Trabajo de Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Desarrollo de algoritmos de planificación de caminos en 3D para Blender y V-REP

Autor: Miguel Fernández-Peteiro Belmonte
Tutor: Iván Maza Alcañiz

Intensificación: Sistemas de Telecomunicación

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

Sevilla, 2016

Trabajo de Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Desarrollo de algoritmos de planificación de caminos en 3D para Blender y V-REP

Autor:
Miguel Fernández-Peteiro Belmonte

Tutor:
Iván Maza Alcañiz
Profesor titular

Dep. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2016

Trabajo de Fin de Grado: Desarrollo de algoritmos de planificación de caminos en 3D para Blender y V-REP

Autor: Miguel Fernández-Peteiro Belmonte

Tutor: Iván Maza Alcañiz

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2016

El Secretario del Tribunal

Agradecimientos

Me gustaría agradecer a mi familia sus ánimos y apoyo incondicional durante todos estos años. También a mis amigos y compañeros por su ayuda y consejos cuando lo he necesitado.

Finalmente, quería agradecer a mi tutor Iván Maza por su ayuda con el proyecto, siempre con la puerta abierta para orientarme y atender cualquiera de mis dudas.

Gracias a todos.

Miguel Fernández-Peteiro Belmonte

Sevilla, 2016

Resumen

La planificación de caminos es una herramienta muy frecuentemente empleada en el ámbito de la ingeniería y la robótica. Permite encontrar el camino más corto entre dos localizaciones dentro de un escenario, evitando los obstáculos entre el punto de origen y el punto de destino. De esta forma un móvil, robot o cualquiera que sea nuestro sujeto es capaz de determinar un camino óptimo que esquive estos obstáculos planeando así una ruta de forma autónoma.

Este trabajo de fin de grado presenta una visión general de la creación de rutas óptimas en diversos entornos, ya sean reales o virtuales. Para ello analizaremos una serie de algoritmos de búsqueda que se irán desarrollando a lo largo de la memoria. Estos algoritmos tendrán que leer un mapa o entorno de trabajo, en el que se presentarán todos los obstáculos a evitar entre el punto de origen y el destino. Cada uno de estos algoritmos será explicado en detalle para entender su lógica y modo de funcionamiento.

Los algoritmos empleados serán tres: Dijkstra, A estrella (A*) y RRT (*Rapidly Exploring Random Trees*). Cada uno de ellos tendrá sus peculiaridades, por lo que analizaremos los pros y los contras de usar uno y otro algoritmo según la situación y el contexto en el que nos encontremos.

El trabajo además consiste en la implementación de estos algoritmos en dos entornos de desarrollo 3D. Los elegidos han sido Blender y V-REP, ambos de código abierto y que presentan una gran cantidad de posibilidades de simulación. Una vez explicados los algoritmos para el planeamiento de rutas, nos centraremos en cómo programarlos en cada caso, analizando las peculiaridades de cada uno y las posibilidades que nos ofrecen sus respectivos lenguajes de programación.

Por último, se mostrarán los resultados obtenidos después de todo el desarrollo previo, analizando las animaciones creadas tanto en 2D como en 3D. Para ello se emplearán las herramientas de animación facilitadas por ambos programas. Además, se utilizarán otras herramientas externas tales como el Pygame, la cual nos ayudará a entender y seguir el desarrollo de los algoritmos de búsqueda en 2D.

Índice

Agradecimientos **v**
Resumen **vii**
Índice **ix**
1 Planificación de caminos. Visión general y planteamiento de objetivos ¡Error! Marcador no definido.
2 Algoritmos a implementar. Explicación y desarrollo **5**
2.1. Algoritmo Dijkstra 6
2.2. Algoritmo A 10*
2.3. Algoritmo RRT 14
2.4. Rendimiento de los algoritmos en distintos escenarios 18
3 Descripción general de Blender e implementación de algoritmos **25**
3.1. Historia de Blender 25
3.2. Descripción general 26
3.3. Interfaz de Blender 28
3.4. Lenguaje Python 30
3.4.1. Fundamentos básicos 30
3.4.2. API de Blender para Python 31
3.5. Implementación del algoritmos y animaciones 32
4 V-REP. Implementación de algoritmos en el ámbito de la robótica **37**
4.1. Descripción general 37
4.2. Interfaz de V-REP 39
4.3. Lenguaje Lua 40
4.4. Motion planning dentro de V-REP 42
4.5. Implementación de algoritmos 44
5 Capitulo de resultados: Animaciones en Blender y V-REP **49**
5.1. Resultados con Blender 49
5.2. Resultados con V-REP 54
6 Conclusiones finales y futuras líneas de investigación **59**
6.1. Conclusiones 59
6.2. Líneas futuras 59
6.3. Comentarios finales 60
Bibliografía y referencias **63**

Anexo A. Códigos para blender: Algoritmos y animaciones **65**
A.1. Dijkstra en 3D *65*
A.2. A en 3D* *67*
A.3. RRT en 3-D *69*
A.4. Funciones comunes, variables y llamadas a las funciones *81*

Anexo B. Códigos para V-REP: Algoritmos y animaciones **85**
B.1. Algoritmo Dijkstra en 3D *85*
B.2. Algoritmo A en 3D* *91*

Anexo C. Animaciones en 2D para pygame **99**
C.1. Algoritmo Dijkstra en 2D *107*
C.2. Algoritmo A en 2D* *115*
C.3. Algoritmo RRT en 2D

Notación

A*	A estrella
D*	D estrella
RRT	Rapidly Exploring Random Tree
V-REP	Virtual Robot Experimentation Platform
API	Application Programming Interface
GPL	General Public License
OMPL	Open Motion Planning Library
dt	Simulation time step
$\leq$	Menor o igual
$\geq$	Mayor o igual
$\Leftrightarrow$	Si y sólo si

1 PLANIFICACIÓN DE CAMINOS. VISIÓN GENERAL Y PLANTEAMIENTO DE OBJETIVOS

El esfuerzo de utilizar las máquinas para emular el pensamiento humano siempre me ha parecido bastante estúpido. Preferiría usarlas para emular algo mejor.

- Edsger Dijkstra, científico de la computación holandés que decribió por primera ver el algoritmo de caminos mínimos en 1959 -

La búsqueda de caminos, más comúnmente conocida en el ámbito de la ingeniería como "*path planning*", consiste en la determinación de acciones necesarias para pasar de un estado inicial a un estado final deseado. En esta búsqueda los componentes de la ruta representan localizaciones y transiciones que nuestro elemento activo (ya sea un móvil, robot, persona etc.) puede realizar, cada una con un coste asociado.

Decimos que una ruta es óptima cuando la suma de sus costes es lo menor posible desde que abandona su estado inicial hasta que alcanza su destino. Para que el algoritmo sea óptimo, tendrá que encontrarla siempre que ésta exista. Además, tendrá que ser capaz de indicarnos si no existe esa ruta que buscamos.

Para entender al detalle el funcionamiento de los algoritmos de búsqueda éstos se explicarán en dos dimensiones. Posteriormente, a la hora de implementarlos tanto en Blender como en V-REP sí que tendremos en cuenta el eje Z. Sin embargo, sería mucho más complicado apreciar el funcionamiento directamente en una representación 3D, por lo que en este apartado del trabajo los esquemas que veremos se realizarán sobre los ejes XY.

En una situación inicial, el elemento que queremos desplazar desde el origen hasta el punto destino no sabe en qué entorno está ni qué camino seguir para alcanzar el su objetivo. La misión del algoritmo consiste básicamente en la búsqueda de la línea verde que vemos en la siguiente figura, uniendo ambos puntos a la vez que evita los obstáculos representados por bloques negros.

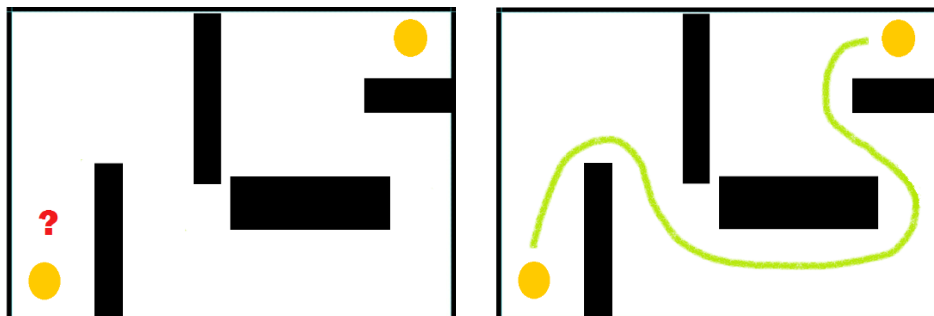


Figura 1.1

Por eso, lo primero que hay que pasarle al programa como dato es el entorno (ya sea real o virtual) en el que desarrollará la búsqueda de la ruta. Este entorno sobre el que se llevará a cabo todo el algoritmo de búsqueda lo denominaremos como "mapa" o "grid", y en él vendrán representados todos los obstáculos a evitar.

El primer paso será por tanto discretizar el contexto en el que trabajaremos. Esto consiste en analizar el espacio en el que nos encontramos y ver si está libre o no. Para entenderlo, imaginémonos que estamos en una simple cocina de una casa a la que se le han añadido una serie de cuadrículas en el suelo que dividen el espacio.

Esta cocina podría representarse como una matriz de datos. Para ello, a lo largo de todo el trabajo supondremos que el "1" representa espacio ocupado o a evitar. A su vez, el "0" representará espacios libres por los que nos podremos mover.

Así, la siguiente cocina de ejemplo vendría representada aproximadamente por la siguiente matriz de datos:

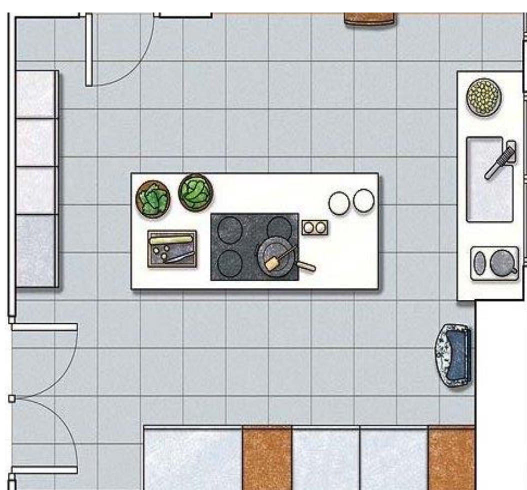


Figura 1.2 – Esquema de una cocina

$$
\begin{aligned}
\textbf{\textit{Mapa}} = [&[0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0],\\
&[1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1],\\
&[1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1],\\
&[1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 1],\\
&[1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 1],\\
&[1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1\ 1],\\
&[0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1],\\
&[0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1],\\
&[0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1],\\
&[0\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1]]
\end{aligned}
$$

La discretización como vemos no es exacta del todo, ya que algunas cuadrículas figuran como libres y están ligeramente ocupadas por una parte, o viceversa. No obstante, esta matriz sí que nos da una representación muy aproximada a la realidad. En cualquier caso, todo dependerá del contexto en el que trabajemos. Si en el caso de la cocina que hemos visto quisiéramos una exactitud mayor, podríamos dividir cada una de las celdas entre cuatro. Pasaríamos entonces a tener una matriz del entorno de dimensión 48 x 40 en vez de 12 x 10. Todo dependerá de la situación en la que se realice la discretización, el grado de exactitud necesario, la capacidad que tengamos para discretizar un entorno etc.

Basándonos en esta idea se desarrollarán todos los algoritmos a lo largo del trabajo, en el que utilizaremos mapas de obstáculos muy diversos, desde matrices con pocas filas y columnas hasta espacios mucho más amplios. Sin embargo, puede que no nos limitemos simplemente a encontrar el camino buscado. En muchos videojuegos y entorno virtuales o situaciones de la vida real se nos puede presentar el reto tener en cuenta otros factores. Es posible que además busquemos reducir el coste empleado en recorrer ese camino o el tiempo que nos supone.

Esto implica que un dato a tener en cuenta antes de empezar a explicar los algoritmos de búsqueda es el de los costes de cada movimiento y los posibles movimientos a realizar. Aunque en la mayoría de las situaciones todos los movimientos suponen el mismo coste o esfuerzo, se pueden dar situaciones en las que esto no suceda. En el desarrollo de los algoritmos entraremos con más detalle en este punto.

En nuestro caso, los posibles movimientos a realizar serán 4 en 2 dimensiones (arriba, abajo, izquierda, derecha) y 6 en el caso tridimensional, no pudiendo realizar el robot desplazamientos diagonales. Además, a todos los movimientos les asignaremos el coste 1, aunque como se acaba de explicar, no siempre tendría por qué ser así y podría influir en la resolución del algoritmo.

Para finalizar, tendremos que tener en cuenta que una vez implementamos un algoritmo en un programa, el método de búsqueda no variará en función del tamaño del mapa. Sí lo hará en cambio el coste computacional, creciendo considerablemente a medida que tenga que analizar espacios mayores, ya que cuanto más grande es el espacio a analizar, más cálculos hay que llevar a cabo.

Tras haber visto los fundamentos que rigen el comportamiento de los algoritmos, intentaremos extrapolar toda la teoría desarrollada a situaciones tanto reales como virtuales. Para ello profundizaremos en dos softwares de diseño 3D, Blender y V-REP, explorando en las posibilidades que estos nos ofrecen e implementando el código necesario para realizar animaciones tridimensionales.

De esta forma, podemos enumerar los objetivos del proyeto de la siguiente manera:

- Explicación y desarrollo de los distintos algoritmos de búsqueda de caminos. Profundizaremos en su fundamento teórico y analizaremos los pros y los contras de cada uno de ellos.

- Implementación de algoritmos y animaciones en Blender, explicando cómo se trabaja con este potente software de animación tridimensional.

- Implementación de algoritmos y animaciones en V-REP, un software enfocado a la robótica que nos permitirá indagar también en el funcionamiento de diversos prototipos ya implementados.

2 ALGORITMOS A IMPLEMENTAR. EXPLICACIÓN Y DESARROLLO

Cuando el mapa del escenario viene representado por una matriz, emplearemos lo que se define como algoritmos de búsqueda gráfica. A lo largo de este capítulo se explicarán paso por paso tres de los algoritmos básicos dentro de la búsqueda de caminos: Dijkstra, A* y RRT.

No obstante, hay que tener en cuenta que serán desarrollados de la forma lo más sencilla posible para comprender su funcionamiento. No profundizaremos en exceso en todas las posibilidades que estos algoritmos nos ofrecen, ya que nuestra misión es entenderlos para posteriormente poder implementarlos en un entorno de animación 3D. Sin embargo hay que tener en cuenta que, aunque en este trabajo solo se describen los tres algoritmos básicos, hay otros algoritmos que se deducen a partir de los ya mencionados.

Entre estos algoritmos que no analizaremos podemos encontrar el D*, hermano del A* pero que a diferencia de este empieza en la casilla de destino. Podemos mencionar también el RRT bidireccional, con el mismo fundamento que el RRT básico pero que inicia la búsqueda de la ruta en el origen y el destino simultáneamente. En el libro "*Introduction to Autonomous Robots*", de Nikolaus Correll, se encuentran desarrollados estos dos algoritmos junto con otros de mayor complejidad.

Lo primero que tendremos que hacer para empezar a desarrollar cada uno de los algoritmos será entender los datos con los que trabajamos. Tendremos que proporcionarle al algoritmo unos datos básicos de entrada. Estos serán:

- Un mapa o gráfico con las celdas numeradas
- Posición inicial o de origen
- Posición final que deseamos alcanzar

El resultado o datos de salida que nos devolverá el algoritmo será una ruta entre los dos puntos deseados. Esta ruta o camino consistirá pues es una serie de celdas contiguas que irán recorriendo el mapa hasta alcanzar nuestro objetivo.

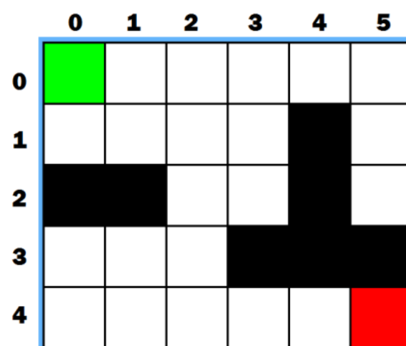


Figura 2.1 – Situación inicial

2.1. Algoritmo Dijkstra

Debido a su modo de funcionamiento, el algoritmo de búsqueda Dijkstra es el más básico de los tres que veremos por lo que será el primero en ser explicado. Éste permite priorizar direcciones a explorar, pudiendo así encontrarnos con dos modalidades dentro de dicho algoritmo. La primera vertiente es la más simple y común y consiste en la exploración por igual de todas las direcciones. Es de gran utilidad para la generación y el análisis de mapas.

Sin embargo, hay una variante del algoritmo que nos permite priorizar unas direcciones frente a otras. En lugar de explorar en todas las direcciones por igual, profundiza más en aquellas que tienen un coste menor. Mediante la asignación de costes para cada movimiento, podemos modificar el comportamiento del algoritmo, evitando por ejemplo el análisis en según qué direcciones nos interese.

Como el entorno en el que trabajaremos no presenta ninguna peculiaridad respecto a los movimientos del móvil, implementaremos el algoritmo Dijkstra de forma que busque por igual en todas las direcciones. Es lo más claro y lógico para los entornos de trabajo en los que trabajaremos, aunque siempre tendremos ahí la posibilidad de modificarlo en función de nuestras necesidades.

Analicemos pues ahora el desarrollo del algoritmo Dijkstra. Una vez tenemos los tres elementos básicos, nos encontramos en la situación inicial. Para entenderlo de una forma más clara, iremos desarrollando la búsqueda sobre un esquema de ejemplo que hemos hecho aleatoriamente (Figura 2.1).

La idea principal tanto en este algoritmo como en el A* que veremos más adelante es la de ir expandiendo la frontera. Esto es como si desde la posición inicial surgiera una especie de anillo que se expandiera analizando todo el entorno.

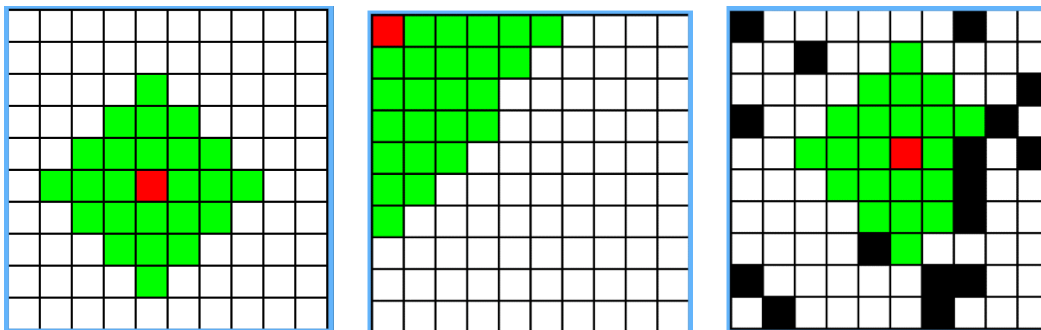


Figura 2.1.1

Descrita la situación inicial inicializaremos una serie de variables que nos ayudarán a ir buscando la ruta más corta hasta el punto de destino. Desarrollando el ejemplo iremos viendo la utilidad de cada una de ellas.

La primera variable que declararemos es la tabla "***visita***". Esta es una tabla o matriz de las mismas dimensiones que el mapa con todas sus casillas a 0. Su función es la de recoger todas las celdas visitadas a lo largo del algoritmo, por lo que cada vez que pasemos por una celda, la matriz visita recogerá este paso y en la respectiva posición de la tabla sustituirá el 0 por un 1.

También declararemos la tabla "***paso***". De las mismas dimensiones que el mapa, va recogiendo en qué paso del algoritmo hemos alcanzado cada celda. En un principio tiene todas sus posiciones a -1, pero conforme vamos avanzando en el algoritmo nos indica cuántos pasos tardamos en llegar a cada posición del mapa. Mantiene el -1 en aquellas casillas por las que no llegamos a pasar.

Otra variable a declarar será la matriz "***acción***". Esta se irá rellenando en función del movimiento realizado desde una celda hacia la que va a continuación. Por ejemplo, si de la casilla [0, 0] quisiera pasar a [0, 1], tendría que hacer un movimiento hacia la derecha. En función de una asignación previa, en la casilla [0, 0] se guardará un número del 1 al 4 (hay 4 movimientos) que represente el movimiento "derecha".

Una vez tenemos declaradas estas 3 tablas, pasamos a inicializar una lista en la que iré guardando las celdas que se van analizando. La lista se llamará "***lista***", valga la redundancia, y en ella se irán realizando todas las iteraciones en la búsqueda de la ruta deseada. Además de las coordenadas de cada celda, guardará el coste de llegar a la misma y se irá ordenando para expandir siempre los nodos cuyo coste sea menor.

El funcionamiento es el siguiente: mientras no hayamos alcanzado el nodo destino, vamos analizando los nodos cuyo coste sea menor. Para ello el nodo con coste menor sale de la lista y entran aquellos a los que puedo acceder desde ese nodo que sale. Para ver más claro cómo funciona empezaremos a resolver el algoritmo con la tabla del ejemplo. Tendremos el mapa, la coordenada de origen y la de destino.

