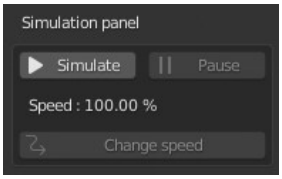


Figura 85: Panel de simulación.

A.5 Módulos adicionales

De forma adicional se han incluido algunas utilidades en la aplicación, como la serialización de escenarios y la ocultación de ciertos objetos.

La exportación de escenarios se realiza utilizando la opción que se encuentra en la pestaña *Options>Files* en el *panel N* (Figura 86). Aquí podemos elegir el lugar donde deseamos almacenar el escenario y el nombre del fichero. Es conveniente utilizar esta herramienta y no la propia de *Blender*, pues con ella conseguimos exportar solamente los elementos que realmente forman parte del escenario y no el estado completo del proyecto *blender* que se está ejecutando.

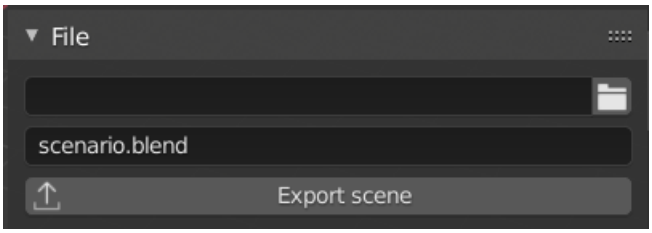


Figura 86: Exportación de escenarios.

La importación se realiza utilizando la opción Abrir *File* > *Open...* (Figura 87).

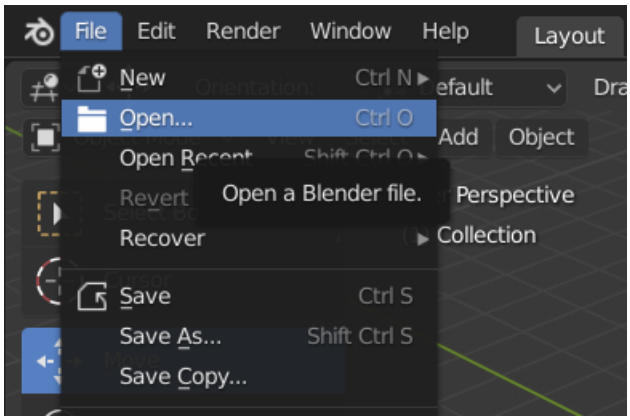


Figura 87: Abrir fichero *.blend*.

Existen varios objetos a los que se les añade información adicional a través de etiquetas de texto (por ejemplo al robot mostrando su ip o a las rutas mostrando el valor de la posición en cada punto).

Para mostrar estas anotaciones, es necesario activar *OpenGL* en la aplicación. Para esto, tenemos la acción *Show* (Figura 88), que hace uso del *Addon Measureit* para activar *OpenGL*. Además, a la derecha de esta acción tenemos

otra con la cual podemos mostrar completamente todas las anotaciones o solamente las de objetos seleccionados. Esto nos es útil cuando tenemos demasiadas anotaciones en la pantalla, como ocurre en el caso de las rutas.

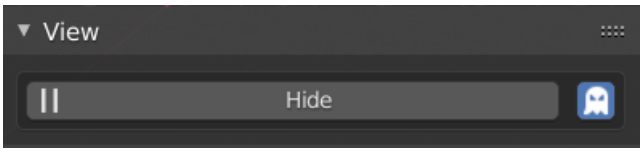


Figura 88: Mostrar notas.

Finalmente, relacionado con la visualización de los escenarios, podemos ocultar ciertos elementos como son los techos y los áreas de seguridad (Figura 89). La ocultación de los techos es útil en caso que creemos un escenario con techo y deseemos seleccionar los objetos del interior.

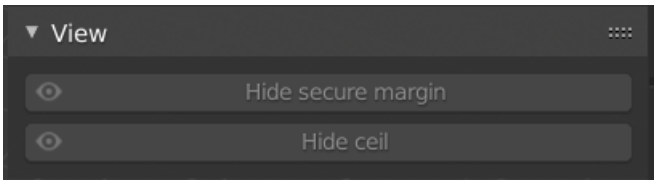


Figura 89: Ocultar techos y áreas de seguridad.

Apéndice B Función *add_obstacle*

```
1         def add_obstacle(self, context, name, size, margin,↩
              r, g, b, a):
2             bpy.ops.mesh.primitive_cube_add()
3             obstacle = bpy.context.selected_objects[0]
4             obstacle.name = name
5
6             # Cambiar tamaño del objeto
7             obstacle.dimensions = Vector((size.x, size.y, ↩
                  size.z))
8
9             # Colocar objeto a ras de suelo teniendo en ↩
                  cuenta sus dimensiones
10            inc_z_location = size.z/2
11            obstacle.location.z += inc_z_location
12
13            obstacle.object_type = 'OBSTACLE'
14
15            if obstacle.active_material is None:
16                mat = bpy.data.materials.new("Material_" + ↩
                      name)
17                obstacle.active_material = mat
18                mat.diffuse_color = Vector((r, g, b, a))
19
20            # Añadimos margen de seguridad
21            x_dim = size.x + 2.0 * margin.x/100.0 * size.x
22            y_dim = size.y + 2.0 * margin.y/100.0 * size.y
23            z_dim = size.z + 2.0 * margin.z/100.0 * size.z
24
25            bpy.ops.mesh.primitive_cube_add()
26            area = bpy.context.active_object
27            area.name = 'margin' + name
28
29            area.dimensions = Vector((x_dim, y_dim, z_dim))
30
31            if area.active_material is None:
32                mat = bpy.data.materials.new("↩
                      Material_margin" + name)
33                area.active_material = mat
34                mat.diffuse_color = Vector((0, 0, 0, 0.2))
35
36            # Centrar area al objeto
37            # Parent set
38            area.location.z += inc_z_location
39            area.object_type = 'OBSTACLE_MARGIN'
40
41            area.select_set(True)
42            obstacle.select_set(True)
43
44            bpy.context.view_layer.objects.active = ↩
```

```
                obstacle
45
46              bpy.ops.object.parent_set()
47
48              area.select_set(False)
49              obstacle.select_set(False)
50
51              area.hide_select = True
```