Mapa = [[0 0 0 0 0 0],
[0 0 0 0 1 0],
[1 1 0 0 1 0],
[0 0 0 1 1 1],
[0 0 0 0 0 0]]

Inicio = [0 , 0]
Fin = [4 , 5]

Para el ejemplo seguiremos la siguiente notación:

Lista = [coordenada x, coordenada y] - Coste

La lista que iremos rellenando tendrá en un estado inicial el nodo origen con su respectivo coste. Por lo tanto, al principio de todo:

Lista = [0 , 0] – 0

A lo largo del algoritmo comprobamos si el nodo que tenemos en ***lista*** es el nodo final. Para ello previamente se declara una bandera o variable booleana inicializada a "false". Tras comprobar que no es el nodo destino, lo que se hace a continuación es sacar ese nodo de la lista y analizar sus posibles sucesores. En el caso del ejemplo nos quedaría:

Lista = ~~[0 , 0] – 0~~
[0 , 1] – 1 [1 , 0] – 1

Para ir rellenado la lista y expandiendo el análisis, siempre analizaremos el nodo con un coste menor. Sin embargo, muchas veces tenemos una situación en la que hay varias celdas con el mismo coste. Da igual cual de las dos expandamos, no variará el resultado final.

A continuación se ve como tras expandir el nodo [0, 1], este sale de la lista y entran el [1, 1] y el [0, 2], ambos con el coste incrementado.

Lista = ~~[0 , 0] – 0~~
~~[0 , 1] – 1~~ [1 , 0] – 1
[0 , 2] – 2 [1 , 1] – 2

El siguiente nodo en salir de la lista sería el [1, 0], ya que es el que tiene un coste menor, pero sin embargo las celdas que lo rodean ya han sido visitadas o directamente son obstáculos. Por lo tanto, al no poder expandirse éste sale de la lista y pasamos a analizar uno de los dos siguientes posibles nodos.

Lista = ~~[0 , 0] – 0~~
~~[0 , 1] – 1~~ ~~[1 , 0] – 1~~
~~[0 , 2] – 2~~ [1 , 1] – 2
[0 , 3] – 3 [1 , 2] – 3

A la vez que vamos rellenando la lista, las tablas declaradas inicialmente también se van modificando. Para verlo más claramente analicemos el estado de las tablas ***paso*** y ***visita*** a estas alturas del algoritmo, las cuales presentarían la situación siguiente:

1	1	1	1	0	0
1	1	1	0		0
		0	0		0
0	0	0			
0	0	0	0	0	0

Figura 2.1.2.- Tabla visita

0	1	3	5		
2	4	6			

Figura 2.1.3 – Tabla paso

Esquemáticamente vemos cómo se va rellenando el mapa tras 6 iteraciones. A continuación se aprecia el estado de ambas tablas tras 15 iteraciones y una vez alcanza el nodo destino tras 22 repeticiones.

0	1	3	5	7	10
2	4	6	8		13
		9	11		
	14	12			
		15			

Figura 2.1.4.- Tabla paso

0	1	3	5	7	10
2	4	6	8		13
		9	11		16
17	14	12			
20	18	15	19	21	22

Figura 2.1.5 – Tabla paso

Así, continuaríamos desarrollando el algoritmo hasta alcanzar el nodo destino [4, 5]. En la tabla ***paso***, todos los nodos que no han sido visitados (en el ejemplo los obstáculos negros ya que el resto de celdas sí se visitan) tendrían valor "-1". Cuando un nodo sale de la lista, entran otros cuyo coste se incrementa en 1 y a la vez vamos almacenando en pasos el número de iteraciones necesarias para alcanzar cada nodo. En este caso, vemos que para alcanzar el nodo destino han sido necesarias 22 iteraciones.

Analicemos ahora una situación distinta, en la que no es posible alcanzar el nodo final debido a los obstáculos encontrados. Para ello, cambiamos el mapa del ejemplo de modo que el camino que antes encontramos en este caso estuviera bloqueado.

La tabla ***paso*** quedaría como se ve en la figura, la lista se vaciaría sin haber encontrado el nodo destino y tendría que aparecernos una advertencia del problema. Para ello, se programa una bandera de aviso de forma que en el momento que ***lista*** está vacía y no se ha alcanzado la celda deseada, esta bandera nos avisa de que es imposible hallar una ruta hacia el destino. En el ejemplo, la última casilla en salir de lista sería [1, 5] en el paso 10, y tras esto no podría seguir expandiéndose el algoritmo hacia ninguna dirección.

0	1	3	5	7	9
2	4	6	8	-1	10
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1

Figura 2.1.6

Volviendo al caso del principio en el que sí que se podía resolver el algoritmo de búsqueda, pasaremos ahora a analizar los movimientos a realizar por el sujeto o robot móvil. Para ello en un inicio ya creamos la tabla ***acción***, que nos irá rellenando los movimientos necesarios a realizar en cada celda del camino.

En el caso de las 2 dimensiones, los pasos posibles a realizar son 4: izquierda, derecha, arriba y abajo. La tabla ***acción*** nos ha ido rellenando a lo largo de toda la búsqueda los movimientos en cada celda para acceder a la siguiente.

Debido a una asignación que hicimos al principio, tenemos que cada número significa:

-1	3	3	3	3	3
2	2	2	2	-1	2
-1	-1	2	2	-1	2
1	1	2	-1	-1	-1
2	2	2	3	3	3

Figura 2.1.7. – Tabla acción

1: Paso hacia la izquierda

2: Paso hacia abajo

3: Paso a la derecha

4: Paso hacia arriba

Esta asignación podría haberse hecho de cualquier otra manera. Lo que nos interesa es tener en cada celda asignado el movimiento respectivo que se hizo previamente para ser alcanza.

Teniendo esto claro, creamos una tabla llamada "***esquema***" que nos ayudará a entender cómo conseguimos el camino definitivo. La variable ***esquema*** es una tabla, de las dimensiones de nuestro mapa, con todos sus valores a 0 y que iremos rellenando desde el final al principio para hallar la ruta definitiva.

La última variable que crearemos será la principal y la razón por la que hemos realizado todo el algoritmo. Esta variable se llamará "***ruta***" y, como su propio nombre indica, devolverá todo el camino a recorrer desde el nodo

inicial hasta el final.

Para ver cómo se va creando y rellenando el camino vemos cómo el primer paso a realizar es añadirle la celda destino:

Ruta = [[4, 5]]

Ayudándonos de la tabla antes comentada ***acción***, vemos que al tener la celda [4, 5] un 3 significa que fue alcanzada tras un paso a la derecha, por lo que la casilla previa tuvo que ser [4, 4]. De la misma forma, vemos que la previa a esta tuvo que ser [4, 3], y así sucesivamente hasta llegar al origen. Siguiendo esta lógica vamos rellenando la lista ruta de la siguiente manera:

Ruta = [[4, 5], [4, 4], [4, 3], [4, 2], [3, 2], [2, 2], [1, 2], [0, 2], [0, 1], [0, 0]]

Por último le tendríamos que dar la vuelta a la lista ***ruta*** y nos quedaría el camino ordenado del todo. Para hacernos una idea veamos gráficamente cómo quedaría la tabla ***esquema***. Esta tabla, aparte de guardar los movimientos, nos sirve para visualmente apreciar cómo se va recorriendo el laberinto.

Aunque no es estrictamente necesaria, sí que nos ayudará a ver cómo se va creando la ruta. La tabla ***esquema*** sólo nos servirá para apreciar visualmente cómo se recorre el laberinto. No nos sirve para realizar cálculos de búsqueda por lo que nos la podríamos ahorrar, aunque a la hora de implementar el algoritmo y depurar también puedes ser de gran utilidad.

>	>	v			
		v			
		v			
		v			
		>	>	>	*

Figura 2.1.8. – Tabla esquema

Una vez descrito y entendido nuestro primer algoritmo de búsqueda Dijkstra, pasamos al siguiente método de exploración. Se trata de un mecanismo de funcionamiento muy parecido al de Dijkstra pero con algunos detalles más complejos.

2.2. Algoritmo A*

Podemos definir el algoritmo A* como una modificación del algoritmo Dijkstra optimizado para un destino concreto. Este algoritmo fue presentado por primera vez en 1968 por Peter Hart, Nils Nilsson y Bertram Raphael. Mientras Dijkstra puede encontrar rutas hacia cualquier dirección, el A* va buscando la ruta hacia una localización específica. Prioriza así las direciones que parecen ir alcanzando el destino especificado.

El algoritmo A* es una variante por tanto del Dijkstra que hemos desarrollado previamente, y su mecanismo también consiste en ir expandiendo gradualmente una lista con las celdas que vamos analizando. Para ver la diferencia entre ambos algoritmos a primera vista, creamos un nuevo y sencillo escenario en el que veremos el funcionamiento del algoritmo en cada caso.

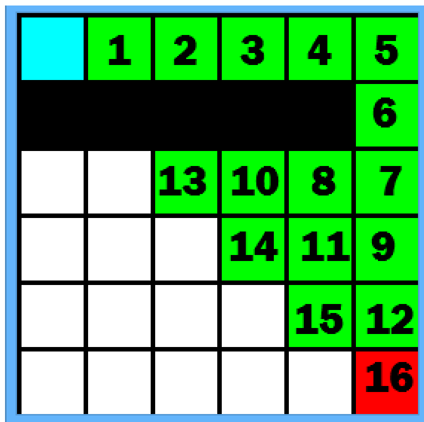


Figura 2.2.1 – Tabla paso con Dijkstra

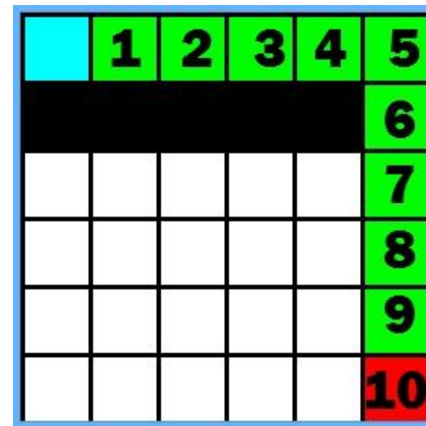


Figura 2.2.2 – Tabla paso con A*

En este ejemplo se aprecia gráficamente cómo funciona el algoritmo A*. Una vez llegamos a la esquina superior derecha, el algoritmo Dijkstra empieza a buscar los caminos posibles dentro del escenario en todas las direcciones hasta que alcanza el destino en el paso 16. Esto quiere decir, recordando la función que la tabla ***paso***, que hemos tenido que realizar 16 iteraciones dentro del algoritmo.

En el caso del A* sin embargo, vemos que con 10 iteraciones nos sirve para llegar al destino deseado. Una vez esquiva la primera fila de obstáculos y llega a la esquina superior derecha, intuye que su objetivo está hacia abajo y por tanto se expande en esa dirección. El A* sabe de alguna manera que a priori cualquier otra ruta hacia el nodo final será más larga, por lo que decide explorar hacia abajo directamente.

Cuanto más grande sea el escenario, más pasos puede ahorrarnos el A* respecto al Dijkstra y por lo tanto más eficiente será. Supongamos ahora un laberinto más amplio, en el que además aparece un obstáculo a evitar antes de alcanzar la celda destino.

En el siguiente ejemplo vemos cómo en el caso del Dijkstra realiza 61 iteraciones por 25 en el caso del A*, siendo por tanto este último mucho más práctico. Cuando se encuentra con el obstáculo lo evita y se sigue expandiendo, siempre buscando en la dirección adecuada. Así, con el mínimo esfuerzo necesario consigue avanzar hacia el destino. Este es el principio básico del algoritmo A*, para el cual necesita apoyarse en la función que veremos a continuación.

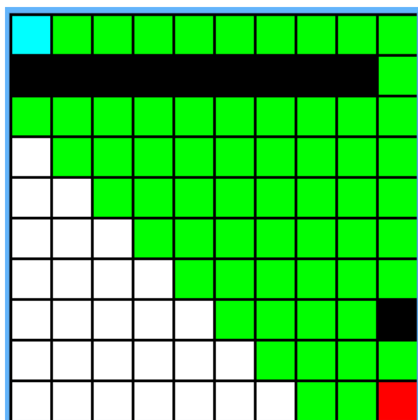


Figura 2.2.3 – Algoritmo Dijkstra

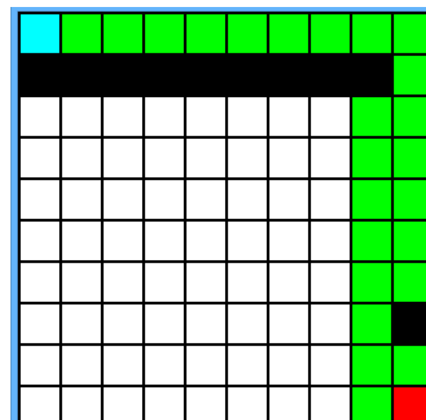


Figura 2.2.4 – Algoritmo A*

La herramienta fundamental empleada por el algoritmo es la función heurística, la cual tiene que ser previamente preparada. Esta función nos devuelve la distancia que hay desde cada punto del mapa hasta el destino, sin tener en cuenta los obstáculos. Por lo tanto, para cada celda de la tabla tendrá asignado un valor que nos indicará el número de pasos hasta una celda destino especificada.

Función heurística: **h(x, y) $\leq$ distancia hasta el destino desde la coordenada (x, y)**

9	8	7	6	5	4
8	7	6	5	4	3
7	6	5	4	3	2
6	5	4	3	2	1
5	4	3	2	1	

5	4	3	2	3	2
4	3	2	1	2	3
3	2	1		1	2
4	3	2	1	2	3
5	4	3	2	3	4

Figura 2.2.5 – Ejemplos de la función heurística en distintos escenarios

Implementar la función heurística es bastante sencillo. En nuestro caso, sabiendo que en 2D solo podemos realizar 4 movimientos (arriba, abajo, izquierda, derecha) es fácil ver cuántos pasos tendría que llevar a cabo nuestro sujeto para llegar al destino (representado en el esquema por una celda roja) si no hubiera obstáculos. Dicha tabla podemos generarla automáticamente en unas pocas líneas de código, tanto en 2D como en 3D.

La clave de esta función es que el valor devuelto no tiene por qué ser exacto, pues su función se limita a ayudarnos a elegir la siguiente casilla a analizar en nuestro análisis del mapa, devolviendo el número de pasos que como mínimo y en el mejor de los casos tendremos que recorrer.

Una vez entendida la utilidad de esta función, la modificación clave en nuestro algoritmo es muy simple. Todas las tablas que declaramos en el Dijkstra ***(visita, acción, paso, esquema*** etc.) se declaran de la misma manera. Además, le añadimos la ya descrita tabla de la función heurística.

Con esto aclarado, lo único que cambia es la la variable ***lista***. En ella, antes almacenábamos las coordenadas y el coste de llegar a cada casilla. Ahora, lista tiene dos campos más. A las coordenadas y el coste que teníamos antes, le añadimos el valor de la función heurística en esa celda y el coste toal, el cual es la suma del coste más la propia función heurística.

Coste total = coste + función heurística (x, y)

Aclaradas estas modificaciones, pasemos a la tabla del ejemplo visto previamente el cual éramos capaces de resolver en solo 10 iteraciones. No olvidemos que el mecanismo para rellenerar las tablas ***paso***, ***visita*** y ***acción*** sigue siendo el mismo. Sin embargo, una vez llegamos a la esquina superior derecha el algoritmo empieza a variar su comportamiento. Detengámonos en el paso 6:

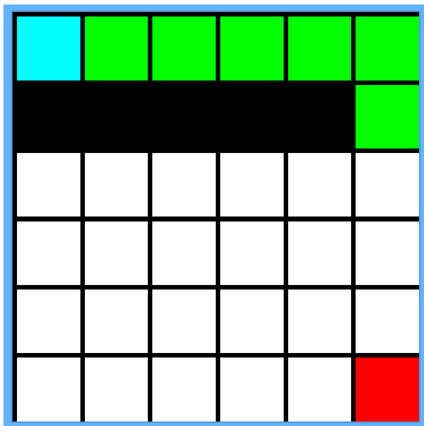


Figura 2.2.6 – Iteración 6 del A*

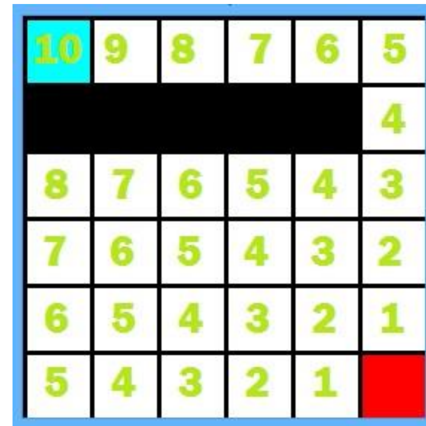


Figura 2.2.7 – Función heurística del escenario

Para ir viendo cómo se rellena la lista emplearemos la siguiente notación:

Lista = [coste, f.heurística, coordenadas] – coste total

A continuación se ve cómo van entrando y saliendo nodos (al ser tachados) de la lista:

Lista = ~~[0, 10, 0, 0] – 10~~
~~[1, 9, 0, 1] – 10~~
~~[2, 8, 0, 2] – 10~~
~~[3, 7, 0, 3] – 10~~
~~[4, 6, 0, 4] – 10~~
~~[5, 5, 0, 5] – 10~~
[6, 4, 1, 5] – 10

Si nos vamos a la celda [1, 5], vemos que en este punto del algoritmo su coste es de 6, es decir, los pasos que nos ha llevado llegar hasta ahí. Si miramos la función heurística, vemos que en ese punto la tabla devuelve un valor de 4, por lo que sumando ambos factores tenemos un coste total de 10. Siguiendo con el mismo procedimiento, en el siguiente paso, nos encontraremos con el nodo con coordenadas [2, 5]

Lista = ~~[6, 4, 1, 5] – 10~~

[7, 3, 2, 5] – 10

En este punto tendremos que analizar las dos siguientes posibles nodos:

Lista = ~~[7, 3, 2, 5] – 10~~

[8, 4, 2, 4] – 12 vs [8, 2, 3, 5] – 10

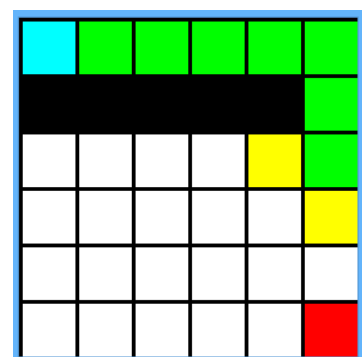


Figura 2.2.8

Vemos que mientras ambos tienen el mismo coste, su función heurística en cada punto es 2 y 4 respectivamente, por lo que a priori la casilla [3, 5] estará dos pasos más cerca que la casilla [2, 4]. Esto significa que A* se expandirá en esta dirección, pues su coste total es menor. Tras avanzar hacia el siguiente nodo nos encontramos con la misma situación de antes:

Lista = ~~[8, 2, 3, 5] – 10~~
[8, 4, 2, 4] – 12 [9, 3, 3, 4] – 12 [9, 1, 4, 5] – 10

Como a lo largo de todo el algoritmo, el coste se ha incrementado en 1, pero otra vez y por el mismo motivo de antes, el coste total debido a la función heurística varía en 2. Seguimos por tanto hacia abajo hasta que por fin alcanzamos el punto de destino, en la casilla con coordenadas [5, 5].

Una vez que alcanzamos el destino, el procedimiento a seguir es exactamente el mismo que con el Dijkstra. No olvidemos que a lo largo de todo el algoritmo, cada vez que visitábamos una celda, rellenábamos la correspondiente posición de la tabla ***acción*** indicando el movimiento realizado para pasar a la celda siguiente. Así, creamos en este caso también la variable ***ruta*** y la vamos rellenando desde la última posición hasta el punto de partida. Posteriormente le damos la vuelta y ya tenemos el dato de salida que esperábamos, que no es otro que una serie de celdas contiguas indicándonos el camino más corto desde la posición inical hasta la final.

Explicado y entendido el A*, pasaremos a explicar el último de los tres algoritmos a implementar: el RRT. Su patrón de comportamiento no tendrá nada que ver con los dos algoritmos de búsqueda vistos hasta ahora.

2.3. Algoritmo RRT

El algoritmo RRT, abreviatura de "*Rapidly Exploring Random Trees*", fue desarrollado recientemente y es un algoritmo con un funcionamiento completamente distinto al Dijkstra y A* vistos hasta ahora.

El RRT aborda el problema de la búsqueda de caminos usando aproximaciones aleatorias que permiten una rápida exploración del área con refinamiento iterativo. La idea básica es seleccionar un punto aleatorio dentro del escenario y conectarlo al árbol de búsqueda que vamos formando. Los puntos aleatorios van siendo conectados a las celdas más cercanas que ya han sido analizadas en cada caso. Se forman así lo que coloquialmente llamamos como "ramas" a la vez que se va expandiendo el árbol aleatorio de búsqueda.

Este algoritmo es un tanto imprevisible ya que, al depender de aproximaciones aleatorias, su grado de efectividad puede variar en unas situaciones y en otras. A diferencia de los dos algoritmos vistos hasta ahora, el RRT no garantiza coger el camino más corto desde un punto hasta otro. Si esto no es así, ¿por qué debería interesarnos este algoritmo?

La respuesta a esta pregunta es la velocidad y el número de cálculos necesarios para conseguir alcanzar el destino. Cuando estamos en escenarios de grandes dimensiones, puede resultar muy costoso en cuanto a cálculos tener que analizar todas las celdas que parten dese el origen hasta que se llega al destino. Por ello, las aproximaciones aleatorias y el modo de funcionamiento del algoritmo RRT hace que en muchos casos este método sea el más adecuado.

Como es lógico, no tiene sentido emplear el RRT en los escenarios de dimensiones 6x5 o 10x10 que hemos visto para los ejemplos del Dijkstra y el A*. Para analizar el algoritmo RRT de forma completa y poder ver sus ventajas, a partir de ahora utilizaremos escenarios de mayores dimensiones.