Apéndice C Creación de paredes introduciendo datos internos del *mesh*

Para crear un nuevo *mesh* a partir de la información de vértices y caras, utilizamos la función *from_pydata* que poseen los *meshes*. Con el nuevo *mesh*, creamos un objeto que añadimos a la escena con la función *object_data_add*.

```
1           # Vertex
2           lfd = Vector((0, 0, 0))
3           lbd = Vector((0, W, 0))
4           rfd = Vector((L, 0, 0))
5           rbd = Vector((L, W, 0))
6           lfu = Vector((0, 0, H))
7           lbu = Vector((0, W, H))
8           rfu = Vector((L, 0, H))
9           rbu = Vector((L, W, H))
10
11          verts = [lfd, lbd, rfd, rbd, lfu, lbu, rfu, rbu]
12
13          edges = [] # Se indican al crear las caras
14
15          foreground = [verts.index(lfd), verts.index(lfu), ↩
                verts.index(rfu), verts.index(rfd)]
16          background = [verts.index(lbd), verts.index(lbu), ↩
                verts.index(rbu), verts.index(rbd)]
17          bottom = [verts.index(lfd), verts.index(lbd), verts↩
                .index(rbd), verts.index(rfd)]
18          top = [verts.index(lbu), verts.index(rbu), verts.↩
                index(rfu), verts.index(lfu)]
19          left = [verts.index(lfu), verts.index(lbu), verts.↩
                index(lbd), verts.index(lfd)]
20          right = [verts.index(rfd), verts.index(rbd), verts.↩
                index(rbu), verts.index(rfu)]
21
22          faces = [foreground, background, bottom, top, left,↩
                 right]
23
24          mesh = bpy.data.meshes.new(name="Wall") # Se crea ↩
                un nuevo mesh
25          mesh.from_pydata(verts, edges, faces) # Se ↩
                introducen los datos de vértices, caras y ↩
```

```
            aristas
26          object_data_add(context, mesh, operator=self)
```

Apéndice D Creación de techos

```
1           def create_ceil(cursor):
2               scene = bpy.context.scene
3
4               # Create plane
5               bpy.ops.mesh.primitive_plane_add(location=↩
                    Vector((cursor.x + 1, cursor.y + 1, cursor.↩
                    z + 0.0)))
6               ceil = bpy.context.object
7
8               save_cursor_loc = bpy.context.scene.cursor.↩
                    location.xyz
9               bpy.context.scene.cursor.location = cursor
10              bpy.ops.object.origin_set(type='ORIGIN_CURSOR')
11
12              # Location
13              x = scene.ceil_props.prop_loc.x
14              y = scene.ceil_props.prop_loc.y
15              z = scene.ceil_props.prop_loc.z
16              # Dimension
17              w = scene.ceil_props.prop_width
18              h = scene.ceil_props.prop_height
19
20              ceil.name = "Ceil"
21              ceil.location = Vector((cursor.x + x, cursor.y ↩
                    + y, cursor.z + z))
22              ceil.dimensions = Vector((w, h, 0))
23
24              bpy.context.scene.cursor.location = ↩
                    save_cursor_loc
25
26              if ceil.active_material is None:
27                  mat = bpy.data.materials.new("↩
                        Material_cursor")
28                  ceil.active_material = mat
29                  mat.diffuse_color = Vector((1, 1, 1, 0.5))
30
31              ceil.object_type = "CEIL"
```

Apéndice E Creación de plataformas robóticas

La función `draw_myrobot` implementa la creación del objeto que representa a *RoboMap* en la escena. Esta función es llamada desde el operador `AddRobotOperator`. Las dos strings que devuelve (el nombre de la platafor-

ma y de objeto que representa el margen de seguridad) son utilizadas, junto a la información aportada por el usuario, para crear una instancia de la clase `MyRobot`, que es almacenada en `RobotSet`.

```
1         def draw_myrobot(context, name, loc, robot_type, ↩
              rot, dim, margin, ip, port):
2
3             # Cuerpo
4             bpy.ops.mesh.primitive_cube_add(location=(loc.x↩
                  , loc.y, dim.z/2.0))
5             myrobot = bpy.context.active_object
6             myrobot.dimensions.xyz = dim.xyz
7             myrobot.rotation_euler.z = radians(rot)
8             myrobot.name = name
9
10            myrobot.lock_location[0:3] = (False, False, ↩
                  True)
11            myrobot.lock_rotation[0:3] = (True, True, False↩
                  )
12            myrobot.lock_scale[0:3] = (True, True, True)
13            myrobot.protected = True
14
15            # margen
16            bpy.ops.mesh.primitive_cube_add(location=↩
                  myrobot.location.xyz[:])
17            myarea = bpy.context.active_object
18            myarea.dimensions.xyz = Vector((dim.x + 2*dim.x↩
                  *(margin.x/100.0), dim.y + 2*dim.y*(margin.↩
                  y/100.0), dim.z + 2*dim.z*(margin.z/100.0))↩
                  )
19            myarea.rotation_euler.z = radians(rot)
20
21            myarea.lock_location[0:3] = (True, True, True)
22            myarea.lock_rotation[0:3] = (True, True, True)
23            myarea.lock_scale[0:3] = (True, True, True)
24            myarea.protected = True
25
26            if myarea.active_material is None:
27                mat = bpy.data.materials.new("↩
                      Material_robot_margin" + name)
28                myarea.active_material = mat
29                mat.diffuse_color = Vector((1, 1, 1, 0.2))
30
31            myarea.object_type = "ROBOT_MARGIN"
32
33            # icosphere
34            minx = min(dim.x, dim.y)
35            bpy.ops.mesh.primitive_ico_sphere_add(radius=↩
                  minx/(2*1.5), location=(loc.x, loc.y, dim.z↩
                  ))
36            myico = bpy.context.active_object
37
38            if myico.active_material is None:
39                mat = bpy.data.materials.new("↩
```

```
                    Material_robot_icosphere" + name)
40                  myico.active_material = mat
41                  mat.diffuse_color = Vector((0, 0, 1, 0.2))
42
43              # Notas
44              note_name = draw_robot_note(context, Vector↩
                    ((0,0,0)), myrobot.name + " | " + str(ip) +↩
                     ":" + str(port), Vector((255,255,255,255))↩
                    , 14, "C")
45
46
47              # Hierarchy area+robot
48              myarea.select_set(True)
49              myrobot.select_set(True)
50              bpy.context.view_layer.objects.active = myrobot
51              bpy.ops.object.parent_set()
52              myarea.select_set(False)
53              myrobot.select_set(False)
54
55              # Set origin
56              myrobot.select_set(True)
57              save_cursor_loc = bpy.context.scene.cursor.↩
                    location.xyz
58              bpy.context.scene.cursor.location.xyz = Vector↩
                    ((loc.x, loc.y, 0))
59              bpy.ops.object.origin_set(type='ORIGIN_CURSOR')
60              bpy.context.scene.cursor.location = ↩
                    save_cursor_loc
61
62              # Hierarchy icosphere + robot
63              myico.select_set(True)
64              myrobot.select_set(True)
65              bpy.context.view_layer.objects.active = myrobot
66              bpy.ops.object.parent_set()
67              myico.select_set(False)
68              myrobot.select_set(False)
69
70              myico.hide_select = True
71              #myrobot.hide_select = False
72              myarea.hide_select = True
73              bpy.data.objects[note_name].parent = myrobot
74
75              bpy.ops.object.camera_add(location=(loc.x, loc.↩
                    y, dim.z + 0.1), rotation=(-pi/2, pi, pi/2 ↩
                    + myrobot.rotation_euler.z))
76              camera = bpy.context.active_object
77              camera.scale = Vector((0.25, 0.25, 0.25))
78
79              camera.select_set(True)
80              myrobot.select_set(True)
81              bpy.context.view_layer.objects.active = myrobot
82              bpy.ops.object.parent_set()
83              camera.select_set(False)
84              myrobot.select_set(False)
```

```
85
86              camera.hide_select = True
87              camera.name = myrobot.name_full + "_camera"
88
89              myrobot.object_type = "ROBOT"
90
91              return myrobot.name, myarea.name
```

Apéndice F Operador de inicio del listener

```
1               class StartPosesListener(bpy.types.Operator):
2                   """
3                   Activa el listener y notifica a los ↩
                        observadores
4                   """
5                   bl_idname = "scene.start_cursor_listener"
6                   bl_label = "Start editor"
7                   bl_description = "Start cursor listener"
8
9                   @classmethod
10                  def poll(cls, context):
11                      exists_robot = len(robot.RobotSet()) > ↩
                            0
12                      robot_selected = context.scene.↩
                            selected_robot_props.prop_robot_id ↩
                            >= 0
13                      running_plan = context.scene.com_props.↩
                            prop_running_nav
14                      paused_plan = context.scene.com_props.↩
                            prop_paused_nav
15                      return not isListenerActive() and ↩
                            exists_robot and robot_selected and↩
                             ((running_plan and paused_plan) or↩
                             (not running_plan))
16
17                  def execute(self, context):
18                      # Indica que se activó el cursor
19                      for scene in bpy.data.scenes:
20                          scene.is_cursor_active = True
21
22                      # Activamos listener
23                      CursorListener.addListener(↩
                            CursorListener())
24                      bpy.app.handlers.depsgraph_update_post.↩
                            append(cursor_update)
25
26                      # Informamos a observers que se activó ↩
                            el listener
27                      for observer in CursorListener.↩
                            _observers:
28                          observer.notifyStart(self)
```

```
29                      return {'FINISHED'}
```

Apéndice G Operador guardar pose

```
1         class SavePoseOperator(bpy.types.Operator):
2             bl_idname = "scene.save_pose"
3             bl_label = "Append Pose"
4             bl_description = "Save pose"
5
6             @classmethod
7             def poll(cls, context):
8                 return context.scene.is_cursor_active
9
10            def execute(self, context):
11                global pd
12                # Valida nuevo Action a añadir (colisiones)
13                pos0 = pd.current_action.p0
14                pos1 = pd.current_action.p1
15                loc0 = pd.current_action.p0.loc
16                loc1 = pd.current_action.p1.loc
17
18                idx = bpy.context.scene.↩
                      selected_robot_props.prop_robot_id
19                robot = robot_tools.RobotSet().getRobot(idx↩
                      )
20                robot_obj = bpy.data.objects[robot.name]
21                area_robot_obj = bpy.data.objects[robot.↩
                      area_name]
22
23                res = is_colliding(idx, robot_obj, ↩
                      area_robot_obj, pos0, pos1)
24
25                self.report({'INFO'}, "Collision : " + str(↩
                      res))
26                if res:
27                    self.report({'ERROR'}, "Robot will ↩
                          collide if takes this path")
28                    return {'FINISHED'}
29                else:
30                    self.report({'INFO'}, "Collision : " + ↩
                          str(res))
31
32
33                # Guardamos nueva action y dibujamos la ↩
                      informacion para la action previa
34                pd.current_action.draw_annotation(context)
35                pc.TempPathContainer().appendAction(pd.↩
                      current_action)
36
37                # Siguiente action
38                p0 = pd.current_action.p1
```

```
39                  p1 = pd.current_action.p1
40                  pd.current_action = path.Action(p0, p1)
41
42                  cl.CursorListener.select_cursor()
43                  return {'FINISHED'}
```