Para entender al detalle este algoritmo, podemos recurrir al tutorial de Steve M. LaValle, "*Motion Planning, Part I: The Essentials*". En este artículo, publicado en 2011 por la *IEEE Robotics & Automation Magazine,* vemos cómo se formula el algoritmo, así como todo el fundamento matemático que tiene detrás y las innumerables aplicaciones en las que puede ser empleado.

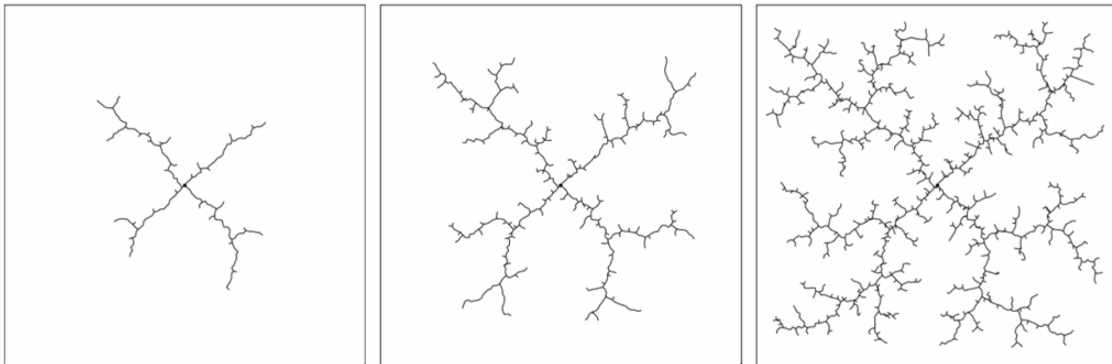


Figura 2.3.1 – Desarrollo del algoritmo RTT conforme va aumentando el número de iteraciones

Adaptándolo a nuestro contexto de trabajo, empezaremos por el final para entender los resultados que este algoritmo puede conseguir en el tipo de escenarios que estamos trabajando. Creamos escenarios al azar de dimensiones 40x40, poniendo el mismo origen y destino en ambos casos y la misma densidad de obstáculos. El resultado es el siguiente:

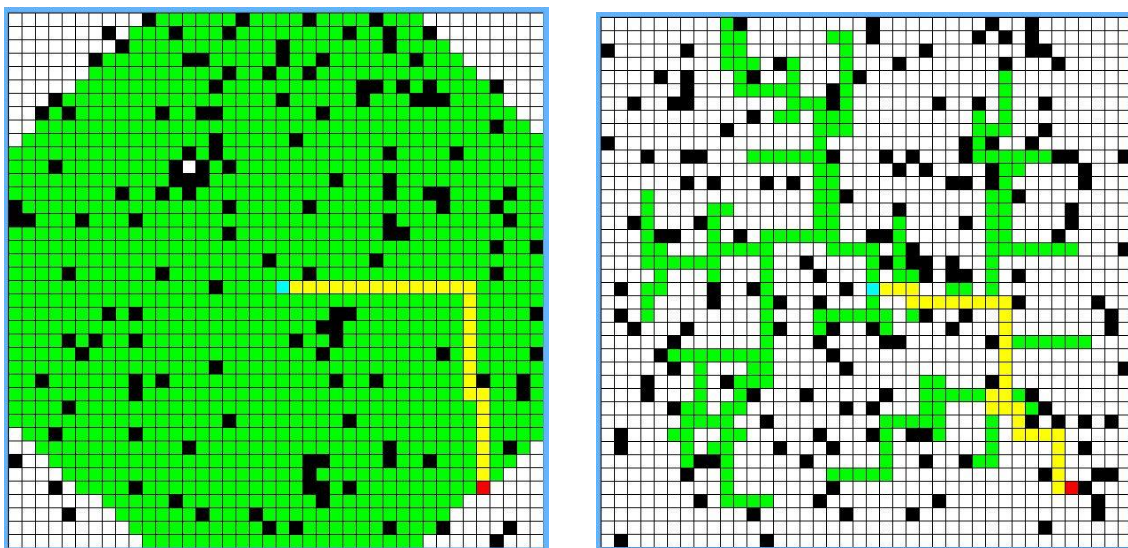


Figura 2.3.2 – Dijkstra vs RRT

Mientras que en el Dijkstra hemos necesitado 1279 iteraciones de un total de 1600 celdas, en el A* hemos conseguido alcanzar el destino con tan solo 303 iteraciones. Podríamos decir que en este caso el algoritmo RRT ha sido 4 veces más eficiente que el Dijkstra. No obstante, el azar juega un papel determinante cuando aplicamos este algoritmo, por lo que podría haberse dado el caso de que aplicando el RRT en el mismo contexto necesitaramos muchas más iteraciones para alcancar el destino.

Teniendo una idea básica de cómo funciona el algoritmo, vamos a desglosarlo y a desarrollarlo paso a paso al igual que hicimos con el Dijkstra y el A*. En primer lugar se define el escenario, que al ser de dimensiones más grandes siempre se realizará mediante funciones aleatorias. Para ello generamos matrices, de las dimensiones que especifiquemos, de "unos" y "ceros" aleatorios. Podemos ir cambiando la densidad de obstáculos, forzando a que esas funciones aleatorias que van rellenando la matriz tiendan a poner más o menos "unos" dentro del escenario.

A continuación declaramos las tablas de ***visita***, ***acción***, ***paso*** y ***esquema***. Recordemos que ***visita*** guardaba las celdas visitadas, ***paso*** la iteración en la que se llegaba a esa celda, ***acción*** el movimiento en cada celda y ***esquema*** era una tabla auxiliar de ayuda.

La siguiente variable que declaramos es nueva en este algoritmo y se llamará "***nodos***". A medida que se expande nuestro árbol, se van creando nuevos vértices o casillas donde acaban esas "ramas". Para la siguiente expansión, partimos de estos vértices hasta llegar a la casilla deseada y creamos un nuevo punto de partida dentro del laberinto. Estos vértices o celdas que progresivamente van formando el árbol aleatorio los iremos guardando en la lista que hemos llamado ***nodos***.

Una vez declaradas las variables empezamos la búsqueda de la ruta. En cada iteración, el algoritmo genera una celda al azar dentro del escenario utilizando la función ya implementada ***random***. Si esta celda es un obstáculo o ya ha sido visitada previamente, vuelve a generar otra celda aleatoria que no cumpla ninguna de las dos condiciones anteriores. Una vez tenemos esta celda aleatoria, recorremos la lista nodos y vemos cual es la casilla más cercana a la nueva celda que queremos incorporar.

Entra en escena ahora otra variable nueva para este algoritmo: "***alcance***". Esta variable sirve para especificar cómo de largas queremos que sean las ramas o nuevos caminos que va generando nuestro árbol. ***Alcance*** influirá por tanto en la forma del árbol, y es una variable que podremos variar en función de las dimensiones del escenario en el que empleemos el algoritmo.

Cuando la función ***random*** nos devuelve una nueva celda aleatoria, no la añadimos directamente a la lista de nodos. Para ello, tienen que estar a una distancia igual o menor que ***alcance*** de alguno de los extremos que forman el árbol. Definimos para este propósito dos sencillas funciones:

- ***Distancia:*** Se le pasan como parámetros dos celdas del escenario. Nos devuelve la distancia en pasos de una celda a otra, por lo que siempre devolverá un número entero.

- ***Siguientepaso:*** Se le entregan como parámetros un vértice del árbol y la celda aleatoria que queremos conectar. Si la distancia de la celda aleatoria al vértice del árbol es menor o igual que alcance, nos devuelve esa misma celda aleatoria. En caso contrario, nos devuelve aquella celda dentro de nuestro alcance que está más próxima a la casilla aleatoria que le hemos pasado como parámetro.

Estas dos funciones interactúan entre sí a la hora de crear nuevas ramas dentro del árbol aleatorio. Para entender cómo se va creando el árbol y la labor de estas dos funciones, partimos desde el principio en un escenario aleatorio de dimensiones 30x30. Empezamos en la esquina superior izquierda y queremos alcanzar la casilla destino, colocada aleatoriamente en la posición [25 , 25]. La longitud máxima de cada rama será 6 posiciones, aunque es un dato que se podría cambiar.

La primera celda aleatoria que genera el código es la [17 ,16]. Ayudándonos de la representación gráfica, tenemos que:

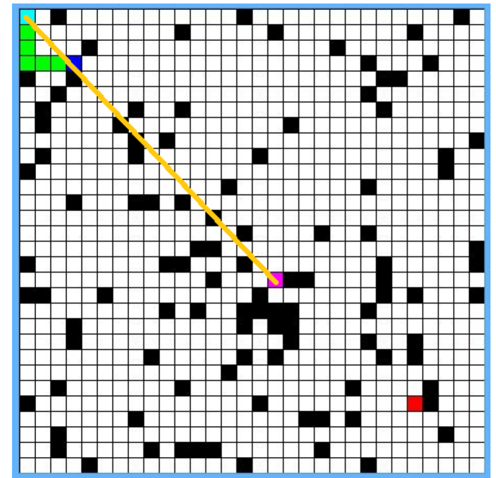


Figura 2.3.3

distancia ([0 , 0] , [17 , 16]) = 33

siguientepaso ([0 , 0], [17 , 16]) = [3 , 3]

Como está fuera del alcance, que en este caso le hemos asignado valor 6, la función siguientepaso nos devuelve la celda [3 , 3]. Ésta es la casilla más próxima a la que podemos llegar en la dirección aleatoria deseada. Por su parte, la lista ***nodos*** queda de la siguiente manera:

Nodos = [[0 , 0] , [3 , 3]]

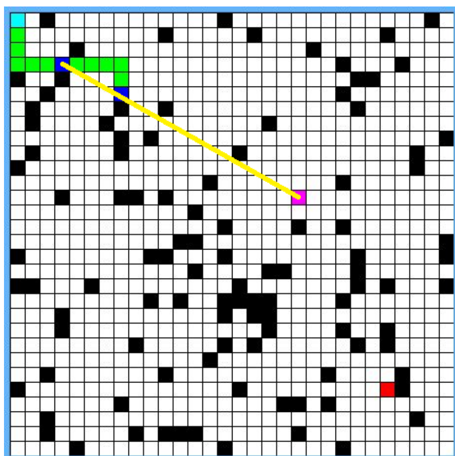


Figura 2.3.4

En la siguiente iteración, la segunda posición aleatoria que nos sale es la [12, 19]. Dentro de ***nodos***, la celda más cercana es [3 , 3], por lo que realizamos los mismos pasos que antes:

distancia ([3 , 3] , [12 , 19]) = 25

siguientepaso ([3 , 3], [12 , 19]) = [5 , 7]

Nodos = [[0 , 0] , [3 , 3] , [5 , 7]]

De esta forma vamos expandiendo el árbol de forma aleatoria en todas las direcciones. A medida que va creciendo el árbol, más nos vamos acercando a la celda destino. Sin embargo, al depender de funciones aleatorias no sabemos con certeza cuánto tardará en alcanzarlo. Veamos cómo queda el árbol de búsqueda después de haberle añadido ya 15 vértices o nodos y tras haber cconseguido llegar al destino:

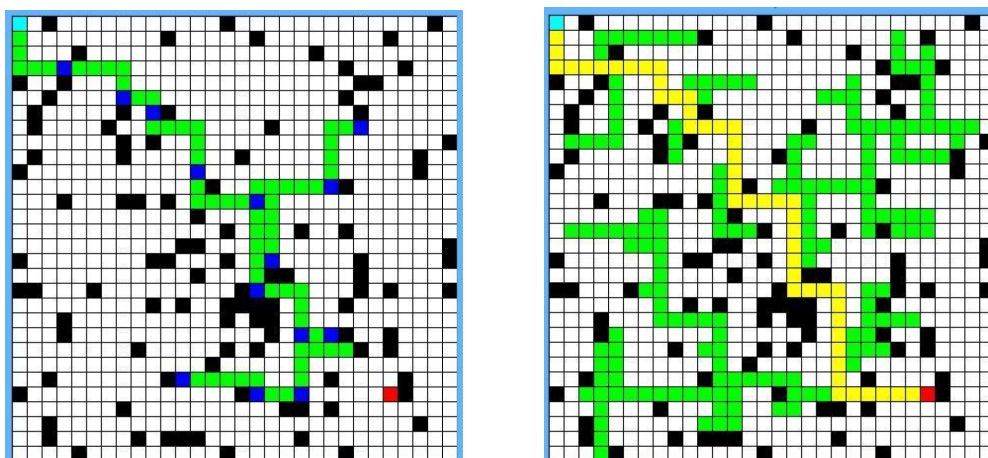


Figura 2.3.5 – RRT tras 15 ramas y tras haber alcanzado el destino deseado

Una vez sabemos el mecanismo según el cual se expande nuestro árbol de exploración, nos queda por definir la función más compleja del algoritmo. La llamaremos "***rama***", y será la encargada de expandir el árbol desde un vértice hasta el siguiente.

Como parámetros se le pasarán dos celdas, la que ya está en ***nodos*** y la que queremos añadir, es decir, la que nos ha devuelto la función ***siguientepaso***. Ésta función nos devolverá el nuevo vértice que añadiremos a la lista ***nodos*** y una bandera que nos indicará si por el camino hemos encontrado o no el destino deseado.

Rama será por tanto la encargada de ir rellenando las tablas de ***visita***, ***paso***, ***esquema*** y ***acción***. Además, tiene que tener en cuenta varios aspectos de gran relevancia durante la expansión del árbol.

Se puede dar el caso de que desde un vértice no se pueda acceder a otro a la primera, por lo que esta función se encarga de analizar todos los caminos posibles. En primer lugar se intenta llegar a la celda deseada moviéndome en vertical y después en horizontal. Si hay un obstáculo en medio esto no es posible, por lo que lo intenta por segunda vez pero realizando primero el movimiento horizontal y después el vertical. Es lo que sucede en la figura 2.3.5:

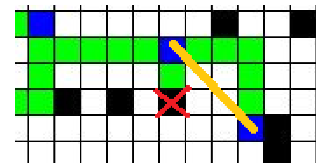


Figura 2.3.5

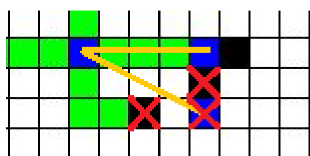


Figura 2.3.6

También se puede dar una situación en la que desde una celda no se pueda accer a otra por ninguno de los dos caminos. En ese caso, tras haber probado por ambos caminos, la función ***rama*** devolvería la posición más lejana que hayamos podido alcanzar en la expansión, y por lo tanto sería el vértice que se añade al árbol.

Explicadas todas las funciones a implementar, resumimos en 4 los pasos a realizar por el algoritmo de búsqueda RRT:

1º. Seleccionamos posición al azar dentro del escenario y comprobamos que no sea un obstáculo ni haya sido visitado previamente.

2º. Recorremos la lista nodos en la que tenemos los extremos de nuestro árbol aleatorio y vemos cuál es el más cercano.

3º. Ayudándono de las funciones ***siguientepaso*** y ***distancia*** determinamos cuál es la nueva celda a añadir a nuestro árbol de búsqueda.

4º. Con la función ***rama*** realizamos la expansión hasta esa nueva celda o hasta lo más cercana a ella que sea posible en caso de haber obstáculo por el camino.

2.4. Rendimiento de los algoritmos en distintos escenarios

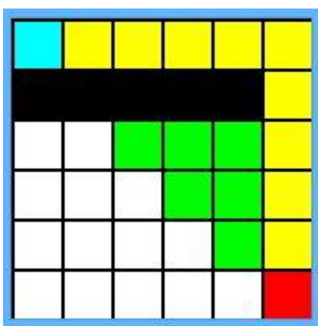
Tras haber analizado los tres algoritmos que implementaremos en las plataformas de desarrollo tridimensionales, llegamos a un punto en el que nos preguntamos qué algoritmo de los tres es el mejor. La respuesta a esta pregunta es relativa, ya que según el contexto y el tipo de escenario en el que nos encontremos un algoritmo puede tener ventajas respecto al otro y viceversa.

Como veremos a lo largo de este apartado, son muchos los fatores que influyen en la eficiencia de nuestros algoritmos de búsqueda. Para sacar nuestras propias conclusiones, empezaremos analizando escenarios de pequeñas dimesiones que hemos ido utilizando para la explicación de los algoritmos. Posteriormente les iremos cambiando parámetro tales como las casillas de origen y destino o las dimensiones del mismo.

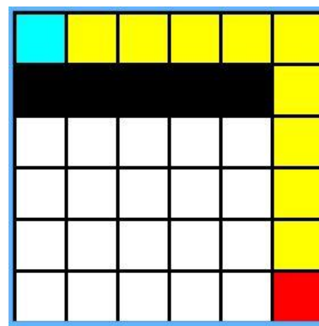
En cada ejemplo se especifiarán las dimensiones del escenario y las iteraciones necesarias llevadas a cabo por cada algoritmo para alcanzar su destino. Lo que buscamos en cualquier caso es que este número de iteraciones realizadas para alcanzar la celda de destino sea lo menor posible. Cuantas menos veces se ejecute el algoritmo significa que es mas eifciente, lo que traducido a aplicaciones dentro de entornos reales puede suponer menor tiempo de resolución, menor coste computacional etc.

No olvidemos que el RRT depende de aproximaciones aleatorias, por lo que el número de iteraciones nos sirve sólo como referencia para tener una idea, sabiendo que podría variar en otras ejecuciones. También hemos de tener en cuenta que el A* es una versión mejorada del Dijkstra, por lo que siempre que podamos elegiremos este segundo. Sin embargo, para poder utilizar el algoritmo A* deberíamos implementar la función heurística y esto no siempre es posible.

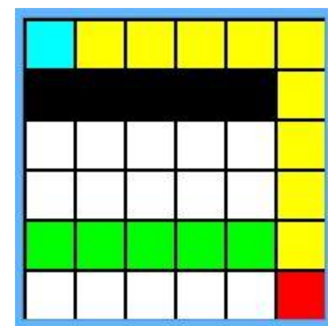
- Escenario con 36 celdas. Empezamos con un escenario básico, típico de los que se emplean para explicar el funcionamiento de los distinos algoritmos.



Dijkstra: 17

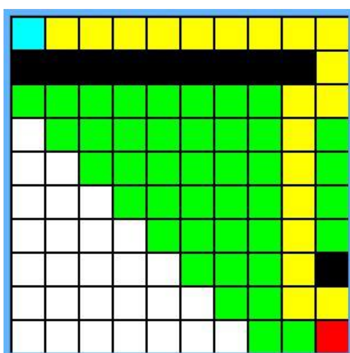


A*: 11

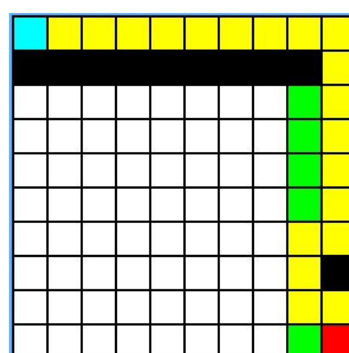


RRT: 16

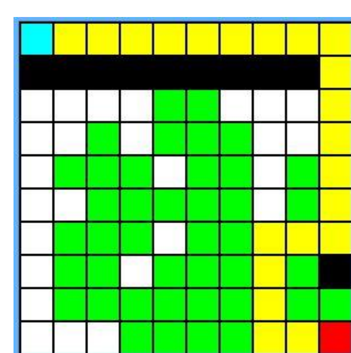
- Escenario con 100 celdas. Conforme más grande vamos haciendo el escenario, más pasos nos ahorra el A* respecto al Dijkstra. Por su parte, el RRT no parece una buena opción respecto a los otros dos en este tipo de escenarios.



Dijkstra: 62

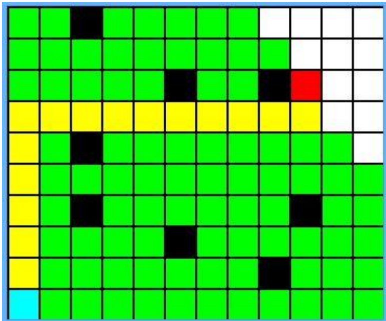


A*: 26

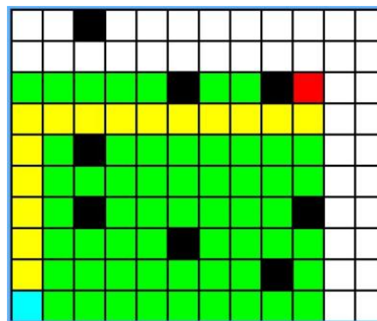


RRT: 63

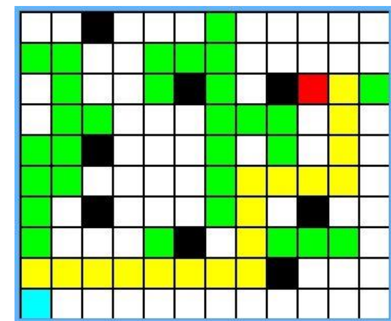
- Escenario con 120 celdas. Siguiendo con escenarios del mismo orden de magnitud, variamos la ubicación de la celda destino. Hasta ahora, origen y destino habían estado en esquinas opuestas. En esta ocasión ponemos la celda de destino en una posición al azar. Comprobamos que en esta situación el RRT puede darnos mejores resultados que el A*.



Dijsktra: 100

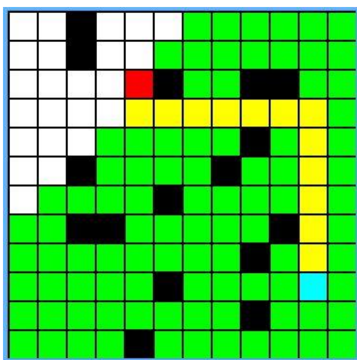


A*: 73

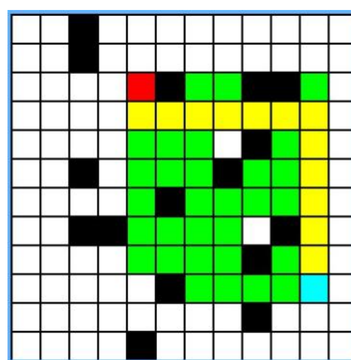


RRT: 47

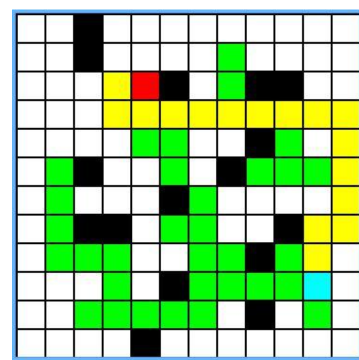
- Escenario on 144 celdas. Ahora no solo ponemos el destino en una celda aleatoria, sino que también ponemos la celda origen en un punto cualquiera. Conseguimos resultados parecidos tanto con el A* como con el RRT.



Dijkstra: 105

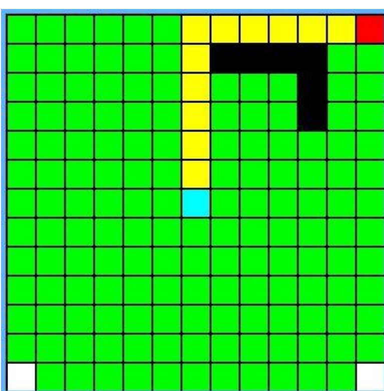


A*: 44

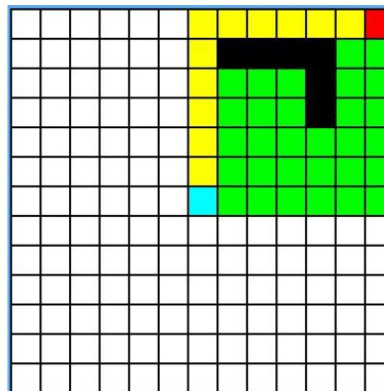


RRT: 49

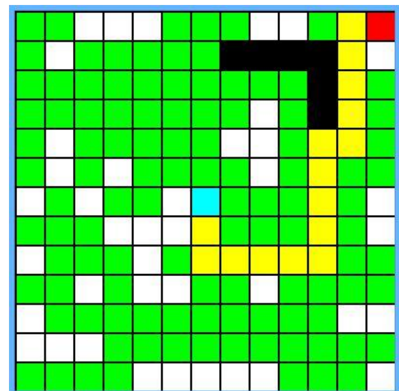
- Escenario con 169 celdas. En este caso mantenemos la celda destino en un extremo, pero situamos el origen en el centro. Comprobamos que el A* es mucho más eficiente que los demás en esta situación.



Dijkstra: 161



A*: 43



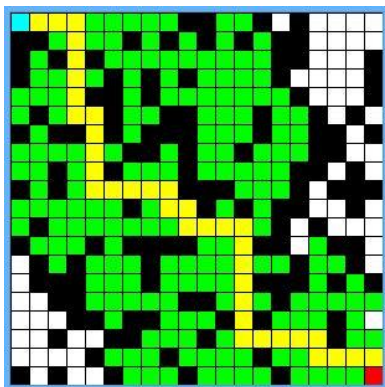
RRT: 122

Hasta aquí podemos deducir nuestras primeras conclusiones. En primer lugar, el A* va mejorando respecto al Dijkstra cuanto más cerca se encuentra del destino y cuanto más centrado está en el escenario, ya que gracias a la función heurística sabe cual es la dirección más adecuada a seguir.

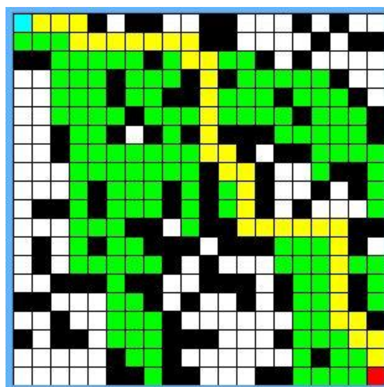
Por otro lado, vamos viendo que el algoritmo RRT no parece una buena opción cuando el destino está en algún borde o extremo del laberinto. Sin embargo, va ofreciendo mejores resultados cuando la celda a la que quiere llegar se encuentra en un punto aleatorio dentro del escenario y no tan pegado a los extremos.

Seguimos con el análisis del rendimiento de los algoritmos, pero en escenarios de orden mayor. A partir de aquí los laberintos se crean de forma aleatoria, pudiendo modificar la densidad de obstáculos a nuestro gusto.

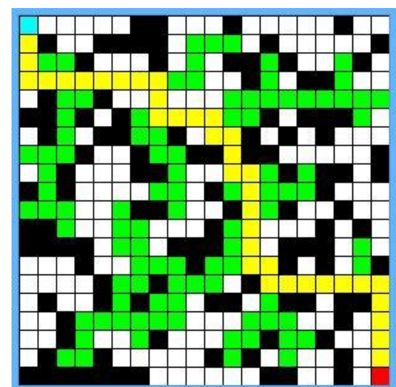
- Escenario de 400 celdas. Aunque en este caso el RRT obtiene el mejor rendimiento, no parece destacar respecto a los otros algoritmos, ya que se podría decir que en esta ejecución la suerte ha estado de cara.



Dijkstra: 218

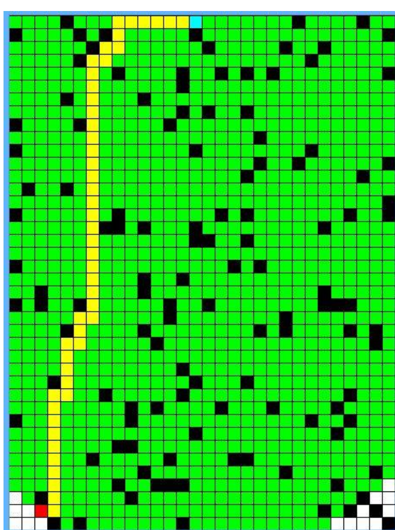


A*: 178

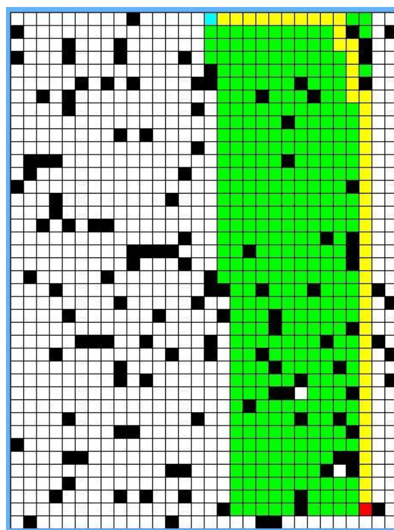


RRT: 133

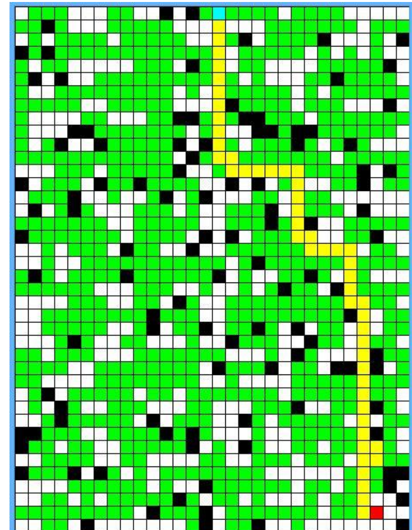
- Escenario de 1.200 celdas. El rendimiento del A* mejora respecto al RRT debido a que la posición origen se encuentra más próxima al destino que en ocasiones anteriores. Además, la densidad de obstáculos en este escenario ha sido menor que en el ejemplo anterior.



Dijkstra: 1062

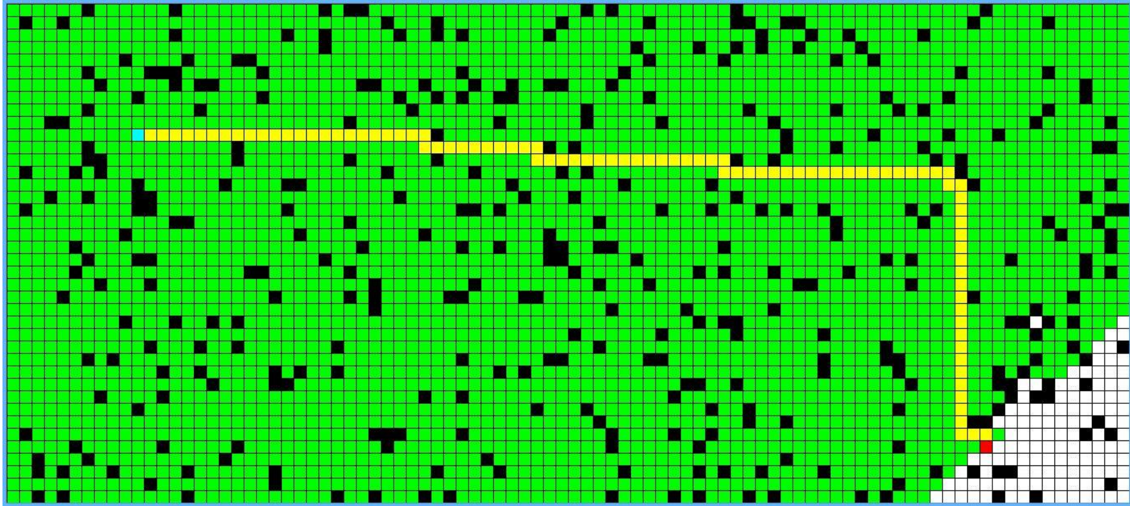


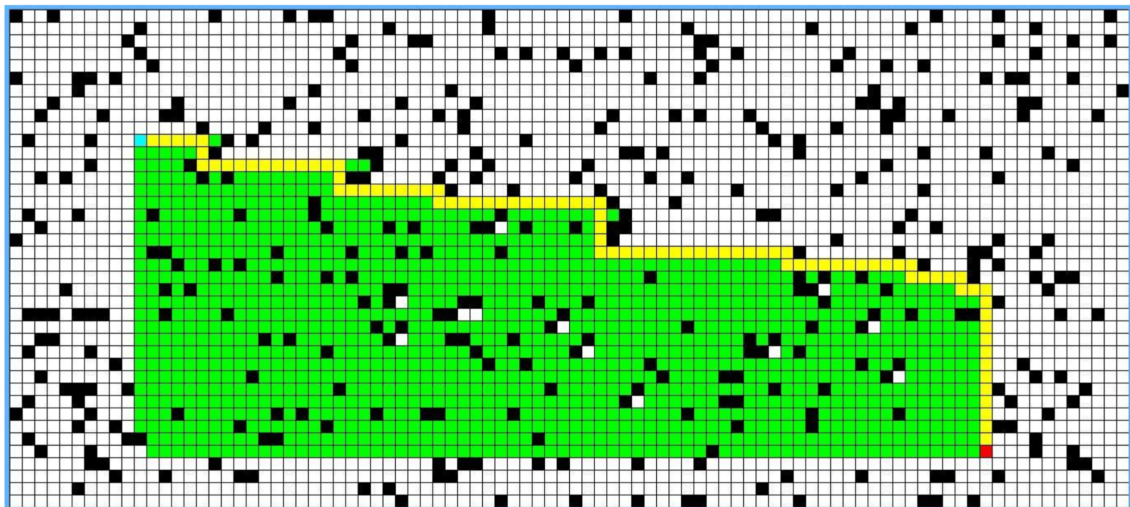
A*: 414

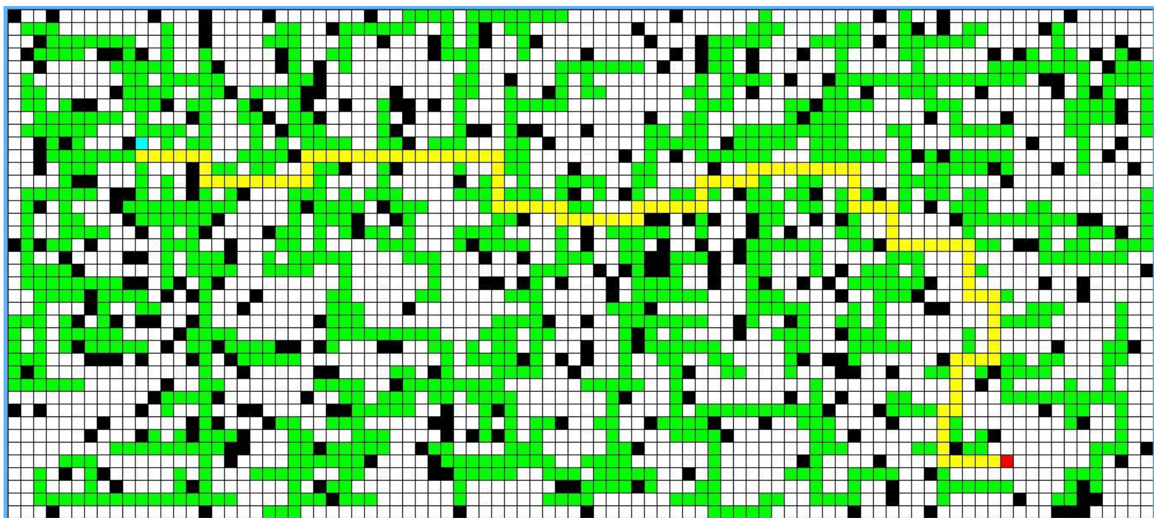


RRT: 717

- Escenario de 3.600 celdas. A pesar del aumento de las dimensiones del escenario, se sigue cumpliendo la misma norma: tanto RTT como A* son mucho más eficientes que el Dijkstra.

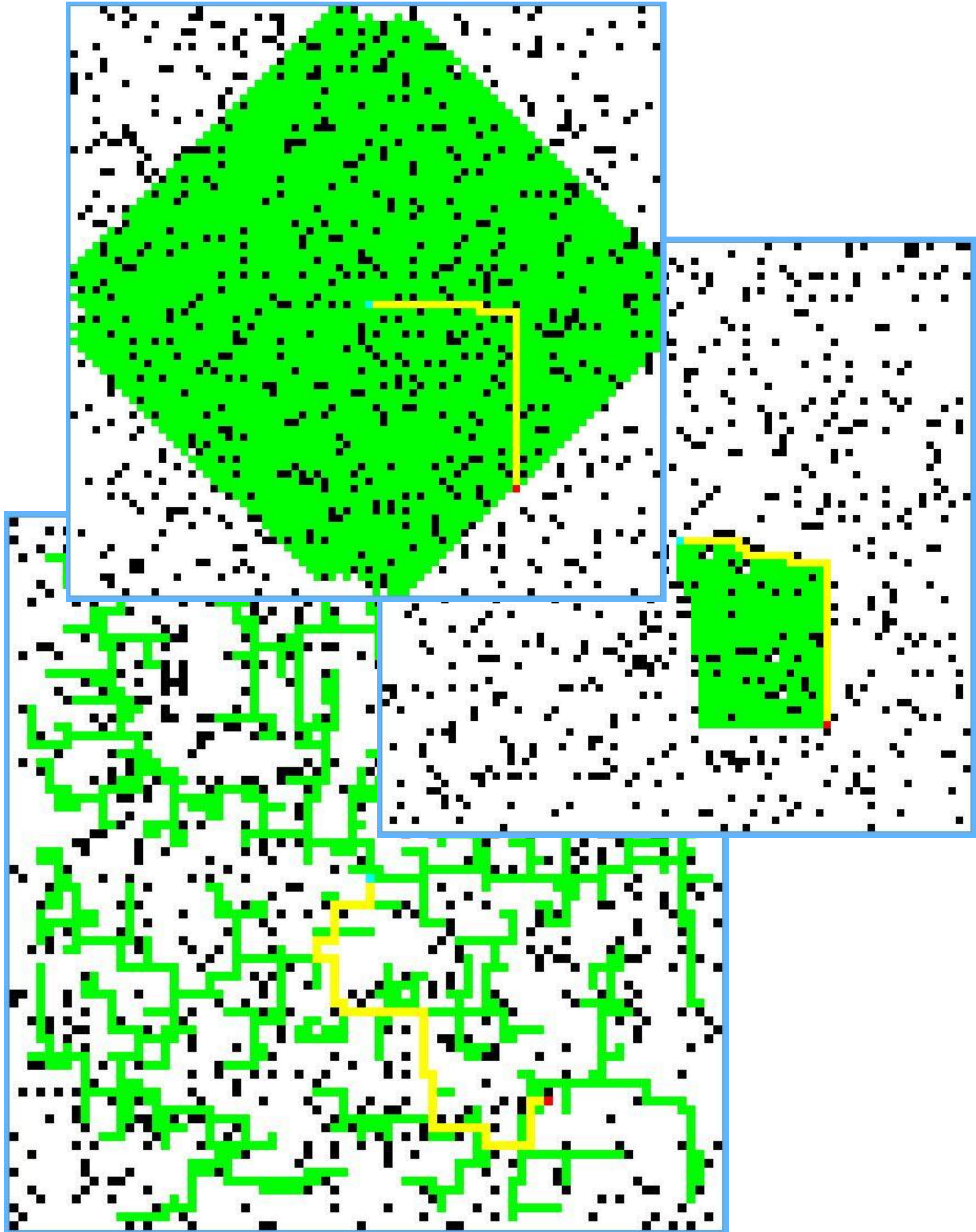






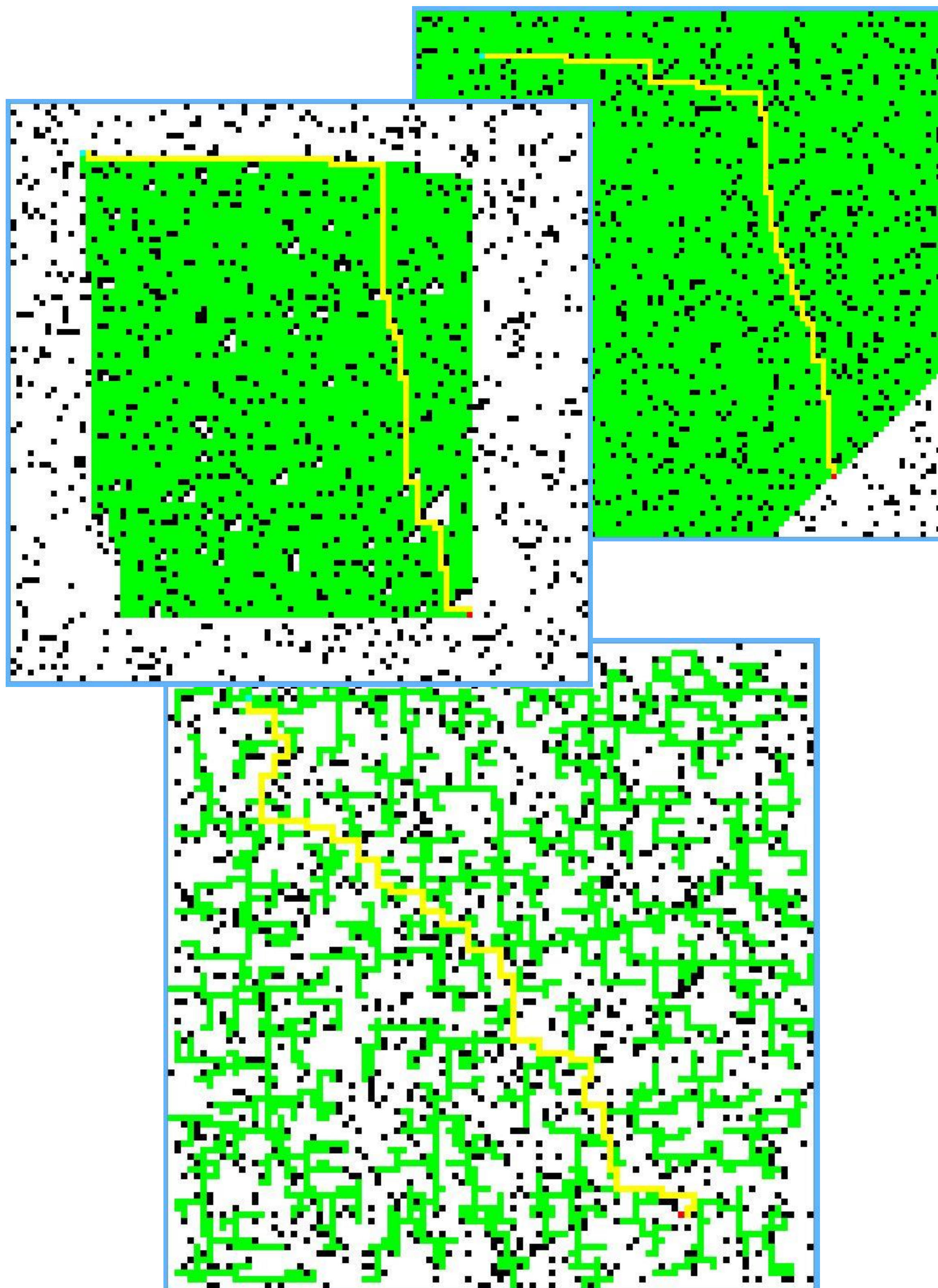
Dijkstra: 3.110 **A*: 1.250** **RRT: 1.291**

- Escenario con 6.400 celdas. De orden mayor, empezamos en el centro, quitando la cuadrícula de forma que se asemeje más a una situación real. Aunque el A* da grandes resultados debido a la proximidad de origen y destino, el RRT también ofrece un buen resultado.