Apéndice H Funciones de chequeo de solapamiento de caras

```
1               """
2               Check a face of a mesh overlap face other mesh
3               """
4
5               def point_in_segment(point, line):
6                   dist = lambda p1, p2: (p2-p1).length
7                   D = dist(line[0], line[1])
8                   d1 = dist(line[0], point)
9                   d2 = dist(point, line[1])
10                  return abs(D - (d1 + d2)) <= bpy.context.↩
                        scene.TOL
11
12              def segments_intersect(line1, line2):
13                  """
14                  Check if two segments intersect
15                  line1: (Vector, Vector)
16                  line2: (Vector, Vector)
17                  :returns: True if line1 and line2 intersect
18                  """
19                  dist = lambda p0, p1: (p1 - p0).length
20
21                  x0, y0 = line1[0], line1[1]
22                  x1, y1 = line2[0], line2[1]
23
24                  res = intersect_line_line(x0, y0, x1, y1)
25                  if res is not None:
26                      r0 = res[0]
27                      r1 = res[1]
28                  # puntos mas cercanos de en una linea a ↩
                        otra linea deben ser iguales
29                  if dist(r0, r1) <= bpy.context.scene.TOL:
30                  # comprobar que el punto de cruce pertenece↩
                         a ambos segmentos
31                  if point_in_segment(r0, line1) and ↩
                        point_in_segment(r0, line2):
32                      return True
33                  return False
34
35              def point_inside_infinite_plane(point, plane):
36                  """
37                  Check if infinite plane contains point
```

```
38                  point: Vector((x,y,z))
39                  plane: BMFace
40                  """
41                  # plane
42                  normal = plane.normal.normalized()
43                  nx, ny, nz = normal.x, normal.y, normal.z
44                  q = plane.verts[0].co
45                  md = nx*q.x + ny*q.y + nz*q.z # constant: -↩
                        d
46                  plane_f = lambda x, y, z : nx*x + ny*y + nz↩
                        *z - md
47
48                  # point
49                  x0, y0, z0 = point.x, point.y, point.z
50
51                  return abs(plane_f(x0, y0, z0)) <= bpy.↩
                        context.scene.TOL
52
53              def point_inside_finite_plane(point, plane):
54                  """
55                  Check if finite plane contains point
56                  point: Vector((x,y,z))
57                  plane: BMFace
58                  """
59                  contain = point_inside_infinite_plane(point↩
                        , plane) # Infinite plane contains ↩
                        point
60                  reflect = bmesh.geometry.↩
                        intersect_face_point(plane, (point.x, ↩
                        point.y, point.z)) # Point reflects ↩
                        over plane
61                  any_vertex_equal = any([(v.co.xyz - point).↩
                        length <= bpy.context.scene.TOL for v ↩
                        in plane.verts]) # Any vertex equals ↩
                        point
62
63                  return (contain and reflect) or ↩
                        any_vertex_equal
64
65              def are_coplanar(plane1, plane2):
66                  """
67                  plane1 and plane2 are both infinite planes
68                  returns: plane1 and plane2 are coplanar
69                  """
70                  # Plano 1
71                  p1_norm = plane1.normal.normalized()
72                  points1 = plane1.verts
73                  # Plano 2
74                  p2_norm = plane2.normal.normalized()
75                  points2 = plane2.verts
76
77                  if p1_norm.length == 0 or p2_norm.length ==↩
                         0:
78                      return False
```

```
79                  angle = p1_norm.angle(p2_norm) # acos(↩
                        p1_norm.dot(p2_norm)/(p1_norm.length*↩
                        p2_norm.length))
80
81                  are_parallel = abs(angle) <= bpy.context.↩
                        scene.TOL or abs(angle - pi) <= bpy.↩
                        context.scene.TOL # planes are parallel↩
                         or coplanar (normal vector angle: 0 or↩
                         pi)
82                  P = points1[0].co.xyz
83                  contains = point_inside_infinite_plane(P, ↩
                        plane2) # plane2 contains a point of ↩
                        plane1
84                  return are_parallel and contains
85
86              def infinite_plane_contains_line(plane, line):
87                  norm = plane.normal.normalized()
88                  v_line = (line[1] - line[0]).normalized()
89
90                  # Infinite plane contains line
91                  prod = norm.dot(v_line)
92                  parallel = prod == 0 or abs(prod) <= bpy.↩
                        context.scene.TOL
93                  contains_point = ↩
                        point_inside_infinite_plane(line[0], ↩
                        plane)
94                  return parallel and contains_point
95
96              def finite_plane_contains_line(plane, line):
97                  """
98                  Checks if plane contains line
99                  plane: BMFace
100                 line: (Vector, Vector)
101                 """
102                 if not infinite_plane_contains_line(plane, ↩
                        line):
103                     # infinite plane does not contains line
104                     return False
105
106                 # Any edge intersect
107                 for edge in plane.edges:
108                 e1 = edge.verts[0].co.xyz
109                 e2 = edge.verts[1].co.xyz
110                 # check intersect
111                 if segments_intersect(line, (e1, e2)):
112                     return True
113                 p1in = point_inside_finite_plane(line[0], ↩
                        plane)
114                 p2in = point_inside_finite_plane(line[1], ↩
                        plane)
115                 return p1in and p2in
116
117             def plane_inside(plane1, plane2):
118                 """
```

```
119                 Check if plane1 is inside plane2
120                 plane1 and plane2 are coplanar and overlap
121                 - plane1, plane2:
122                 BMFace
123                 """
124                 # Plano 1
125                 p1_norm = plane1.normal.normalized()
126                 points1 = plane1.verts
127                 # Plano 2
128                 p2_norm = plane2.normal.normalized()
129                 points2 = plane1.verts
130
131                 # Los planos infinitos que contienen a ↩
                        plane1 y plane2 son coplanares
132                 if not are_coplanar(plane1, plane2):
133                     return False
134                 # Se cruza algun edge o está uno dentro de ↩
                    otro
135                 for edge in plane1.edges:
136                     e1 = edge.verts[0].co.xyz
137                     e2 = edge.verts[1].co.xyz
138                 if finite_plane_contains_line(plane2, (e1, ↩
                    e2)):
139                     return True
140                 for edge in plane2.edges:
141                     e1 = edge.verts[0].co.xyz
142                     e2 = edge.verts[1].co.xyz
143                 if finite_plane_contains_line(plane1, (e1, ↩
                    e2)):
144                     return True
145                 return False
146
147             def face_overlap(bm1, bm2):
148                 """
149                 Check if any face of two bmesh overlap
150                 -
151                 """
152                 for face1 in bm1.faces:
153                     for face2 in bm2.faces:
154                         res_a = plane_inside(face1, face2)
155                         res_b = plane_inside(face2, face1)
156                 if res_a or res_b:
157                     return True
158                 return False
```