Dijkstra: 3.231 **A*: 429** **RRT: 1.364**

- Escenario de 10.000 celdas. Concluimos este capítulo con un laberinto de enormes dimensiones, en el que la posición inicial y final se encuentran alejadas la una de la otra. El RRT es el que mejor rendimiento ofrece en este caso, gracias a las búsquedas aleatorias del árbol de expansión.



Dijkstra: 8.554 **A*: 4.856** **RRT: 2.841**

3 ENTORNO BLENDER. DESCRIPCIÓN GENERAL DE BLENDER E IMPLEMENTACIÓN DE ALGORITMOS

Para definir qué es Blender y en qué consiste, qué mejor que recurrir a su página web (https://www.blender.org). En esta, Blender es descrito brevemente como una herramienta de animación libre y de código abierto para la creación tridimensional.

Ampliando su definición, podemos referirnos a Blender como un software que cubre todos los aspectos relacionados con el mundo 3D, tales como modelado, animación, simulación física, animación con imágenes reales, edición y creación de video, módulo de scripting, creación y procesamiento de variadísimos materiales y texturas etc. Por si fuera poco, también incluye un motor de videojuegos.

Entre sus principales características destacan la de ser un software gratuito y libre con código abierto (*open source*). A continuación veremos cómo surge Blender y su evolución a lo largo de más de veinte años de vida.

3.1. Historia de Blender

Para saber de dónde viene Blender, que no fue libre en un principio, tenemos que remontarnos a 1988. Fue entonces cuando Ton Roosendal, el creador de Blender, co-fundó NeoGeo, un estudio de animación holandés. Esta empresa obtuvo gran reconocimiento, y a inicios de los 90 consiguió premios a nivel internacional, teniendo como clientes a grandes compañías como Philips.

Tras varios años de uso, la herramienta empezó a quedarse obsoleta, por lo que en 1995 Tom opta por mejorar el software interno de animación que usaba la compañía. Decide rediseñar desde cero un nuevo software de animación y creación 3D que a la postre sería presentado como Blender.

En 1998 Ton decide que el software que han hecho internamente dentro de NeoGeo tiene viabilidad comercial para que otras empresas y colectivos dedicados a la animación lo puedan usar. Decide por tanto fundar una nueva compañía con el nombre de NaN (Not a Number), la cual se encargaba de distribuir el software y vender productos y servicios empleando esta herramienta. Ya bajo el nombre de Blender, la empresa tenía como uno de sus objetivos principales el que se pudiese llegar a distribuir esta herramienta gratuita de creación de gráficas en 3D, compacta y multiplataforma. Por aquel entonces, la mayor parte de programas dedicados al diseño gráfico suponían un desembolso de grandes cantidades de dinero para el usuario.

En 1999 Ton hace una gran campaña para dar a conocer este software y participa en la conferencia de la Siggraph, donde se confirma el gran potencial de Blender ganando una gran cantidad de adeptos. Muchos inversores se interesaron en el proyecto y pudieran recaudar alrededor de 4,5 millones de euros. A finales del 2000, la versión 2.0 ya contaba con más de 250 mil usuarios.

Convertida ya en una multinacional y con representación en varios países del mundo, en abril de 2001 se lanzó *Blender Publisher*. Este producto integraba aplicaciones para contenidos web, algo único en esa época ya que los demás paquetes comerciales no brindaban la posibilidad de hacer graficas webs interactivas en 3D. Sin embargo, una serie de dificultades para la compañía tanto económicamente como en lo referido al número de ventas supusieron un parón en el desarrollo de Blender. Esta crisis hizo que muchos inversores abandonaran el proyecto, poniendo en serio peligro la continuidad de este proyecto.

Imagen 3.1 – Ton Roosendaal

Ton Roosendaal no quiso permitir la desaparición de este programa, y en 2002 decide crear y lanzar la organización no lucrativa *Blender Fundation*, una especie de ONG para salvar el desarrollo de esta herramienta. Decide así crear un plan de acción, cuyo objetivo consiste en liberar el código fuente y pasar a convertirlo en una plataforma *open source* basándose en una comunidad de usuarios.

En julio de ese mismo año Ton lideró una campaña con la que consiguió recolectar 100 mil euros entre todos los miembros de la comunidad de usuarios y antiguos clientes de *Blender Publisher*. Este dinero fue empleado para comprar los derechos del código fuente, y tras alcanzar este objetivo, en octubre de 2002 Blender se presentó como una herramienta open source con licencia GPL (Licencia Pública General).

A día de hoy continúa desarrollándose y evolucionando gracias a la colaboración de la comunidad Blender. Esta comunidad está compuesta por usuarios de todas las partes del mundo que comparten sus proyectos, ideas y dudas, haciendo que el software mejore progresivamente y expandiendo su uso entre nuevos usuarios.

3.2. Descripción general

Volviendo a sus campos de aplicación, a pesar de ser gratuito, Blender es considerada una de las mejores herramientas 3D que existen en la actualidad. Para entender lo que es capaz de hacer esta herramienta, recurrimos de nuevo a su página web, pudiendo ver a continuación algunas de las infinitas posibilidades que ofrece este software:

Modelado: Herramientas de modelado que permiten diseñar, transformar y editar todo tipo de modelos. Para ello tenemos múltiples comandos de teclado, así como accesos directos que hacen esta tarea mucho más fluida y rápida. Además, tenemos la opción de usar scripts en código Python para añadir nuestras propias herramientas y creaciones. Todo esto se verá más en detalle en el apartado siguiente de implementación de algoritmos y animaciones.

Imagen 3.2.1

Herramientas de animación: Nos da la opción de representar nuestros objetos después de modelarlos. Para ello nos permite la implementación de trayectos, trazos de curvas o definición de movimientos para los sólidos creados. Permite incluso la intuición automática de movimientos y la sincronización de estos con el sonido.

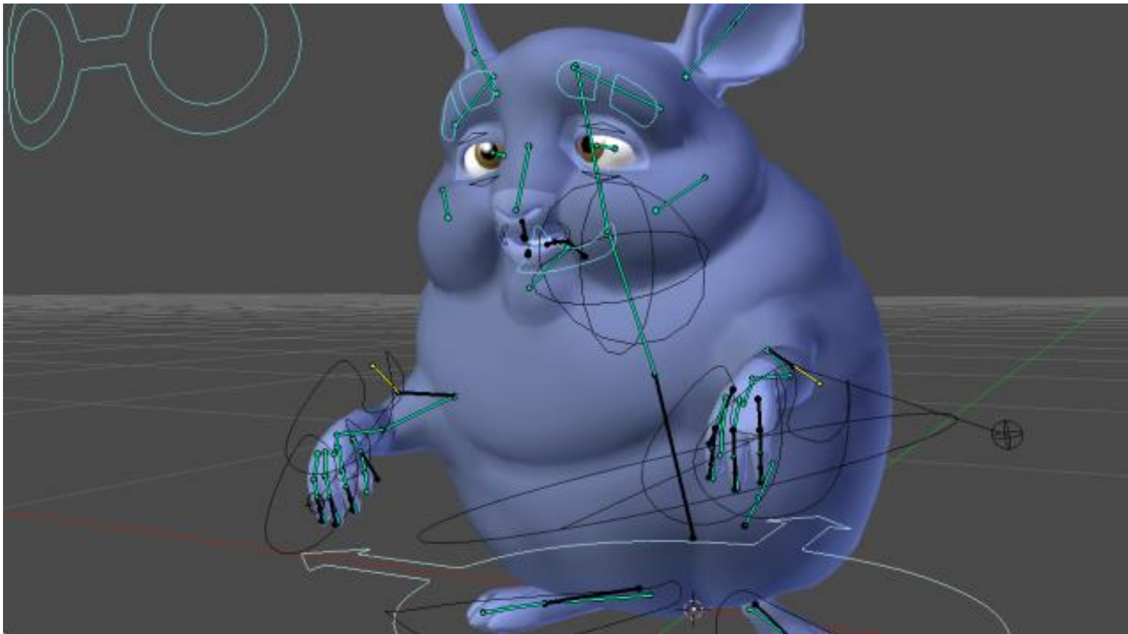


Imagen 3.2.2

Seguimiento de cámara y objetos: Incluye la opción de realizar un seguimiento con cámaras de la escena tanto virtual como automático, pudiendo grabar desde varios puntos que desee el usuario a la vez. Permite manejar también aspectos como la iluminación para visionar la escena.

Motor de juegos integrado: Nos permite crear aplicaciones 3D interactivas y videojuegos, capacidad para importar modelos de otro motor de videojuegos a Blender, creación de tu propia lógica de juego, librerías de audio para manejar el sonido etc.

Podríamos seguir resumiendo tareas capaces de ser realizadas por Blender, pero este proyecto no profundiza en esos aspectos. No se trata pues de modelar un escenario al detalle, ni tampoco de llevar a cabo una iluminación precisa y detallista de la escena. El trabajo a realizar se centrará en primero representar un escenario básico, en nuestro caso una especie de laberinto en 3D, y posteriormente una animación de un sujeto recorriendo ese laberinto. Para recorrerlo, tendremos que haber realizado un planeamiento de rutas gracias a los algoritmos de búsqueda explicados previamente.

Imagen 3.2.3

Respecto a las propiedades principales de Blender hay que destacar que se trata de un software ligero. Esto significa que funciona con unos requisitos mínimos, no haciendo falta recursos sofisticados como tarjetas gráficas potentes ni ordenadores de última generación. Podríamos usarlo con un equipo de trabajo básico; su bajo consumo de memoria permite usarlo en prácticamente cualquier ordenador. Además, funciona bajo

cualquier plataforma, habiendo versiones para Windows, Mac OS y Linux.

A su vez la comunidad de usuarios que Blender tiene detrás es enorme. Usada en todo el mundo, es innumerable la cantidad de tutoriales, foros, video-blogs y páginas que nos dan la posibilidad de consultar cualquier duda. De hecho, para el desarrollo de este proyecto he recurrido a tutoriales en los que se explican los conceptos básicos para usuarios que estén empezando a usar el programa.

Es un software muy potente, que evoluciona continuamente, saliendo versiones cada pocos meses que incorporan nuevas propiedades o mejoras. Sin embargo, su funcionamiento general no varía mucho por lo que podemos usar versiones distintas sin grandes problemas. La versión empleada en este trabajo ha sido la más actual (versión 2.77a).

Por último, Blender se puede extender todo lo que queramos con el lenguaje de programación Python. Éste nos brinda la posibilidad de añadir cualquier script para funciones concretas, pudiendo modificarse y adaptarse a las necesidades de la persona o personas que lo usen. En los siguientes apartados profundizaremos en este aspecto.

Vistas las características de Blender que más nos interesan para el trabajo que desarrollaremos, así como sus numerosos campos de aplicación, nos centraremos en el módulo de scripting. Es aquí donde serán implementados los algoritmos de búsqueda, crearemos los distintos escenarios y llevaremos a cabo la animación con la que veremos cómo se resuelve la búsqueda de caminos en entornos tridimensionales.

3.3. Interfaz de Blender

Cuando iniciamos el programa, nos encontramos con una interfaz que a primera vista puede resultar poco atractiva. Por defecto aparecen una cuadrícula acompañada de un cubo y una cámara con la que podemos realizar nuestras primeras pruebas. Sin embargo, Blender nos ofrece muchísimas posibilidades a la hora de personalizar este entorno.

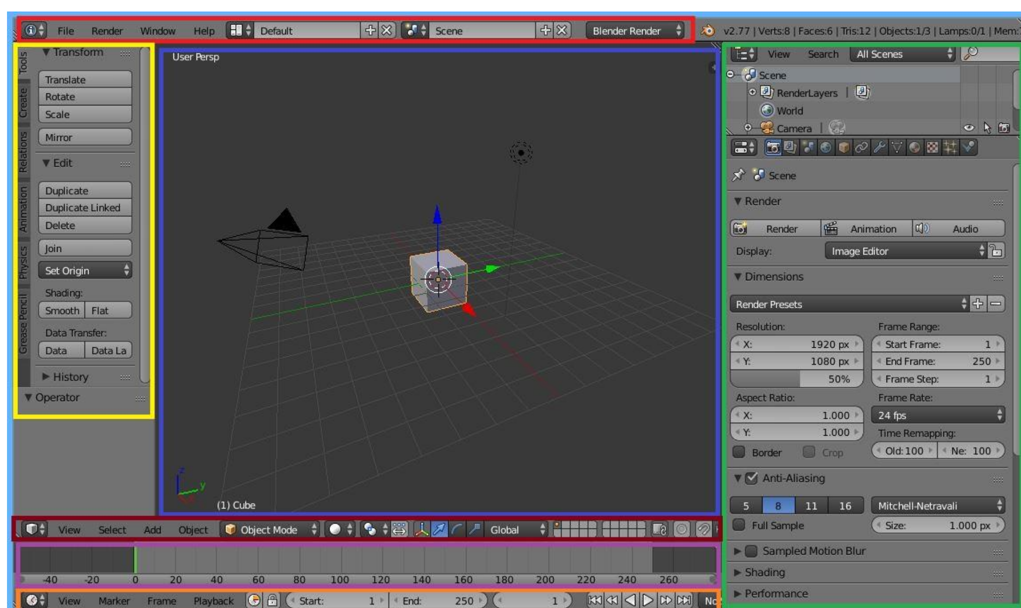


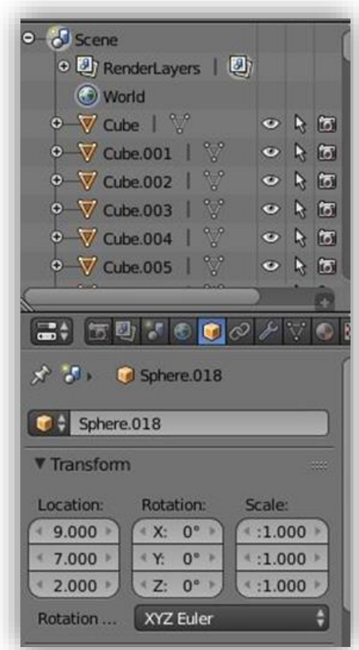
Figura 3.3.1

Lo más aconsejable para aquellas personas que empiecen a usar el programa desde cero es recurrir a la infinidad de tutoriales que hay en la página www.youtube.com. En ellos se explican los aspectos fundamentales, empezando por los comandos más básicos y avanzando por todas las posibilidades que nos brinda el programa.

En un principio la interfaz puede resultar poco amigable y compleja, pero a medida que trabajemos con ella se nos irá haciendo más sencilla de utilizar. Es configurable en todos los aspectos y tras familiarizarse un poco con ella es muy intuitiva. Para resumir las propiedades de esta interfaz, enumeraremos los aspectos más importantes a tener en cuenta:

- El espacio de trabajo se divide en ventanas o áreas de trabajo, y cada ventana tendrá asignado un panel de control o barra de herramientas.

- En el panel superior o cabecera tenemos las opciones propias de Blender, pudiendo abrir, guardar y manejar ficheros en general, seleccionar el espacio de trabajo, agregar y seleccionar objetos de la escena o cambiar el tipo de visualización entre otras opciones. Por defecto aparece el espacio de trabajo *default*, pero tendremos la opción de cambiarlo. En esta pestaña aparecen espacios de trabajo predefinidos para trabajos específicos tales como animación, scripting, edición de video etc. También podemos definir un espacio de trabajo propio y personalizarlo.



- En el panel izquierdo tenemos el *object tools* o herramientas de edición del objeto que permiten escalar, rotar, trocear y modificar un objeto de múltiples formas.

- El panel derecho se muestran los componentes de nuestro proyecto, bien sea por capas, objetos, materiales etc. En él encontramos las características y parámetros de cada elemento de la escena, pudiendo también modificar todo el entorno. En este panel encontramos además las opciones de crear texturas, materiales, elegir colores etc.

- En Blender tenemos la posibilidad de desplazar las líneas de división, pudiendo cambiar el tamaño del área de cada ventana y asignarle la función que prefiramos. Se pueden ampliar, reducir y ocultar el tamaño de todos estos paneles para tener mayor área o áreas de trabajo. Así podremos personalizar la interfaz a nuestro gusto, con las ventanas y barras de herramientas que nos sean de más utilidad.

Aunque tiene la posibilidad de poner todos los menús y elementos en español, he trabajado en todo momento con la interfaz en inglés, ya que es la que te viene por defecto y la que usa la inmensa mayoría de los usuarios. El resto de configuraciones y parámetros dentro de la interfaz también se han dejado por defecto para no añadir complejidad a la implementación del proyecto.

Una vez tenemos una idea general de cómo funciona Blender, nos centraremos en su espacio de trabajo para programar, denominado *scripting*. El lenguaje de programación empleado será el Python, por lo que antes de entrar en detalle en la implementación de los algorirmos y la animación, vamos a ver unas pinceladas básicas de este lenguaje.

3.4. Lenguaje Python

En este apartado trataremos por encima los aspectos básicos del lenguaje Python para la implementación de algoritmos y las posibilidades que éste nos ofrece para interactuar con el software Blender.

Veremos ciertos detalles a tener en cuenta a la hora de programar en Phyton, sin entrar en detalles específicos de la sintaxis ni analizar todas las posibilidades que nos ofrece. No olvidemos que no es una guía de Python, sino una forma de ver cómo podemos combinar este lenguaje de programación con la herramienta Blender.

3.4.1. Fundamentos de Python

Es un lenguaje de programación de alto nivel que empezó a desarrollarse a finales de los 80. Puede usarse tanto en scripts como en programas independientes. Una de sus principales características es la sintaxis clara e intuitiva, ya que en muchos aspectos recuerda a otros lenguajes de programación como Java o C.

Una vez nos ponemos a programar este proyecto, tenemos que tener en cuenta una serie de premisas importantes como:

- No hace falta declarar las variables al principio, pudiendo inicializarse directamente una vez vayan a ser usadas. Las expresiones en cada línea no terminan con un punto y coma (;), y las expresiones de control para los distintos tipos de bucles no hace falta ponerlas entre paréntesis.

- Los bloques se agrupan en función de la tabulación en vez de con paréntesis o corchetes. Tanto para bucles como para funciones, las órdenes realizadas en cada caso vienen recogidas en función del sangrado. Cada bloque comienza con el carácter dos puntos (:). Aunque pueda parecer un poco incómodo, está configurado de forma que cuando saltamos de línea tras los dos puntos, el sangrado correspondiente se pone automáticamente. Además, también se puede usar el tabulador que por defecto viene configurado para realizar correctamente la función de sangrado.

Respecto a la sintaxis, Python nos ofrece una serie de palabras clave de gran utilidad a lo largo de los scripts implementados. En general estas palabras son muy intuitivas, y nos ayudan a ahorrarnos funciones que tendríamos que implementar en caso de que dichas palabras no existieran. Entre las más usada a lo largo del proyecto están: ***len, range, and, or, sort, pop***…

```
if x == fin[0] and y == fin[1] and z == fin[2]:
    completada = True
else:
    for i in range(len(movimiento)): #Analizo los 6 movimientos posibles
        x2 = x + movimiento[i][0]
        y2 = y + movimiento[i][1]
        z2 = z + movimiento[i][2]
```

Ejemplo 3.4.1 – Script en Python para Blender

En el siguiente ejemplo apreciamos como se programa un bucle ***if else***, seguido de un buble ***for***, a la vez que se emplean algunas de las palabras claves antes mencionadas.

En Python todas las variables son tratadas como objetos. Cada objeto tiene asociado un identificador, un tipo (dependiendo de sus atributos), los métodos que dependen del tipo que sea el objeto y un valor. Tanto el identificador como el tipo de un objeto son invariables.

Python también ofrece la posibilidad de crear nuestras propias clases, asignarles atributos y métodos. Los

métodos de una case operan sobre los atributos de esta. Sin embargo, no profundizaremos en este aspecto ya que no hemos creado clases propias para la implementación de algoritmos.

Entre los tipos de objetos que más se usan en Blender y a lo largo de este proyecto, podemos decir que los más empleados en el trabajo son las tablas y las listas. Las tablas, como vimos en la primera parte del trabajo, sirven no sólo para describir el laberinto en el que trabajaremos tanto en 2D como en 3D. También sirven para ir analizando dicho escenario hasta alcanzar el camino buscado. Por su parte, las listas nos sirven para ir metiendo y sacando las localizaciones analizadas y sus características correspondientes, ya sean movimientos, costes o coordenadas.

3.4.2. API de Blender para Python

Cuando hablamos de una API (*Application Program Interface*) nos referimos a un conjunto de funciones con las que controlar un programa externo. Blender incorpora una API para Python que nos ofrece muchísimas posibilidades dentro del programa, llegando a ser en ocasiones demasiado extensa y compleja.

Para combatir la complejidad del programa en ciertas ocasiones, Blender nos ofrece recursos de gran utilidad. Dispone de una ventana opcional indicada como "*info*". En esta podemos ver reflejadas en lenguaje Python todas las acciones que vamos realizando con los comandos del ratón y el teclado.

Supongamos que quisiéramos crear un cubo o cualquier otra figura y asignarle un determinado tamaño en lenguaje Python. En un principio puede que no sepamos como hacerlo, pero si realizamos esta tarea con el ratón, en la ventana *info* van apareciendo todos los comandos traducidos al lenguaje Python. En el siguiente ejemplo vemos como tras realizar una serie de acciones en la escena, los movimientos quedan reflejados en la ventana de comandos

```
File   Render   Window   Help   Scripting   Scene   Blender Render
alse, False, False, False, False, False, False))
bpy.ops.mesh.primitive_uv_sphere_add(size=0.1, view_align=False, enter_editmode=False, location=(9, 7,
2), layers=(True, False, False, False, False, False, False, False, False, False, False, False, False, F
alse, False, False, False, False, False, False))
bpy.ops.text.run_script()
bpy.context.space_data.context = 'TEXTURE'
bpy.ops.transform.translate(value=(0, 0, 1.68573), constraint_axis=(False, False, True), constraint_ori
entation='GLOBAL', mirror=False, proportional='DISABLED', proportional_edit_falloff='SMOOTH', proportio
nal_size=1, release_confirm=True)
bpy.ops.object.delete(use_global=False)
```

Ejemplo 3.4.2 – Ventana *info*

Dentro de esta API encontramos una serie de bibliotecas que nos permiten trabajar con funciones de todo tipo y manejar la totalidad del programa mediante Python. Estas bibliotecas de funciones, conocidas en Blender como módulos, permiten controlar el programa desde la ventana de *scripting*. Así, podremos añadir figuras, localizarlas dentro del escenario, modificarlas y prácticamente realizar cualquier acción que hagamos con el ratón y el teclado mediante Python.

Por otro lado, existen también módulos dedicados a funciones matemáticas o que nos permiten controlar el sistema de sonido. Siendo conscientes del potencial de esta API, pasamos a la implementación del algoritmo, la cual podemos dividir en dos procesos: implementación de los algoritmos de búsqueda y desarrollo de la animación.

3.5. Implementación de algoritmos y animaciones

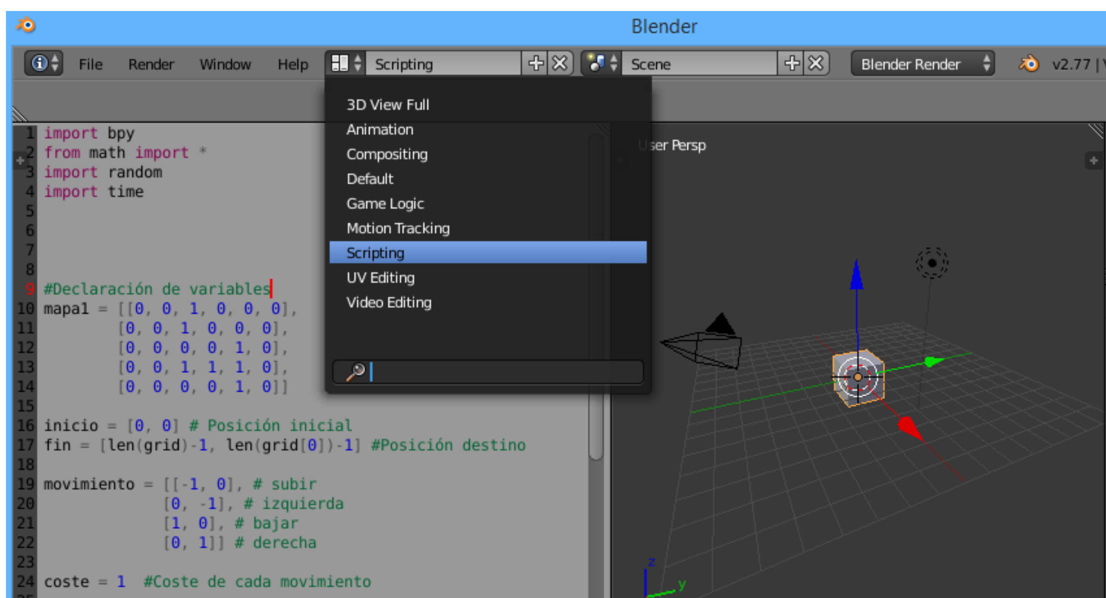
Una vez visto qué es Blender, las características principales de su interfaz y las posibilidades que tenemos de controlarlo con Python, empezamos la implementación del proyecto. Nos centraremos en las dos ventanas de trabajo sobre las que desarrollaremos la implementación de los algoritmos de búsqueda.

Primero trabajaremos sobre el espacio de trabajo de *scripting*, en el cual desarrollaremos los algoritmos de búsqueda y las animaciones. En segundo lugar pasaremos al espacio de trabajo *animation*, en el que podremos representar la animación creada en la que nuestro sujeto recorrerá el laberinto especificado.

Para ello abrimos el módulo de *scipting* y empezamos a programar. Para alguien que nunca haya programado en Blender o no sepa muy bien cómo empezar, de nuevo en la plataforma www.youtube.com hay bastantes ejemplos de cómo funciona esta herramienta. Hay disponibles varios tutoriales en los que mediante el diseño de scripts muy básicos vamos captando la forma de trabajar dentro de este módulo.

Lo primero que haremos será importar las librerías necesarias para el proyecto. Para trabajar con objetos dentro de Blender hace falta importar la librería *bpy*. Si queremos usar funciones aleatorias (como en el RRT) hará falta la librería *random*, para trabajar con frames la librería *time* etc.

Los módulos se importan mediante la palabra clave *import*. En nuestro caso, hemos usado *import bpy*, ya que importamos el módulo general *bpy*, aunque en proyectos más amplios es recomendable importar módulos particulares para evitar posibles conflictos de nombres. Existe otra posibilidad a la hora de importar funciones; importar los módulos particulares directamente. Es lo que hacemos en el caso de la librería *math*.



Ventana 3.5.1 – Ventana de scripting e importación de módulos

Tras esto, el siguiente paso a realizar consiste en declarar las variables. Para ello tenemos que manejar aspectos característicos de este lenguaje. Por ejemplo, a la hora de acceder a los elementos de una fila o tabla, tendremos que tener en cuenta que los índices empiezan en 0. Esto es un detalle importante, ya que no siempre es así. De hecho, en el programa V-REP con el que trabajaremos más adelante, los índices empiezan por 1, cambiando así

todo el acceso a tablas, declaración de variables etc.

A continuación pasamos a declarar las funciones que nos irán haciendo falta a lo largo del programa. Python, al igual que otros lenguajes de programación, nos ofrece la posibilidad de ir llamando a funciones que nos devuelvan las variables buscadas. Así, definimos funciones que iremos llamando a lo largo del programa.

El modo de declarar sus variables también es muy cómodo y eficiente. En el siguiente ejemplo vemos cómo en unas pocas líneas declaramos y rellenamos la tabla de 3D con todos los datos de la función heurística.

```
20    heuristic = [[[0 for z in range(len(grid[0][0]))] for y in range(len(grid[0]))] for x in range(len(grid))]
21    for x in range(len(grid)):
22            for y in range(len(grid[0])):
23                for z in range(len(grid[0][0])):
24                    dist=abs(x-fin[0])+abs(y-fin[1])+abs(z-fin[2])
25                    heuristic[x][y][z]=dist
```

Ventana 3.5.2 – Función heurística en 3D

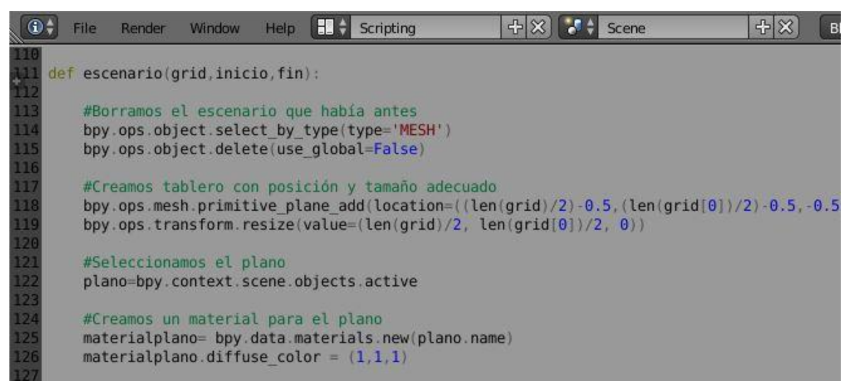
Por lo tanto dentro del script iremos definiendo funciones. La primera función será ***escenario,*** y servirá para crear el entorno en el que se llevará a cabo la animación. En ella, recurriendo al módulo *bpy* que permite manejar y editar objetos de Blender, seguimos los siguientes pasos:

- Borramos todos los objetos de la escena para empezar desde cero.

- Creamos un tablero que nos servirá como base del escenario y de referencia visual, en la posición que nos interese y con el tamaño adecuado.

- Dibujamos todos los obstáculos que forman el laberinto. Para ello recorremos la matriz ***mapa*** (o ***grid***) que le pasamos como parámetro a la función escenario.

Dentro de esta función es importante también la creación de materiales. Aunque no es un aspecto en el que profundizamos, sí que es importante definir materiales para los distintos elementos del laberinto. Esto quiere decir que tanto el plano como los obstáculos como otros componentes de la animación estarán hechos de distintos materiales, de forma que sea posible distinguirlos con facilidad.



Ventana 3.5.3 – Trozo de código de *escenario*

En los ejemplos vemos como el plano es prácticamente de color blanco mientras que a los obstáculos les hemos asignado un material de color azul. Una vez que tenemos el escenario creado tendremos que implementar la función central de este proyecto. Ésta no es otra que la que se encarga de la búsqueda de un camino desde la posición inicial que especifiquemos hasta una posición deseada dentro del laberinto. Desarrollaremos los tres algoritmos de búsqueda explicados con detalle en el capítulo previo, y en función de cuál queramos emplear llamaremos a una función u otra. Todo el código para Blender, tanto de los algoritmos como del resto de

funciones, viene adjuntado en el Anexo A. Nos quedarían por tanto tres funciones de búsqueda de caminos a las que recordemos que teníamos que pasarles como datos de entrada el escenario, la posición inicial y la posición final.

- ***Dijkstra*(mapa, inicio, fin)**
- ***A**(mapa, inicio, fin)**
- ***RRT*(mapa, inicio, fin)**

La terccera función que implementaremos será ***animación***. Esta función será la encargada de crear un sujeto que recorra el laberinto. En Blender, una de los objetos más peculiares que se pueden añadir a la escena es la figura de un mono que viene ya predefinido. Elegimos por lo tanto este mono y lo colocamos en las coordenadas iniciales.

A lo largo de la animación la figura que recorrerá el laberinto tras haber resuelto uno de los tres algoritmos de búsqueda será el objeto llamado ***mono.*** Para añadirlo a la escena, veamos el tipo de sentencia que utilizamos para hacernos una idea de cómo funciona el módulo *bpy*:

bpy.ops.mesh.primitive_monkey_add(radius=0.3, view_align=False, location=(x, y, z))

***Mono* = bpy.context.object**

En la función ***animación*** también creamos un material que será empleado para el ***mono***. Normalmente le aplicaremos un material vistoso, para que destaque dentro del entorno en el que se va a mover. A los materiales les podemos asignar numerosas características, tales como densidad, rugosidad, brillo etc. Sin embargo, en nuestro caso nos centraremos en el color.

Los colores dentro de Blender, al igual que en otras muchas plataformas, vienen descritos por tres parámetros que varían entre 0 y 1. Estos números representan tres colores básicos y se combinan entre sí para formar el resto de colores. Para entenderlo con claridad, si queremos definir los colores negro, blanco o rojo deberíamos hacerlo de la siguiente manera:

Blanco = (1 , 1 , 1) Negro = (0 , 0 , 0) Rojo =(1 , 0 , 0)

Para crear un material nuevo y asignarle un determinado color, recurrimos a las siguientes sentencias:

***Material* = bpy.datta.materials.new(nombredelmaterial)**

***material*.diffuse_color = color**

Para finalizar, cuando tenemos el objeto (que en este aso será ***mono***) y el material con su correspondiente color, le asignamos a nuestro objeto este material con la siguiente sentencia:

***Mono*.data.materials.append(*material*)**

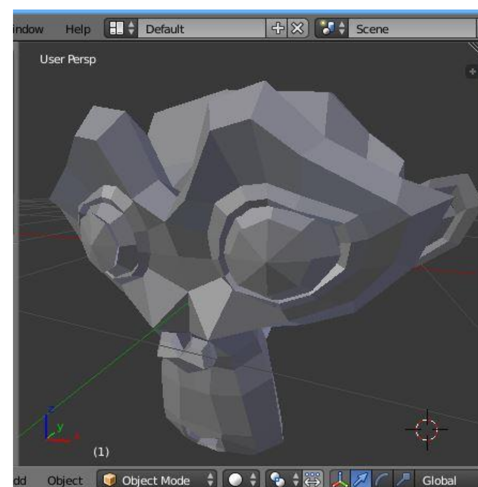


Figura 3.3.4 – Mono que recorrerá el labetinto

Creado el objeto que recorrerá el laberinto, con su correspondiente color, seguimos con la función ***animación***. A esta le pasamos como parámetro la ruta que tendrá que recorrer el mono desde el inicio hacia su destino final, por lo que vamos recorriendo las coordenadas contiguas que forman el camino. Para estas transiciones trabajaremos con el concepto de *frames*, los cuales funcionan como imágenes intermedias entre dos estados concretos. Cuántos más *frames* pongamos entre coordenada y coordenada, más suave y detallada será la animación, pero también más pesada en cuanto a uso de recursos y memoria.

Dentro de esta función usamos las herramientas con las que se manipulan las animaciones de Blender. A parte del número de *frames* por estado, podemos especificar cuándo acaba y cuándo empieza la animación, así como la velocidad de la misma o si queremos que se detenga en un determinado punto.

Para depurar la animación y observar si nos produce el resultado deseado trabajaremos en el espacio de trabajo *animation*. Como su propio nombre indica, es una ventana diseñada para trabajar con animaciones, en la que la interfaz se divide en secciones que permiten analizar la escena desde distintas perspectivas. Además, podemos pausar, adelantar y analizar la animación, así como modificar otros aspectos dentro de la misma.

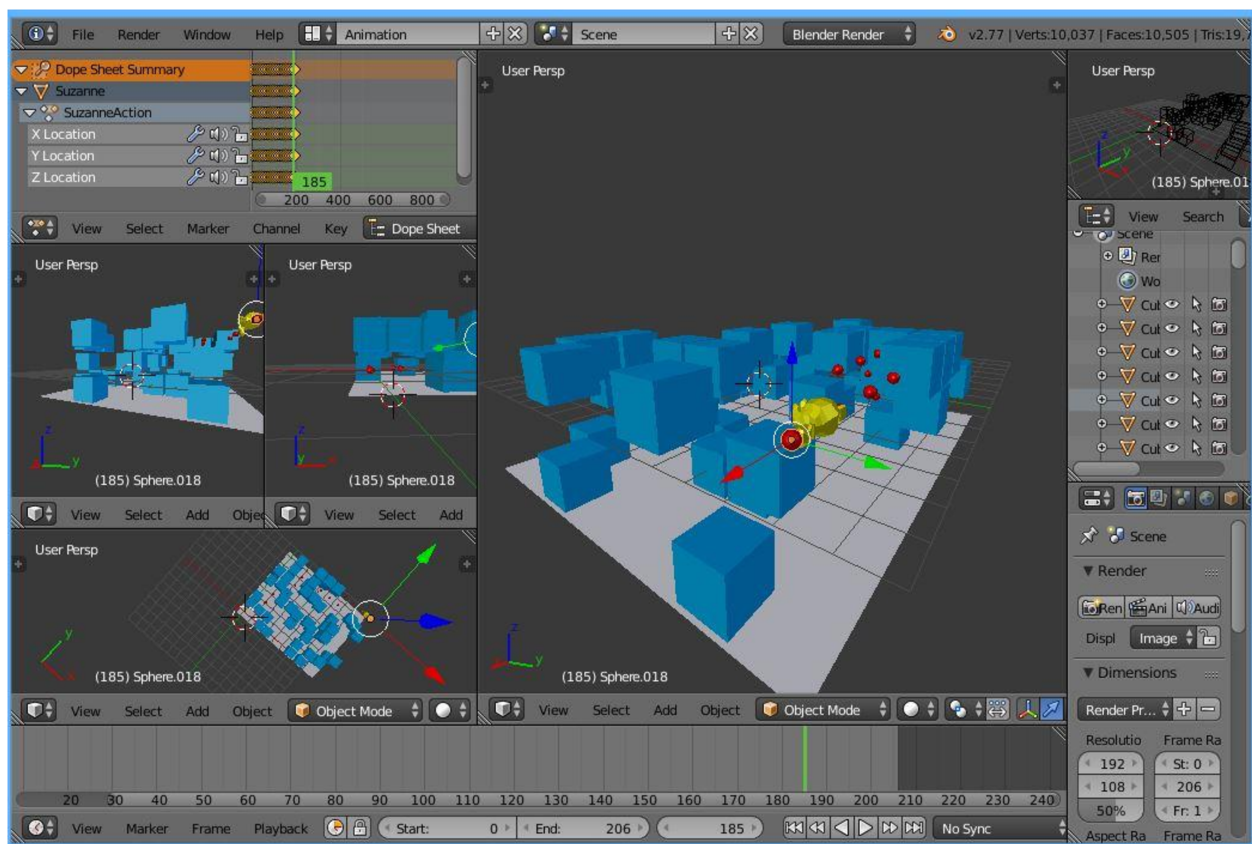


Figura 3.3.5 – Ventana de trabajo *animation*

Si nos detenemos en este punto del trabajo podríamos pensar que la implementación del proyecto está finiquitada. Tenemos un escenario tridimensional, un algoritmo de búsqueda que nos indica la ruta desde un punto hasta otro y por último una animación en la que nuestra figura va recorriendo ese camino calculado previamente.

Sin embargo, en ocasiones no sólo buscamos el mero hecho de atravesar el laberinto, sino que también queremos saber por dónde atravesarlo. En muchos escenarios se da el caso de que al tener grandes dimensiones no se aprecia de forma cl earal camino que nuestro objeto ***mono*** ha de seguir a lo largo del laberinto. Tenemos una

animación en la que un sujeto recorre una ruta, pero sin embargo esa ruta no somos capaces de apreciarla.

Este es el motivo por el que implementamos la última función: ***dibujoruta***. Ésta es bastante sencilla ya que su labor consiste en dibujar el camino encontrado desde el origen hasta el destino dentro de la animación. Como parámetro por lo tanto sólo habrá que pasarle el camino calculado previamente por el algoritmo de búsqueda especificado.

Aunque no es estrictamente necesaria, si que nos ayudará a ver reflejado nuestro trabajo, pues en todo momento aparece representado el camino calculado. Esto nos permite analizar la ruta en cualquier punto del laberinto mientras se está ejecutando la animación.

Definidas todas las funciones a emplear, podemos concluir resumiendo el código implementado en los siguientes pasos:

- Declaración de variables: definimos el mapa, las coordenadas del punto de origen y del fin.
- Llamada a la funión escenario, a la que le pasamos como parámetro la matriz mapa.
- Llamada a uno de los algoritmos de búsqueda, a los que le pasamos como parámetros el mapa, el origen y el destino. Nos devolverá una variable con la ruta si esque existe.
- Representación de la animación, a la que le pasamos como parámetro la ruta calculada.
- Representación del camino dentro de la animación.

Para finalizar con este capítulo del proyecto, en el Anexo A se adjunta todo el código desarrollado para el proyecto. Bastaría con abrir Blender desde el principio, añadir en el módulo de scripting todo el código implementado y por último ejecutar el script. Como resultado aparecerá una animación en la que una figura recorre un laberinto tridimensional, el cual era nuestro objetivo principal al principio de este trabajo. No obstante, más adelante el trabajo también incluye un capítulo con capturas de ejemplo de las animaciones creadas.

4 ENTORNO V-REP. IMPLEMENTACIÓN DE ALGORITMOS EN EL ÁMBITO DE LA ROBÓTICA

El segundo software de simulación en el que implementaremos los algoritmos de búsqueda de caminos en entornos 3D será V-REP, simulador desarrollado por la empresa Coppelia Robotics. Recurriendo a su página web (www.coppeliarobotics.com/), *Virtual Robot Experimentation Platform* es una plataforma suiza con el siguiente lema: "Create. Compose, simulate. Any robot"

4.1. Descripción general

Al igual que Blender, V-REP se trata de un un software gratuito y de código abierto. Pensado para la simulación de robots, funciona con Windows, Linux y OS. La finalidad de este programa, del mismo modo que otros simuladores robóticos, es permitir el manejo de robots virtuales, creando y simulando aplicaciones y tareas sin necesidad de realizar un desembolso en el prototipo real del robot. Facilita la investigación de las posibilidades que los diversos robots nos pueden ofrecer, así como el aprendizaje de cómo funcionan e interactúan entre sí los distintos robots.

V-REP no solo está pensado para programar tareas que posteriormente serán realizadas por robots. También tiene una labor educativa, profundizando en aspectos teóricos del funcionamiento de los prototipos y en materias robóticas. De hecho, este software tiene tres versiones:

- V-REP PRO EVAL, versión enfocada al ámbito comercial.
- V-REP PLAYER, diseñada específicamente para realizar simulaciones.
- V-REP PRO EDU, versión educativa y que será con la que trabajaremos en la implementación de este proyecto. Concretamente con la versión 3.3.1.

Como simulador robótico industrial, está enfocado a la representación de una situación en la que el robot debe satisfacer una serie de exigencias. Se tratará pues de recrear escenarios en los que el robot interactúe con el entorno de la manera lo más cercana posible a la realidad.

V-REP trae implementados numerosos robots y sus respectivas piezas, tales como hélices, pinzas, motores o ruedas. Tras crear un escenario determinado, mediante este software podremos representar los movimientos y acciones de un robot industrial. Esto permite diseñar situacciones que se ajusten a la realidad, ya que la mayoría de robots implementados son prototipos reales.