Apéndice I Pruebas de solapamiento de caras

```
1
2               def check_overlap(obj1, obj2):
3                   bm1 = cc.create_bmesh(obj1)
4                   bm2 = cc.create_bmesh(obj2)
```

```
5                   res = cc.is_overlapping(bm1, bm2)
6                   bm1.free()
7                   bm2.free()
8                   return res
9
10              def is_overlapping_test():
11                  print("is_overlapping_test: ", end="")
12                  cube1 = oc.create_cube()
13                  cube2 = oc.create_cube()
14                  cube1.location = Vector((1,1,1))
15                  cube2.location = Vector((1,1,1))
16
17                  # 3 dim equals
18                  assert not check_overlap(cube1, cube2), "\↩
                        n3 dim equals test : FAIL"
19                  # 2 dim equals
20                  cube1.dimensions.xyz = Vector((1, 2, 2))
21                  assert not check_overlap(cube1, cube2), "\↩
                        n2 dim equals test : FAIL"
22                  # 1 dim equals
23                  cube1.dimensions.xyz = Vector((1, 1, 2))
24                  assert not check_overlap(cube1, cube2), "\↩
                        n1 dim equals test : FAIL"
25                  # Inside
26                  cube1.dimensions.xyz = Vector((1, 1, 1))
27                  assert not check_overlap(cube1, cube2), "\↩
                        nInside test : FAIL"
28                  # Overlap
29                  cube1.location.x += 1
30                  assert check_overlap(cube1, cube2), "\↩
                        nOverlap test : FAIL"
31                  # Not overlap
32                  cube1.location.x += 10
33                  assert not check_overlap(cube1, cube2), "\↩
                        nNo overlap test : FAIL"
34
35                  oc.delete_object(cube1)
36                  oc.delete_object(cube2)
37
38                  print("passed")
39
40              def points_inside_test():
41                  print("points_inside_test: ", end="")
42                  cube = oc.create_cube()
43                  cube.location.xyz = Vector((1,1,1))
44                  cube.dimensions.xyz = Vector((4,4,4))
45
46                  points = [Vector((0, 0, 0)), Vector((0, 0, ↩
                        0)), Vector((0, 1, 0)),
47                  Vector((0, 1, 1)), Vector((1, 0, 0)), ↩
                        Vector((1, 0, 1)),
48                  Vector((1, 1, 0)), Vector((1, 1, 1)),
49                  Vector((0, 0, 5)),
```

```
50                  Vector((0, 5, 0)), Vector((0, 5, 5)), ↩
                        Vector((5, 0, 0)),
51                  Vector((5, 0, 5)), Vector((5, 5, 0)), ↩
                        Vector((5, 5, 5))]
52                  labels = [True, True, True,
53                  True, True, True,
54                  True, True, False,
55                  False, False, False,
56                  False, False, False]
57                  bm_cube = cc.create_bmesh(cube)
58                  res = cc.points_inside(points, bm_cube)
59                  bm_cube.free()
60
61                  assert all([ex == re for ex, re in zip(↩
                        labels, res)]), "\nPoints inside test :↩
                         FAIL"
62
63                  oc.delete_object(cube)
64                  print("passed")
65
66              def check_is_inside(obj1, obj2):
67                  bm1 = cc.create_bmesh(obj1)
68                  bm2 = cc.create_bmesh(obj2)
69                  res = cc.is_inside(bm1, bm2)
70                  bm1.free()
71                  bm2.free()
72                  return res
73
74              def is_inside_test():
75                  print("is_inside_test: ", end="")
76                  cube1 = oc.create_cube()
77                  cube2 = oc.create_cube()
78                  cube1.location = Vector((1,1,1))
79                  cube2.location = Vector((1,1,1))
80
81                  # 3 dim equals
82                  assert not check_is_inside(cube2, cube1), "↩
                        \n3 dim equals test : FAIL"
83                  # 2 dim equals
84                  cube1.dimensions.xyz = Vector((1, 2, 2))
85                  assert not check_is_inside(cube2, cube1), "↩
                        \n2 dim equals test : FAIL"
86                  # 1 dim equals
87                  cube1.dimensions.xyz = Vector((1, 1, 2))
88                  assert not check_is_inside(cube2, cube1), "↩
                        \n1 dim equals test : FAIL"
89                  # Inside
90                  cube1.dimensions.xyz = Vector((1, 1, 1))
91                  assert check_is_inside(cube2, cube1), "\↩
                        nInside test : FAIL"
92                  # Overlap
93                  cube1.location.x += 1
94                  assert check_is_inside(cube2, cube1), "\↩
                        Inside test : FAIL"
```

```
95                  # Not overlap
96                  cube1.location.x += 10
97                  assert not check_is_inside(cube2, cube1), "↩
                        \nNo inside test : FAIL"
98
99                  oc.delete_object(cube1)
100                 oc.delete_object(cube2)
101
102                 print("passed")
103
104             def point_in_segment_test():
105                 print("point_in_segment_test : ", end="")
106                 p1 = Vector((1, -1, 1))
107                 p2 = Vector((-1, 3, 2))
108
109                 q2 = Vector((0, 1, 3/2))
110                 q1 = Vector((1/2, 0, 5/4))
111                 q3 = Vector((-1/2, 2, 7/4))
112
113                 q2_2d = Vector((0, 1, 0))
114                 q1_2d = Vector((1/2, 0, 0))
115                 q3_2d = Vector((-1/2, 2, 0))
116
117                 assert cc.point_in_segment(p1, (p1,p2)), "↩
                        Point in segment : FAIL"
118                 assert cc.point_in_segment(p2, (p1,p2)), "↩
                        Point in segment : FAIL"
119
120                 assert cc.point_in_segment(q1, (p1,p2)), "↩
                        Point in segment : FAIL"
121                 assert cc.point_in_segment(q2, (p1,p2)), "↩
                        Point in segment : FAIL"
122                 assert cc.point_in_segment(q3, (p1,p2)), "↩
                        Point in segment : FAIL"
123
124                 assert not cc.point_in_segment(Vector((1, ↩
                        1, 1)), (p1,p2)), "Point not in segment↩
                         : FAIL"
125                 assert not cc.point_in_segment(Vector((-1, ↩
                        -1, -1)), (p1,p2)), "Point not in ↩
                        segment : FAIL"
126                 assert not cc.point_in_segment(Vector((1, ↩
                        1, 1)), (p1,p2)), "Point not in segment↩
                         : FAIL"
127
128                 assert not cc.point_in_segment(q1_2d, (p1,↩
                        p2)), "Point not in segment : FAIL"
129                 assert not cc.point_in_segment(q2_2d, (p1,↩
                        p2)), "Point not in segment : FAIL"
130                 assert not cc.point_in_segment(q3_2d, (p1,↩
                        p2)), "Point not in segment : FAIL"
131
132                 print("passed")
133
```

```
134             def point_inside_infinite_plane_test():
135                 print("point_inside_infinite_plane: ", end=↩
                        "")
136 
137                 plane1 = oc.create_plane()
138                 plane1.location.xyz = Vector((1, 2, 1))
139                 ry = Matrix.Rotation(math.radians(-45.0), ↩
                        3, 'X')
140                 plane1.rotation_euler = Vector(ry.to_euler↩
                        ())
141                 plane1_f = lambda x, y: 3-y
142 
143                 bm_plane1 = cc.create_bmesh(plane1)
144                 face_plane1 = bm_plane1.faces[0]
145 
146                 for x in np.linspace(-1, 3, 5):
147                     for y in np.linspace(0, 3, 5):
148                         z = plane1_f(x, y)
149                         ppoint = Vector((x, y, z))
150                         #oc.create_point(ppoint)
151                         assert  cc.↩
                                point_inside_infinite_plane(↩
                                ppoint, face_plane1), "\nInside↩
                                 test : FAIL"
152                         z = random.randint(-2, 2)
153                         ppoint.z += 1 if z == 0 else z
154                         #oc.create_point(ppoint)
155                         assert not cc.↩
                                point_inside_infinite_plane(↩
                                ppoint, face_plane1), "\nNot ↩
                                inside test : FAIL"
156 
157 
158                 plane2 = oc.create_plane_from_points(Vector↩
                        ((0, 0, 1)), Vector((1, 0, 0)), Vector↩
                        ((1, 1, -1)), Vector((0, 1, 0)))
159                 plane2_f = lambda x, y: 1 - x - y
160 
161                 bm_plane2 = cc.create_bmesh(plane2)
162                 bm_plane2.faces.ensure_lookup_table()
163                 face_plane2 = bm_plane2.faces[0]
164 
165                 for x in np.linspace(-1, 2, 8):
166                     for y in np.linspace(-1, 2, 8):
167                         z = plane2_f(x, y)
168                         ppoint = Vector((x, y, z))
169                         assert cc.↩
                                point_inside_infinite_plane(↩
                                ppoint, face_plane2), "\nInside↩
                                 test : FAIL"
170 
171                 z = random.randint(-2, 2)
172                 ppoint.z += 1 if z == 0 else z
```

```
173                 assert not cc.point_inside_infinite_plane(↩
                        ppoint, face_plane2), "\nNot inside ↩
                        test : FAIL"
174
175
176                 oc.delete_object(plane1)
177                 oc.delete_object(plane2)
178                 bm_plane1.free()
179                 bm_plane2.free()
180                 print("passed")
181
182             def point_inside_finite_plane_test():
183                 print("point_inside_finite_plane: ", end=""↩
                        )
184
185                 plane1 = oc.create_plane()
186                 plane1.location.xyz = Vector((2, 2, 2))
187                 ry = Matrix.Rotation(math.radians(90.0), 3,↩
                         'X')
188                 plane1.rotation_euler = Vector(ry.to_euler↩
                        ())
189                 plane1_f = lambda x, z: 2
190
191                 bm_plane1 = cc.create_bmesh(plane1)
192
193                 face_plane1 = bm_plane1.faces[0]
194
195                 for x in np.linspace(0, 4, 8):
196                     for z in np.linspace(0, 4, 8):
197                         y = plane1_f(x, z)
198                         ppoint = Vector((x, y, z))
199                         inside = 1 <= x <= 3 and 1 <= z <= ↩
                                3 and y == 2
200                         assert inside == cc.↩
                                point_inside_finite_plane(↩
                                ppoint, face_plane1), "Inside ↩
                                test: FAIL"
201
202                 y = random.randint(-2, 2)
203                 ppoint.y += 1 if y == 0 else y
204                 assert not cc.point_inside_finite_plane(↩
                        ppoint, face_plane1), "Inside test: ↩
                        FAIL"
205
206                 oc.delete_object(plane1)
207                 bm_plane1.free()
208                 print("passed")
209
210             def coplanar_check(plane1, plane2):
211                 bm1 = cc.create_bmesh(plane1)
212                 bm2 = cc.create_bmesh(plane2)
213
214                 bm1_face = bm1.faces[0]
215                 bm2_face = bm2.faces[0]
```

```
216
217                 res = cc.are_coplanar(bm1_face, bm2_face)
218
219                 bm1.free()
220                 bm2.free()
221                 return res
222
223             def are_coplanar_test():
224                 print("are_coplanar: ", end="")
225                 plane1 = oc.create_plane()
226                 plane2 = oc.create_plane()
227
228                 plane1.location = Vector((1, 1, 1))
229                 res1 = coplanar_check(plane1, plane2) # ↩
                        false
230                 assert not res1, "Not coplanar test: FAIL"
231                 plane2.location = Vector((1, 1, 1))
232                 res2 = coplanar_check(plane1, plane2) # ↩
                        true
233                 assert res2, "Coplanar test: FAIL"
234
235
236                 plane1.rotation_euler = Vector((45, 90, ↩
                        -40))
237                 res3 = coplanar_check(plane1, plane2) # ↩
                        false
238                 assert not res3, "Not coplanar test: FAIL"
239                 plane2.rotation_euler = Vector((45, 90, ↩
                        -40))
240                 res4 = coplanar_check(plane1, plane2) # ↩
                        true
241                 assert res4, "Coplanar test: FAIL"
242
243                 plane1.location += Vector((1, 1, 1))
244                 res5 = coplanar_check(plane1, plane2) # ↩
                        false
245                 assert not res5, "Not coplanar test: FAIL"
246                 plane2.location += Vector((1, 1, 1))
247                 res6 = coplanar_check(plane1, plane2) # ↩
                        true
248                 assert res6, "Coplanar test: FAIL"
249
250                 plane1.rotation_euler = Vector((2.0*math.pi↩
                        /4.0, 0, 0))
251                 plane2.rotation_euler = Vector((2.0*math.pi↩
                        /4.0, 0, 0))
252                 plane1.location += Vector((0, 0, 10))
253                 res1 = coplanar_check(plane1, plane2) # ↩
                        true
254                 assert res1, "Coplanar test: FAIL"
255
256                 oc.delete_object(plane1)
257                 oc.delete_object(plane2)
258                 print("passed")
```

```
259
260             def infinite_plane_contains_line_test():
261                 print("infinite_plane_contains_line : ", ↩
                        end="")
262                 plane = oc.create_plane()
263                 plane.rotation_euler = Vector((0, 2*math.pi↩
                        /8, 0))
264                 plane.location.xyz += Vector((1, 0, 0))
265                 bm_plane = cc.create_bmesh(plane)
266
267                 plane_face = bm_plane.faces[0]
268
269                 plane_f = lambda x, y: 1-x
270
271
272                 O = Vector((1, 0, 0))
273                 a = Vector((1 - math.sqrt(2.0)/2.0, -1, +↩
                        math.sqrt(2.0)/2.0))
274                 b = Vector((1 - math.sqrt(2.0)/2.0, +1, +↩
                        math.sqrt(2.0)/2.0))
275                 c = Vector((1 + math.sqrt(2.0)/2.0, -1, -↩