Muchas herramientas de simulación son desarrolladas por la propia marca del robot con el que se pretende trabajar, proporcionando éstas unos softwares de simulación adaptados exclusivamente a sus productos. Sin embargo, estos programas presentan el inconveniente de servir sólo para una marca en concreto, además de tener

un coste económico bastante elevado. La utilización de una herramienta de simulación *open source* nos permite trabajar con robots industriales de distintos fabricantes tales como ABB, HiBot o KUKA.

Este tipo de simuladores son recomendables para la programación y depuración de las tareas que le queramos asignar a un robot. Así, únicamente la versión final y depurada realizada con el simulador sería la que le implementaríamos al robot real para comprobar su correcto funcionamiento.

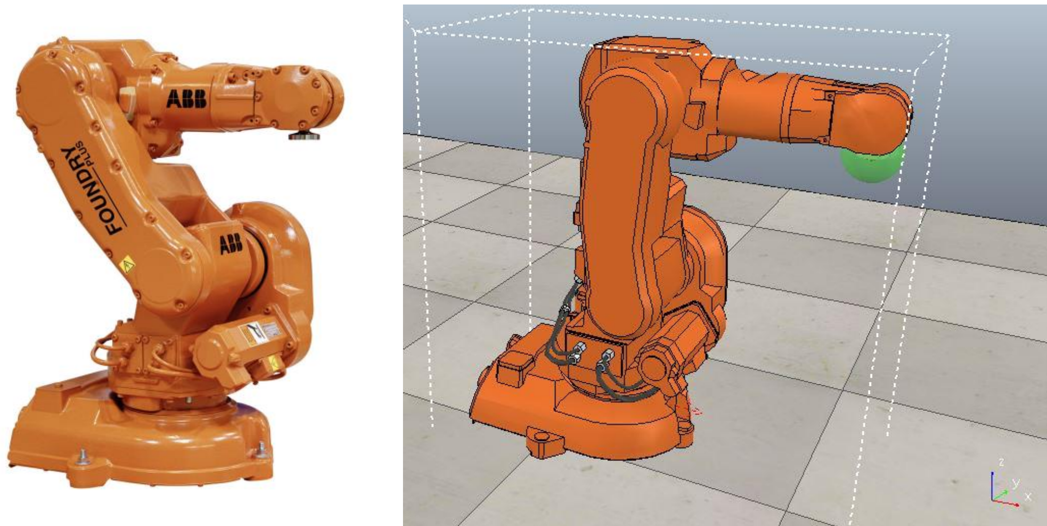


Figura 4.1 – Robot real vs Modelo V-REP

Además, V-REP permite programar scripts en Python, C++, Java o Lua entre otros lenguajes. Podremos implementar una serie de códigos que determinen el comportamiento de los elementos dentro de la escena. Estos scripts pueden ser asignados a los distintos elementos, ejecutándose de manera simultánea y realizando así tareas multi-robóticas. Para la implementación de los distintos algoritmos de búsqueda emplearemos el lenguaje Lua.

Aunque no entraremos en los siguientes aspectos, este programa nos permite trabajar con sensores de proximidad o de visión, así como con detectores de colisiones. Dispone de otras muchas herramientas para la simulación de escenarios reales y fluidos como el agua o la arena.

V-REP nos ofrece herramientas más allá de los robots que trae implementados, y aunque permite la creación de múltiples escenarios, no tiene herramientas de modelado tan potentes como Blender. De hecho, en este trabajo crearemos nosotros nuestro propio laberinto o escenario, y recorreremos dicho laberinto con un sólido implementado por el usuario.

Podemos concluir que este software está más enfocado a la investigación y desarrollo que a la creación de modelos en sí. Las posibilidades de simulación no son tan amplias como en Blender. Sin embargo, sí que te permite acceder a códigos fuentes de robots ya implementados y reprogramarlos para modificar su comportamiento y apreciar cómo funcionan.

4.2. Interfaz de V-REP

La interfaz de V-REP nada más iniciar el programa es bastante llamativa, más agradable a primera vista que la de Blender. Su entorno gráfico es vistoso, ofreciéndonos la posibilidad de mover las distintas pestañas o modificar el tamaño y posición de las mismas para trabajar a nuestro gusto. Esta interfaz de usuario está dividida en ventanas, siendo la ventana principal de simulación la más importante de todas. En ella aparecerán todos los elementos que vayamos incorporando a la escena.

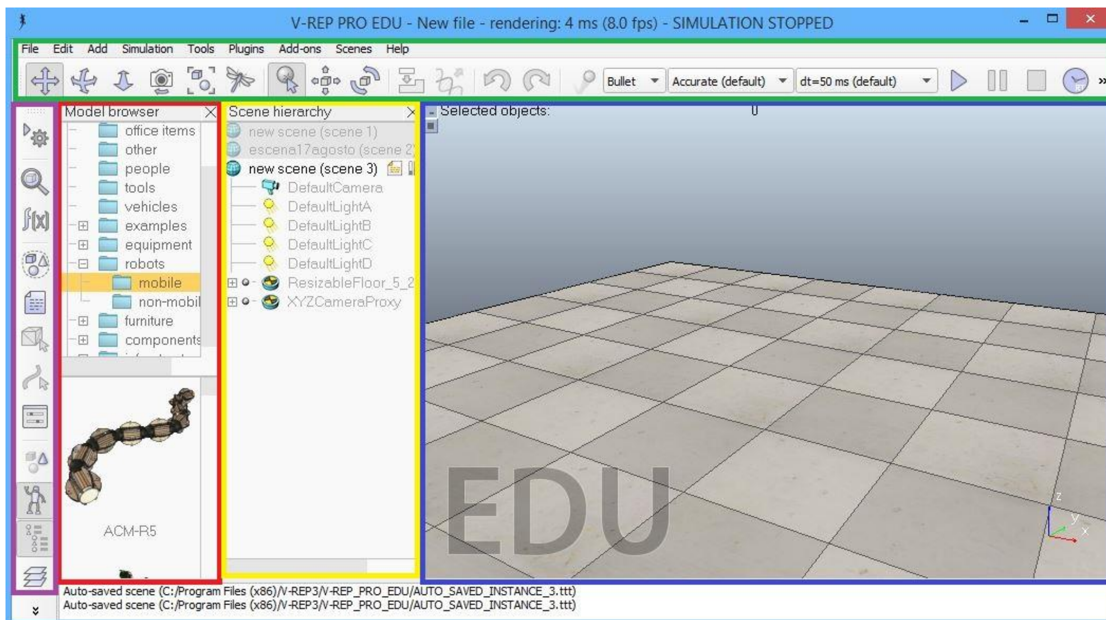


Figura 4.2.1 - Interfaz inicial de V-REP

V-REP cuenta con dos barras de herramientas. La vertical está conformada por botones que son enlaces a las funcionalidades principales del programa. Además, cuenta con selectores de vistas que nos permiten ver la escena desde distintas perspectivas. En la horizontal por su parte aparecen comandos para la navegación y exploración de elementos, así como para el manejo de las animaciones.

A la izquierda tenemos la ventana de jerarquía de la escena, en la que aparecen todos los objetos que hemos ido añadiendo en la escena de simulación. En esta ventana aparece reflejada la relación jerárquica que hay entre estos objetos, pudiendo depender entre sí o teniendo más prioridades determinados elementos respecto a otros.

Más a la izquierda tenemos el explorador de modelos. Aludiendo de nuevo a su página web, un modelo es un objeto propio de V-REP en el cual se encapsulan una serie de funcionalidades. Así, cada modelo tiene asignado un fragmento de código que determinará su comportamiento.

V-REP trae ya incorporados muchos modelos que nos ayudarán a representar la realidad de manera más rápida y cómoda. No obstante, entre los modelos ya cargados no solo aparecen robots. Veamos a continuación una especie de habitación con componentes añadidos al azar para ver el tipo de elementos que trae ya implementados el programa.

Como se aprecia en la imagen, podemos añadirle desde el mobiliario típico de una casa (estanterías, mesas, sillas, puertas etc.) hasta personas o plantas. Por supuesto a la vez podremos tener en la escena los robots que deseemos interactuando con los demás componentes.

Figura 4.2.2 – Habitación con componentes de V-REP

Dentro de la carpeta de robots incorporados, encontramos dos subcarpetas: robots móviles y robots no móviles. En los no móviles encontramos pinzas o brazos robóticos que desempeñan una función pero permanecen en un sitio fijo. En los móviles podemos ver robots mucho más variados con forma de gato o de grúa por ejemplo que van recorriendo el escenario en el que se encuentran.

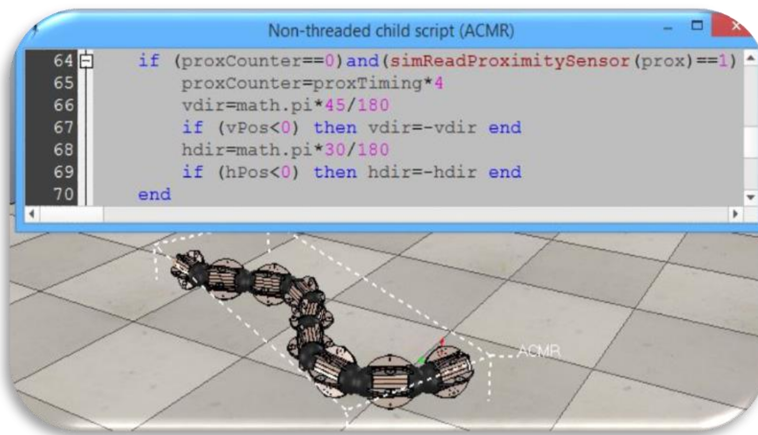


Figura 4.2.3 – Implementación del prototipo ACM R-5

Estos robots traen incorporados unos scripts que determinan su funcionamiento. Nosotros podremos analizar ese script, modificarlo y hacer tantas pruebas con el robot como queramos. En el siguiente ejemplo vemos como el prototipo de robot ACM R-5 incorpora un script que determina sus movimientos utilizando sus distintos sensores. Cuando pongamos en marcha la escena, su comportamiento variará en función de los obstáculos y sus sensores de proximidad.

4.3. Lenguaje Lua

Ahora que tenemos una idea de cómo funcióna el programa, pasaremos a la parte de programación o *scripting*. Para implementar nuestros algoritmos de búsqueda emplearemos scripts escritos en lenguaje Lua. A continuación veremos algunos de los aspectos más a tener en cuenta de este lenguaje antes de empezar a escribir nuestro código.

Recordemos también que cuando hablamos de una API (*Application Program Interface*) nos referimos a un conjunto de funciones con las que controlar un programa externo. Como se ve en el siguiente esquema, dentro de la arquitectura de V-REP disponemos de una API en lenguaje Lua en la que gracias a scripts que interactúan entre sí podemos ir manejando los distintos componentes que el programa nos permite implementar. Estos scripts tienen una estructura jerárquica, por lo que al igual que ocurre con los objetos, unos pueden estar por encima de otros y tener más prioridad.

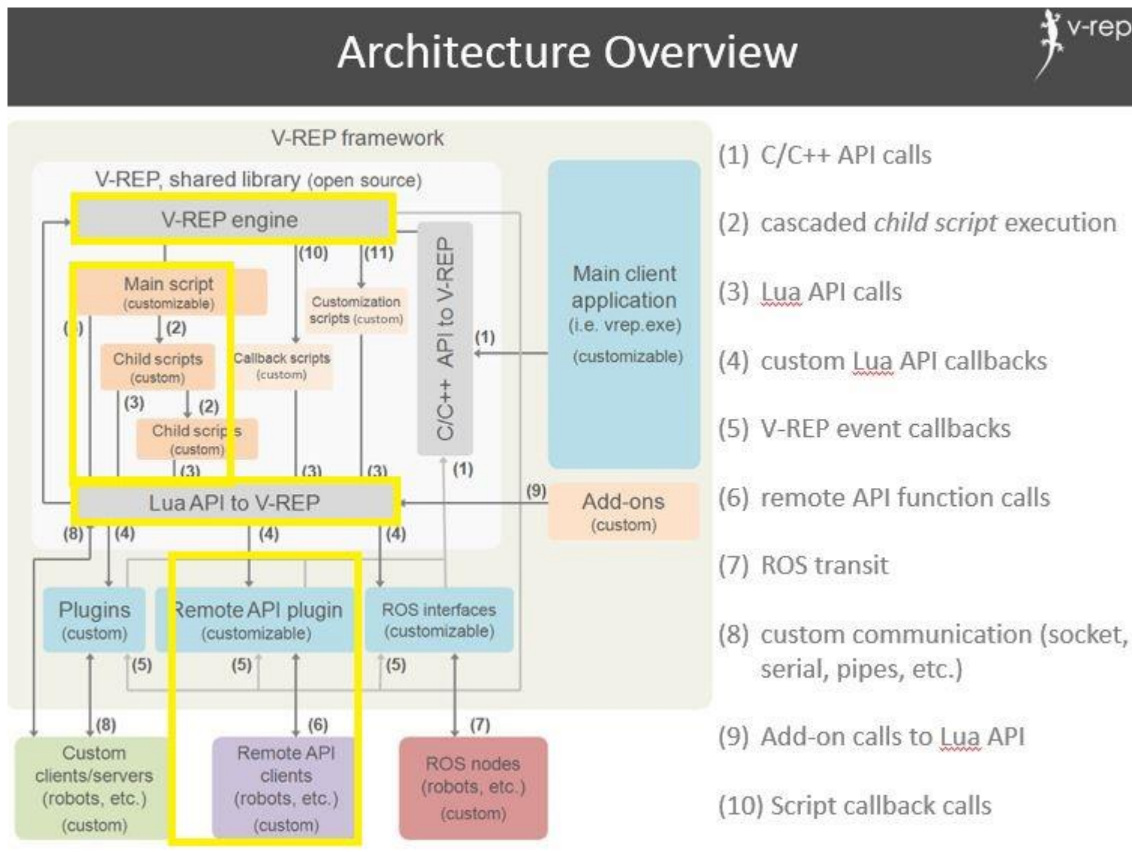


Figura 4.3 - Arquitectura V-REP

Lua se trata de un lenguaje de programación con una gran velocidad de ejecución, algo que lo hace especialmente atractivo para el desarrollo de juegos y aplicaciones móviles. Aunque destaca en su uso para el desarrollo de juegos, como cualquier otro lenguaje se puede usar para otras muchas tareas.

Este lenguaje tiene su propia página web en la que encontramos un manual detallado del lenguaje (www.lua.org). En éste se explican todos sus fundamentos y características, tales como tipos de variables, bucles, sentencias, palabras clave etc. Como hicimos con Python, no entraremos a analizar en detalle todo el lenguaje en sí, sino que dejaremos constancia de aquellas particularidades del lenguaje a tener en cuenta a la hora de implementar el proyecto. Enumerando las más importantes:

- Las variables no tienen por qué ser declaradas al principio, pudiendo inicializarse directamente cuando vayan a ser usadas. Respecto al fin de línea, podemos acabar las sentencias con un ";" pero no es obligatorio. Por comodidad, no se ha usado ese carácter a lo largo del proyecto.

- Dentro de los bucles como ***if else*** o ***for*** se pueden usar paréntesis, pero no son obligatorios. El sangrado tampoco es necesario. Aunque a priori hace la tarea de programa más cómoda, al final siempre se le acaba aplicando un margen a los distintos bucles y sentencias para poder ir siguiendo el código más cómodamente y tenerlo bien ordenado.

En cuanto a la sintaxis, Lua tiene sus palabras clave al igual que todos los lenguajes de programación. En este proyecto destaca el uso del carácter "#", el cual nos devuelve el tamaño de una lista o una table. Las sentencias condicionales tienen su propia sintaxis, sencilla y con un formato parecido al de otros lenguajes como C o Python. Se usan con frecuencia las palabras como "***and***", "***or***", "***then***", etc.

Respecto a las tablas, muy importantes a lo largo de toda la implementación de algoritmos, tendremos en cuenta dos detalles importantes: Se declaran con llaves "{...}" en vez de con corchetes "[...]". Además, a diferencia de Python, los índices empiezan en el 1, por lo que si queremos acceder al primer elemento de una tabla tendríamos que usar la sentencia "tabla[1]"

Cabe destacar que el uso de tablas en Lua está muy desarrollado, trayendo funciones para trabajar con las mismas tales como ***insert*** o ***remove***. La función ***insert*** por ejemplo tiene un parámetro en el que podemos especificar en qué posición de la tabla queremos añadir un nuevo elemento. En Python en cambio tenías que ir rellenando la variable ***ruta*** desde el final hasta el principio para posteriormente darle la vuelta. Aquí te permite ir añadiendo las nuevas celdas al principio de la lista que ira almacenando el camino calculado.

Por último, como otros muchos lenguajes Lua nos permite trabajar con funciones, a las que pasándole una serie de parámetros nos devuelve el dato o datos que hayamos calulado dentro de esa función. Sin embargo, en este proyecto prácticamente no se han usado. La estructura de los scripts en este caso hace más sencillo escribir el código paso por paso que estar llamando a funciones dentro de un mismo elemento. No obstante, si quisiéramos definir funciones como hicimos con Blender también podríamos hacerlo.

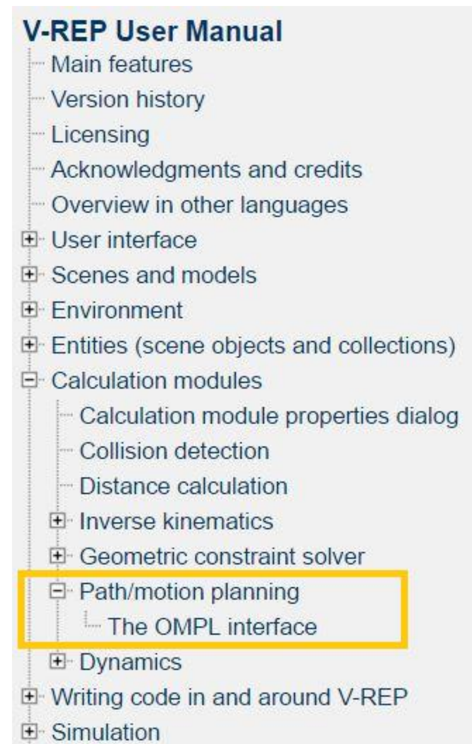
4.4. *Motion planning* dentro de V-REP

El manual de usuario de V-REP (www.coppeliarobotics.com/helpFiles/) consiste en una serie de capítulos en los que se recogen todas las funcionalidades y posibilidades que encontramos dentro del programa. Para la implementación de los algoritmos de búsqueda recurrimos a esta guía, centrándonos en aquellos capítulos que nos puedan ser útiles.

Dentro del manual hay una sección directamente relacionada con este proyecto. Se llama *Path/motion planning*, y en ella se desribe cómo V-REP ofrece funciones de búsqueda de rutas y movimientos. Esto lo hace mediante plugins que extienden y amplían la funcionalidad del programa permitiendo usar una librería OMPL (*Open Motion Planning Library*). Esta librería es un paquete software que usando algortimos de búsqueda permite planear movimientos automáticamente, independientemente del entorno en el que se encuentre.

Para comprobar el funcionamiento de este módulo del programa, examinamos las escenas que ya trae incorporadas V-REP y que consisten en la búsqueda de caminos desde un punto hasta otro evitando los obstáculos que pudiera encontrarse.

Una de las escenas es en 2D y otra en 3D. En ambas, aparece un laberinto como los que hemos empleado a lo largo de este proyecto, y una pieza que ha de desplazarse desde una configuración inicial hasta una posición final.



Esquema 4.4.1 – Manual de usuario V-REP

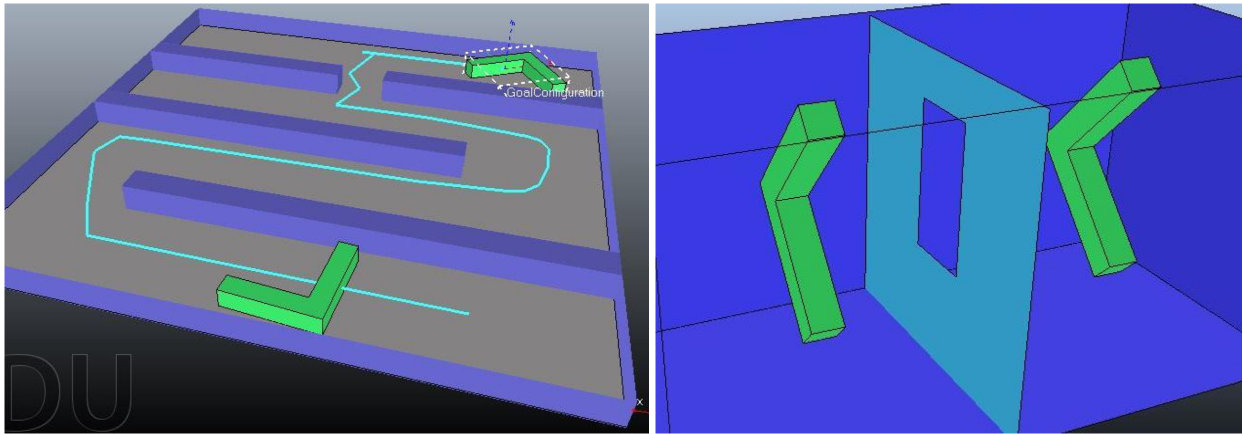


Figura 4.4.2 – Escenas *path planning* en 2D y 3D incorporadas ya en V-REP

Si ejecutamos estas escenas, vemos que efectivamente funcionan como deberían, desplazándose la figura a lo largo del laberinto de forma que esquiva todos los obstáculos hasta llegar a su destino. Analizamos por tanto los scripts que controlan el funcionamiento de estas animaciones y nos centramos en la siguiente línea:

simExtOMPL_setAlgorithm(t,sim_ompl_algorithm_RRT)

En esta línea de código, apreciamos como existe una función de V-REP que a su vez emplea la librería OMPL para el planeamiento de la ruta deseada. Si analizamos esta función, comprobamos que tiene dos parámetros; uno para indicar el objeto que ha de recorrer el camino y un segundo parámetro que especifica el algoritmo de búsqueda deseado. En el segundo parámetro, la función admite la siguiente lista de algoritmos implementados:

sim_ompl_algorithm_***BiTRRT*** sim_ompl_algorithm_BITstar sim_ompl_algorithm_BKPIECE1
sim_ompl_algorithm_CForest sim_ompl_algorithm_LazyPRMstar sim_ompl_algorithm_FMT
sim_ompl_algorithm_KPIECE1 sim_ompl_algorithm_EST sim_ompl_algorithm_LazyPRM
sim_ompl_algorithm_***LazyRRT*** sim_ompl_algorithm_***TRRT*** sim_ompl_algorithm_***LBTRRT***
sim_ompl_algorithm_PDST sim_ompl_algorithm_PRM sim_ompl_algorithm_PRMstar
sim_ompl_algorithm_ ***pRRT*** sim_ompl_algorithm_ pSBL sim_ompl_algorithm_ ***RRT***
sim_ompl_algorithm_***RRTstar*** sim_ompl_algorithm_SBL sim_ompl_algorithm_***RRTConnect***
sim_ompl_algorithm_SPARS sim_ompl_algorithm_SPARStwo sim_ompl_algorithm_STRIDE

Como podemos comprobar, el módulo para planeamiento de rutas de V-REP trae incorporados una gran cantidad de algoritmos de búsqueda. Muchos de ellos son algoritmos más enfocados a la estimación de movimientos que a la búsqueda de caminos. Esto se debe a que la mayoría están diseñados para desempeñar labores en el ámbito robótico y no en la búsqueda de caminos, que es nuestro caso.