                        math.sqrt(2.0)/2.0))
276                 d = Vector((1 + math.sqrt(2.0)/2.0, +1, -↩
                        math.sqrt(2.0)/2.0))
277                 e = Vector((1.0/2.0, 0, 1.0/2.0))
278                 f = Vector((1 -math.sqrt(2.0)/2.0, -3, math↩
                        .sqrt(2.0)/2.0))
279
280                 P = Vector((1, -2, 0))
281                 Q = Vector((1, -1.0/2.0, 0))
282                 R = Vector((1, +1.0/2.0, 0))
283                 S = Vector((1, +2, 0))
284                 T = Vector((1, -1, 0))
285                 U = Vector((1, 1, 0))
286
287                 X1 = Vector((-math.sqrt(2.0)/2.0, -1, 1))
288                 X2 = Vector((2 + math.sqrt(2.0)/2.0, 1, 0))
289                 X3 = Vector((+10, +10, +10))
290                 X4 = Vector((-10, -10, -10))
291                 X5 = Vector((-math.sqrt(2.0)/2.0, +1, 0))
292                 X6 = Vector((2+math.sqrt(2.0)/2.0, -1, 0))
293
294                 # true
295                 line1 = b, c
296                 line2 = a, c
297                 line3 = P, R
298                 line4 = P, Q
299                 line5 = R, S
300                 line6 = T, U
301                 line7 = e, S
302                 line8 = P, f
303                 # false
304                 line9 = X1, X2
305                 line10 = X1, X3
```

```
306                 line11 = X1, X4
307                 line12 = X5, X6
308
309                 assert cc.infinite_plane_contains_line(↩
                        plane_face, line1), "Plane contains ↩
                        line test: FAIL"
310                 assert cc.infinite_plane_contains_line(↩
                        plane_face, line2), "Plane contains ↩
                        line test: FAIL"
311                 assert cc.infinite_plane_contains_line(↩
                        plane_face, line3), "Plane contains ↩
                        line test: FAIL"
312                 assert cc.infinite_plane_contains_line(↩
                        plane_face, line4), "Plane contains ↩
                        line test: FAIL"
313                 assert cc.infinite_plane_contains_line(↩
                        plane_face, line5), "Plane contains ↩
                        line test: FAIL"
314                 assert cc.infinite_plane_contains_line(↩
                        plane_face, line6), "Plane contains ↩
                        line test: FAIL"
315                 assert cc.infinite_plane_contains_line(↩
                        plane_face, line7), "Plane contains ↩
                        line test: FAIL"
316                 assert cc.infinite_plane_contains_line(↩
                        plane_face, line8), "Plane contains ↩
                        line test: FAIL"
317
318                 assert not cc.infinite_plane_contains_line(↩
                        plane_face, line9), "Plane not contains↩
                         line test: FAIL"
319                 assert not cc.infinite_plane_contains_line(↩
                        plane_face, line10), "Plane not ↩
                        contains line test: FAIL"
320                 assert not cc.infinite_plane_contains_line(↩
                        plane_face, line11), "Plane not ↩
                        contains line test: FAIL"
321                 assert not cc.infinite_plane_contains_line(↩
                        plane_face, line12), "Plane not ↩
                        contains line test: FAIL"
322
323                 oc.delete_object(plane)
324                 bm_plane.free()
325
326                 print("passed")
327
328             def finite_plane_contains_line_test():
329                 print("finite_plane_contains_line : ", end=↩
                        "")
330                 plane = oc.create_plane()
331                 plane.rotation_euler = Vector((0, 2*math.pi↩
                        /8, 0))
332                 plane.location.xyz += Vector((1, 0, 0))
333                 bm_plane = cc.create_bmesh(plane)
```

```
334
335                 plane_face = bm_plane.faces[0]
336
337                 plane_f = lambda x, y: 1-x
338
339
340                 O = Vector((1, 0, 0))
341                 a = Vector((1 - math.sqrt(2.0)/2.0, -1, +↩
                        math.sqrt(2.0)/2.0))
342                 b = Vector((1 - math.sqrt(2.0)/2.0, +1, +↩
                        math.sqrt(2.0)/2.0))
343                 c = Vector((1 + math.sqrt(2.0)/2.0, -1, -↩
                        math.sqrt(2.0)/2.0))
344                 d = Vector((1 + math.sqrt(2.0)/2.0, +1, -↩
                        math.sqrt(2.0)/2.0))
345                 e = Vector((1.0/2.0, 0, 1.0/2.0))
346                 f = Vector((1 -math.sqrt(2.0)/2.0, -3, math↩
                        .sqrt(2.0)/2.0))
347
348                 P = Vector((1, -2, 0))
349                 Q = Vector((1, -1.0/2.0, 0))
350                 R = Vector((1, +1.0/2.0, 0))
351                 S = Vector((1, +2, 0))
352                 T = Vector((1, -1, 0))
353                 U = Vector((1, 1, 0))
354
355                 X1 = Vector((-math.sqrt(2.0)/2.0, -1, 1))
356                 X2 = Vector((2 + math.sqrt(2.0)/2.0, 1, 0))
357                 X3 = Vector((+10, +10, +10))
358                 X4 = Vector((-10, -10, -10))
359                 X5 = Vector((-math.sqrt(2.0)/2.0, +1, 0))
360                 X6 = Vector((2+math.sqrt(2.0)/2.0, -1, 0))
361
362                 # true
363                 line1 = b, c
364                 line2 = a, c
365                 line3 = P, R
366                 line4 = P, Q
367                 line5 = R, S
368                 line6 = T, U
369                 line7 = e, S
370                 # false
371                 line8 = P, f
372                 line9 = X1, X2
373                 line10 = X1, X3
374                 line11 = X1, X4
375                 line12 = X5, X6
376
377                 assert cc.finite_plane_contains_line(↩
                        plane_face, line1), "Plane contains ↩
                        line test: FAIL"
378                 assert cc.finite_plane_contains_line(↩
                        plane_face, line2), "Plane contains ↩
                        line test: FAIL"
```

```
379                 assert cc.finite_plane_contains_line(↩
                        plane_face, line3), "Plane contains ↩
                        line test: FAIL"
380                 assert cc.finite_plane_contains_line(↩
                        plane_face, line4), "Plane contains ↩
                        line test: FAIL"
381                 assert cc.finite_plane_contains_line(↩
                        plane_face, line5), "Plane contains ↩
                        line test: FAIL"
382                 assert cc.finite_plane_contains_line(↩
                        plane_face, line6), "Plane contains ↩
                        line test: FAIL"
383                 assert cc.finite_plane_contains_line(↩
                        plane_face, line7), "Plane contains ↩
                        line test: FAIL"
384
385                 assert not cc.finite_plane_contains_line(↩
                        plane_face, line8), "Plane not contains↩
                         line test: FAIL"
386                 assert not cc.finite_plane_contains_line(↩
                        plane_face, line9), "Plane not contains↩
                         line test: FAIL"
387                 assert not cc.finite_plane_contains_line(↩
                        plane_face, line10), "Plane not ↩
                        contains line test: FAIL"
388                 assert not cc.finite_plane_contains_line(↩
                        plane_face, line11), "Plane not ↩
                        contains line test: FAIL"
389                 assert not cc.finite_plane_contains_line(↩
                        plane_face, line12), "Plane not ↩
                        contains line test: FAIL"
390
391                 oc.delete_object(plane)
392                 bm_plane.free()
393
394                 print("passed")
395
396             def plane_inside_check(plane1, plane2):
397                 bm1 = cc.create_bmesh(plane1)
398                 bm2 = cc.create_bmesh(plane2)
399
400                 bm1_face = bm1.faces[0]
401                 bm2_face = bm2.faces[0]
402
403                 res = cc.plane_inside(bm1_face, bm2_face)
404
405                 bm1.free()
406                 bm2.free()
407                 return res
408
409             def plane_inside_test():
410                 print("plane_inside : " , end="")
411
412                 plane1 = oc.create_plane()
```

```
413                 plane2 = oc.create_plane()
414
415                 plane1.rotation_euler = Vector((0, 2*math.↩
                        pi/8, 0))
416                 plane2.rotation_euler = Vector((0, 2*math.↩
                        pi/8, 0))
417                 plane1.location.xyz = Vector((1, 1, 0))
418                 plane2.location.xyz = Vector((1, 1, 0))
419
420                 assert plane_inside_check(plane1, plane2), ↩
                        "Plane inside test : FAIL"
421
422                 plane2.location.xyz = Vector((1, 2, 0))
423                 assert plane_inside_check(plane1, plane2), ↩
                        "Plane inside test : FAIL"
424
425                 plane2.location.xyz = Vector((0, 1, 1))
426                 assert plane_inside_check(plane1, plane2), ↩
                        "Plane inside test : FAIL"
427
428                 plane2.location.xyz = Vector((0, 0, 1))
429                 assert plane_inside_check(plane1, plane2), ↩
                        "Plane inside test : FAIL"
430
431                 plane2.dimensions.y = 1
432                 assert plane_inside_check(plane1, plane2), ↩
                        "Plane inside test : FAIL"
433
434                 plane2.dimensions.x = 1
435                 plane2.location.xyz = Vector((1, 1, 0))
436                 assert plane_inside_check(plane1, plane2), ↩
                        "Plane inside test : FAIL"
437
438                 plane2.location.xyz = Vector((1, -2, 0))
439                 assert not plane_inside_check(plane1, ↩
                        plane2), "Plane not inside test : FAIL"
440
441                 plane2.location.xyz = Vector((4, -2, 0))
442                 assert not plane_inside_check(plane1, ↩
                        plane2), "Plane not inside test : FAIL"
443
444                 plane1.rotation_euler = Vector((0, 0, 0))
445                 plane2.rotation_euler = Vector((0, 0, 0))
446                 plane1.location.xyz = Vector((1, 1, 0))
447                 plane2.location.xyz = Vector((1, 1, 0))
448
449                 assert plane_inside_check(plane1, plane2), ↩
                        "Plane inside test : FAIL"
450
451                 plane2.location.xyz = Vector((1, 1, 1))
452                 assert not plane_inside_check(plane1, ↩
                        plane2), "Plane inside test : FAIL"
453
```

```
454                 plane1.rotation_euler = Vector((0, 2*math.↩
                        pi/4, 0))
455                 plane2.location.xyz += Vector((0, 0, 1))
456                 assert not plane_inside_check(plane1, ↩
                        plane2), "Plane inside test : FAIL"
457
458                 print("passed")
459
460                 oc.delete_object(plane1)
461                 oc.delete_object(plane2)
462
463             def face_overlap_check(obj1, obj2):
464                 bm1 = cc.create_bmesh(obj1)
465                 bm2 = cc.create_bmesh(obj2)
466
467                 res = cc.face_overlap(bm1, bm2)
468
469                 bm1.free()
470                 bm2.free()
471
472                 return res
473
474             def face_overlap_test():
475                 print("face_overlap: ", end="")
476                 cube1 = oc.create_cube()
477                 cube2 = oc.create_cube()
478
479                 cube1.location.xyz = Vector((1, 1, 1))
480                 cube2.location.xyz = Vector((1, 1, 1))
481                 assert face_overlap_check(cube1, cube2), "↩
                        Face overlap test: FAIL"
482
483                 cube2.location.xyz = Vector((1, 1, 3))
484                 assert face_overlap_check(cube1, cube2), "↩
                        Face overlap test: FAIL"
485
486                 cube2.location.xyz = Vector((1, 1, -1))
487                 assert face_overlap_check(cube1, cube2), "↩
                        Face overlap test: FAIL"
488
489                 cube2.location.xyz = Vector((1, 3, 1))
490                 assert face_overlap_check(cube1, cube2), "↩
                        Face overlap test: FAIL"
491
492                 cube2.location.xyz = Vector((1, 3, 1))
493                 assert face_overlap_check(cube1, cube2), "↩
                        Face overlap test: FAIL"
494
495                 cube2.location.xyz = Vector((1, -1, 1))
496                 assert face_overlap_check(cube1, cube2), "↩
                        Face overlap test: FAIL"
497
498                 cube2.location.xyz = Vector((3, 1, 1))
```

```
499                 assert face_overlap_check(cube1, cube2), "↩
                        Face overlap test: FAIL"
500
501                 cube2.location.xyz = Vector((-1, 1, 1))
502                 assert face_overlap_check(cube1, cube2), "↩
                        Face overlap test: FAIL"
503
504                 cube2.location.xyz = Vector((1, 1, 1))
505                 cube2.dimensions.x = 1
506                 assert face_overlap_check(cube1, cube2), "↩
                        Face overlap test: FAIL"
507
508                 cube2.dimensions.y = 1
509                 assert face_overlap_check(cube1, cube2), "↩
                        Face overlap test: FAIL"
510
511                 cube2.dimensions.z = 1
512                 assert not face_overlap_check(cube1, cube2)↩
                        , "Face overlap test: FAIL"
513
514                 cube2.location.xyz = Vector((2, 1, 1))
515                 assert not face_overlap_check(cube1, cube2)↩
                        , "Face overlap test: FAIL"
516
517                 cube2.location.xyz = Vector((4, 1, 1))
518                 assert not face_overlap_check(cube1, cube2)↩
                        , "Face overlap test: FAIL"
519
520                 cube2.location.xyz = Vector((4, 4, 4))
521                 assert not face_overlap_check(cube1, cube2)↩
                        , "Face overlap test: FAIL"
522
523                 oc.delete_object(cube1)
524                 oc.delete_object(cube2)
525
526                 print("passed")
527
528             print("==============================")
529             print(datetime.datetime.now())
530             is_overlapping_test()
531
532             is_inside_test()
533
534             face_overlap_test()
535
536             print("All test passed!")
537             print("==============================")
```