Entre estos algoritmos no sólo encontramos el RRT, sino que además están implementadas una serie de variaciones del mismo como pueden ser el RRTstar o el LazyRRT. Por este motivo, en la implementación que llevaremos a cabo no se programará el RRT sino que nos centraremos en el diseño de los algoritmos Dijkstra y A* y su posterior representación dentro del escenario creado.

4.5. Implementación de algoritmos

Tras haber visto la interfaz sobre la que trabajaremos y algunas nociones de cómo se trabaja con el lenguaje de programación Lua, empezamos con la implementación del proyecto. Aunque en algunos puntos será parecida a la realizada con Blender, veremos que en otros aspectos difiere bastante.

Lo primero que hacemos es crear una nueva escena en la que empezamos desde cero. Cuando hacemos esto, por defecto nos aparece una escena con luces, un suelo o tablero al que le podemos modificar las dimensiones y una cámara que apunta a este tablero. Podemos elegir entre trabajar sobre este plano ya predefinido o eliminarlo y crear nuestro propio plano que actuará como base. Sobre él representaremos todo el laberinto y la búsqueda de caminos.

Por otro lado, las escenas de V-REP traen un script principal o *main script*. Este no está pensado para ser modificado, ya que sin él no sería posible llevar a cabo las simulaciones reales, consistentes en llamar a los scripts secundarios (*child scripts)*.

El código del proyecto se implementará por lo tanto en un scrip secundario. Para ello, empleamos la herramienta que nos ofrece el programa para crear scripts. Hay que aclarar que los scripts secundarios siempre van encapsulados en algun componente de la escena, por lo que creamos uno nuevo y lo asociamos a algún elemento de los que ya hay añadidos. Por comodidad, se lo podemos asociar a la cámara que nos enfoca el laberinto.

Dentro del script lo primero que hay que hacer es declarar las variables con las que trabajaremos. Entre estas variables se encuentran los colores que emplearemos para la representación. También declaramos las dimensiones del laberinto, y por supuesto el propio laberinto, generado automáticamente gracias a funciones aleatorias del lenguaje Lua.

El desarrollo del algoritmo será el mismo que el explicado en el segundo capítulo del trabajo. A su vez, seguiremos los mismos pasos que con Blender, realizando los cambios que Lua nos requiere respecto a Python. Mientras en Blender podemos definir un escenario aleatorio en una sola línea de código, en V-REP necesitamos seis líneas. El escenario seguirá viniendo representado por una matriz de unos y ceros. Una vez tenemos esta matriz, pasamos a la representación del escenario.

```
229 grid = [[[round(random.random()-.3) for z in range(dimz)] for y in range(dimy)] for x in range(dimx)]
```

```
23 for x=1,dimx,1 do
24     grid[x]={}
25     for y=1,dimy,1 do
26         grid[x][y]={}
27         for z=1,dimz,1 do
28             grid[x][y][z]=math.modf(math.random()+0.2) end end end
```

Esquema 4.4.1 – Declaración de mapa en Blender vs declaración en V-REP

Para crear una animación completamente desde cero decidimos eliminar el tablero que viene por defecto como base de la escena. Lo primero que haremos dentro de V-REP será crear un plano que nos servirá como suelo o base del laberinto. Para ello mediante la barra de cabecera podríamos hacer lo siguiente:

Add → Primitive Shape → Plane

Sin embargo, el escenario puede cambiar de dimensiones, y como a priori no sabremos cuáles son, programaremos el script para que lo cree automáticamente acoplándose a las dimensiones del escenario deseado. De esta forma no tendrá que añadirlo nadie manualmente.

En este punto nos empezamos a adentrar en la API para V-REP. En su página web, tenemos una lista de funciones que nos permiten manipular múltiples elementos dentro del software, así como acceder y modificar sus propiedades. Esta lista de funciones es muy extensa y brinda infinidad de posibilidades a la hora de programar, por lo que nos centraremos en aquellas funciones que vayan a ser útiles durante la implementación del algoritmo (http://www.coppeliarobotics.com/helpFiles/en/apiFunctionListAlphabetical.htm).

Las funciones actúan de la siguiente manera; cuando las llamamos, tenemos que especificar una serie de parámetros que modificarán el resultado de esta llamada. La página web de *Coppelia Robotics* nos indica la labor de las funciones, el significado de cada parámetro y la sintaxis que hay que seguir.

simCreatePureShape

Description	Creates a pure primitive shape. See also simCreateMeshShape, simCreateHeightfieldShape and simAddParticleObject.
C synopsis	simInt simCreatePureShape(simInt primitiveType,simInt options,const simFloat* sizes,simFloat mass,const simInt* precision)
C parameters	**primitiveType**: 0 for a cuboid, 1 for a sphere, 2 for a cylinder and 3 for a cone **options**: Bit-coded: if bit0 is set (1), backfaces are culled. If bit1 is set (2), edges are visible. If bit2 is set (4), the shape appears smooth. If bit3 is set (8), the shape is respondable. If bit4 is set (16), the shape is static. If bit5 is set (32), the cylinder has open ends **sizes**: 3 values indicating the size of the shape **mass**: the mass of the shape **precision**: 2 values that allow specifying the number of sides and faces of a cylinder or sphere. Can be NULL for default values
C return value	-1 if operation was not successful, otherwise the handle of the newly created shape
Lua synopsis	number objectHandle=simCreatePureShape(number primitiveType,number options,table_3 sizes,number mass,table_2 precision=nil)
Lua parameters	Same as C-function
Lua return values	Same as C-function

Esquema 4.4.2 – Descripción de *simCreatePureShape*

Vayamos como ejemplo con la primera función empleada, *simCreatePureShape*, la cual sirve para añadir figuras básicas dentro de la escena. En su descripción vemos que según le pasemos un número del 0 al 3 podremos crear un cubo, esfera, cilindro o cono. Además, le tendremos que especificar otros parámetros como las dimensiones o la masa. Visto esto, la línea de código dentro del script para crear el plano sería:

plano=simCreatePureShape(0, 1, { dimx , dimy , 0 }, 1)

Una vez creado el plano, pasamos a crear el resto del escenario. Para ello trabajamos de forma muy parecida a Blender. A medida que recorremos la matriz que nos hace de mapa, vamos colocando cubos en las coordenadas que ésta nos indique. Sin embargo, no nos bastará con la función vista anteriormente, pues solo se limitaba a colocar una figura y darle dimensiones.

Tendremos que recurrir por tanto a la librería de funciones para encontrar aquella función que nos permita asignarle una determinada posición a cada componente. Ésta será *simSetObjectPosition,* y sirve para especificar

las coordenadas en las que se encuentra el objeto indicado. Como con todas las funciones, en el manual se explican los parámetros y el modo de empleo. En nuestro caso, para cada cubo que compone el laberinto deberíamos llamar a dicha función e indicarle las coordenadas específicas.

simSetObjectPosition(cubo , -1 , { x , y , z })

Como comentamos anteriormente, V-REP es una herramienta mucho más sencilla y ligera a la hora de modelar y diseñar entornos. Si queremos decorar o modificar una escena nos ofrece muchas menos posibilidades, lo cual puede ser ventajoso o no en función de cual sea el objetivo final.

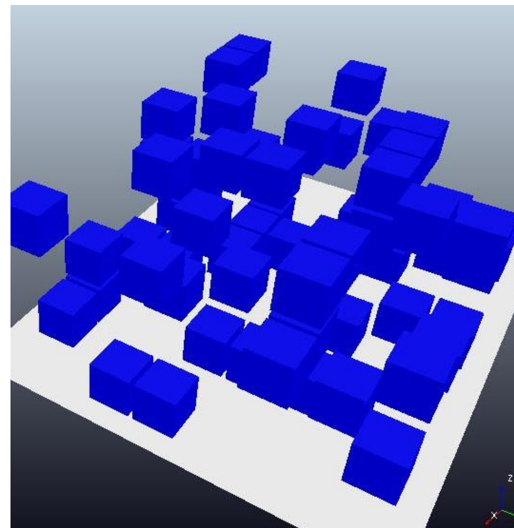


Figura 4.4.3 - Escenario

En Blender, para los elementos de la escena había que crear un material específico, con su textura, color y demás características que nos permitían llegar a conseguir resultados muy detallistas. La simplicidad de V-REP, sin embargo, hace que dentro de la librería de funciones encontremos una función que nos permite asignarle a los objetos un color directamente. Teniendo previamente definidos los colores, la función que emplearemos para decorar la escena será:

simSetShapeColor(objeto , null, 0, color)

Con el escenario ya creado se realiza la implementación del algoritmo que deseemos, tanto del Dijkstra como del A*. A la hora de programarlos, no podremos olvidar las notables diferencias entre un lenguaje y otro. Aquí se enumeran las más importantes:

- Se usan llaves "{}" en vez de corchetes "[]".
- No hace falta un sangrado específico.
- Las declaraciones son más largas y pesadas, aunque más claras también.
- Los índices de una tabla o matriz empiezan en 1 y no en 0.

Teniendo en cuenta y estos grandes detalles, podremos implementar el algoritmo paso a paso como hicimos anteriormente. V-REP nos ofrece la posibilidad de abrir una consola o ventana en la que podemos ir analizando lo que va haciendo el código. Es una herramienta muy útil a la hora de depurar el código, ya que en la pantalla se puede ir viendo cómo se rellenan las tablas y listas específicas y si lo están haciendo de la manera correcta. Para ello, la función empleada es:

consola=simAuxiliaryConsoleOpen('Cabecera de la consola' , 100 , 1)

mensaje=simAuxiliaryConsolePrint(consola , 'Mensaje o variable que desee ver \n')

Tras haber desarrollado el algoritmo de búsqueda correspondiente, nos queda una variable ***ruta*** que almacena todo el camino a seguir desde el origen hasta el destino. Recordemos que el escenario ya estaba creado, por lo

que en el momento que hemos conseguido calcular la ruta empezamos con el desarrollo de la animación.

V-REP funciona de una forma distinta a Blender en cuanto al desarrollo dinámico de las acciones. No nos permite trabajar con el concepto de *frame* en sí, por lo que hay que buscar alternativas respecto a la representación realizada en Blender. En cambio, dispone de un parámetro llamado *simulation time step* (abreviamos como "dt") que nos permite controlar la velocidad de ejecución del programa.

Por defecto, cuando ejecutamos una simulación, el script principal está llamando continuamente a los scripts secundarios. El parámetro dt nos permitirá establecer el tiempo entre estas llamadas. El tiempo funciona como un parámetro virtual, y por tanto podemos hacer que los segundos para el entorno V-REP duren lo más, menos o lo mismo que en la realidad. Supongamos que quisiéramos simular la actuación de un robot en un periodo de 10 minutos; gracias a este parámetro podríamos ver esta simulación en un intervalo de tiempo mucho menor.

Aunque esto a priori no tiene mucho que ver con la implementación de la animación, se explicará a continuación por qué es un parámetro tan importante. Para pasar de un estado a otro utilizamos una referencia temporal, asignándole a cada "segundo" un estado o posición concreta a lo largo de la ruta y dibujando así el camino entre los estados.

Para hacer esto se emplea el siguiente método. Cada vez que se llama al script en el que está implementado el algoritmo, tenemos una variable "***segundo***" que almacena el momento de esta llamada. Con el *simulation time step* controlamos cada cuánto se hace esta llamada. La variable ***segundo***, como se aprecia en el ejemplo, nos muestra el segundo con su parte entera y su parte decimal. Analizando su parte entera podemos detectar por lo tanto el cambio de un segundo a otro.

Figura 4.4.4 – Ejecucción con dt=100 ms

Aquí viene el fundamento de la animación. Para rellenar el camino, lo que hacemos es coger esa parte decimal que nos da ***segundo*** y añadirla a las coordenadas que tenemos en un estado fijo. Esto quiere decir que, si desde una posición hasta otra contigua tenemos que avanzar en el eje X, las coordenas Y y Z permanecerán fijas mientras que a la coordenada X le iremos asignando la parte decimal del tiempo. Luego, estas coordenadas se las asignamos a un sólido dentro del escenario de forma que vamos recorriendo el camino que nos indica la variable ***ruta***.

Esta es la parte más compleja del código y en un principio puede resultar complicada de entender. Sin embargo, con el siguiente ejemplo se verá mucho más clara la base del funcionamiento.

Nos encontramos en una escenario en el que la posición inicial está en las coordenadas (0 , 0 , 0), y la celda contigua a la que hay que moverse es la (1 , 0 , 0). Cuando ejecutamos la simulación, empezamos a recoger la variable ***segundo***. Supongamos que le hemos asignado un dt = 200 ms para verlo con más facilidad. ***Segundo*** iría recogiendo los siguientes datos:

Segundo = 0.00	→	Parte decimal = 0	→	Coordenada(0 , 0 , 0)
Segundo = 0.20	→	Parte decimal = 0.20	→	Coordenada(0.20 , 0 , 0)
Segundo = 0.40	→	Parte decimal = 0.40	→	Coordenada(0.40 , 0 , 0)
Segundo = 0.60	→	Parte decimal = 0.60	→	Coordenada(0.60 , 0 , 0)
Segundo = 0.80	→	Parte decimal = 0.80	→	Coordenada(0.80 , 0 , 0)
Segundo = 1.00	→	Cambio de segundo	→	Coordenada (1 , 0 , 0)

Coordenada dentro de ruta tras el cambio de segundo: (1 , 0 , 0)
Siguiente posición dentro de la variable ruta: (1 , 1 , 0)

Segundo = 1.20 → Parte decimal = 0.20 → Coordenada(1 , 0.20 , 0)
Segundo = 1.40 → Parte decimal = 0.40 → Coordenada(1 , 0.40 , 0)
Segundo = 1.60 → Parte decimal = 0.60 → Coordenada(1 , 0.60 , 0)
Segundo = 1.80 → Parte decimal = 0.80 → Coordenada(1 , 0.80 , 0)
Segundo = 2.00 → Cambio de segundo

A lo largo de la ejecución del script vamos creando una serie de coordenadas contiguas relacionadas directamente con el tiempo de ejecución, pasando a la siguiente coordenada dentro de ***ruta*** cuando se cambia de segundo. Atendiendo a la parte decimal, vamos recogiendo los valores y se los vamos añadiendo a la coordenada en cuya dirección hay que desplazarse. A medida que pasan los segundos rellenamos las coordenadas con las figuras que deseemos vayan apareciendo en la ruta.

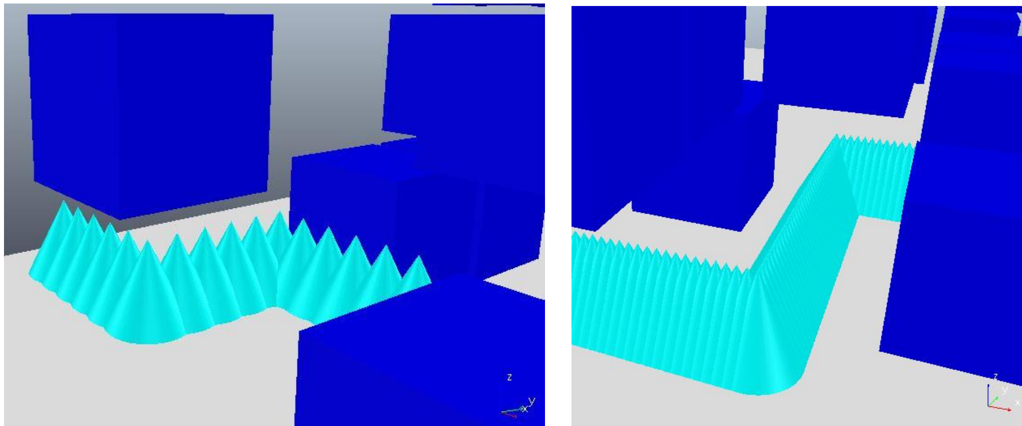


Figura 4.4.5. – Animación con dt=200 ms y dt=50 ms respectivamente

Cada vez que obtenemos una nueva posición, lo que hacemos es colocar un sólido en esa coordenada de forma que vamos rellenando el camino calulado previamente mediante nuestro algoritmo de búsqueda. Si quisiéramos rellenarlo con conos, nos quedaría un camino de conos desde el origen hasta la posición destino. Sin embargo, parece más razonable rellenarlo con esferas al tratarse de una figura simétrica y que se aprecia igual desde cualquier ángulo.

De la misma foma que hicimos con Blender, al sólido encargado de ir definiendo la ruta desde el origen hasta el destino le asignamos un color llamativo, de forma que destaque a lo largo del laberinto. Cuando ha recorrido todo el camino, mediante la siguiente línea de código damos por concluida la animación.

simPauseSimulation()

En el siguiente capítulo se recogen capturas de todo el trabajo realizado. En la parte de V-REP, se apreciará cómo podemos jugar con las distintas figuras que van definiendo el camino, así como con los colores para los distintos componentes de la animación. Además, todo el código implementado en este software se recoge en el Anexo B.

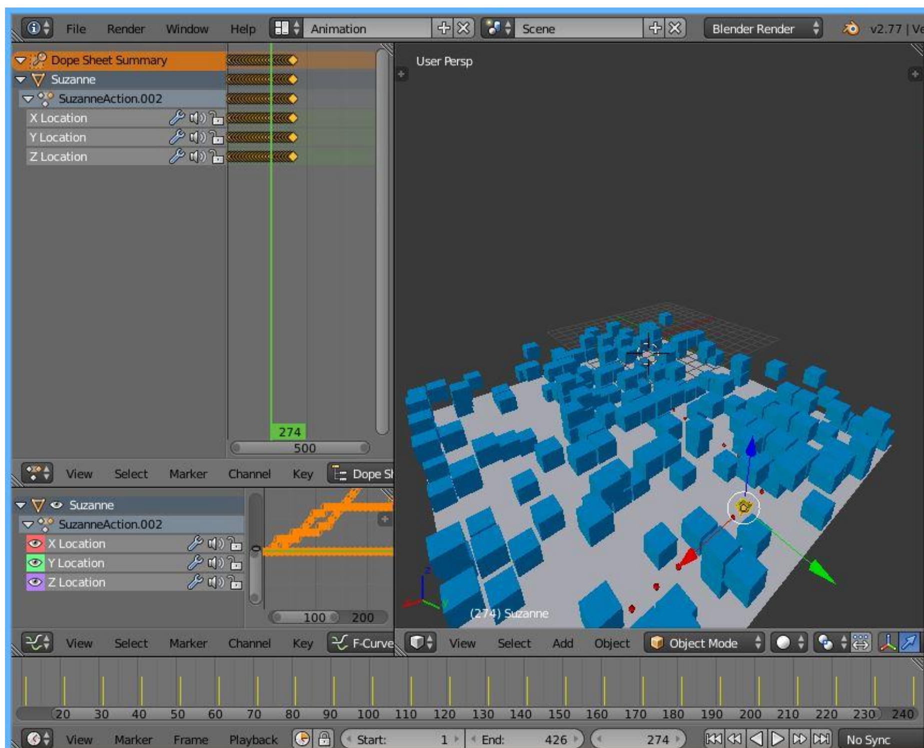
5 CAPÍTULO DE RESULTADOS: ANIMACIONES EN BLENDER Y V-REP

Tras desribir los algoritmos de búsqueda de caminos, vimos dos softwares de animación 3D en los que estos algoritmos podían ser implementados. Además, tanto con Blender como con V-REP disponemos de herramientas que nos permiten realizar animaciones con las que poder escenificar el trabajo realizado de forma mucho más clara y atractiva.

Estas animaciones nos ayudan a representar tridimensionalmente y de forma animada los resultados calculados teóricamente. A lo largo de este capítulo aparecen una serie de pantallazos en los que se aprecia el resultado final del proyecto en ambos entornos de trabajo.

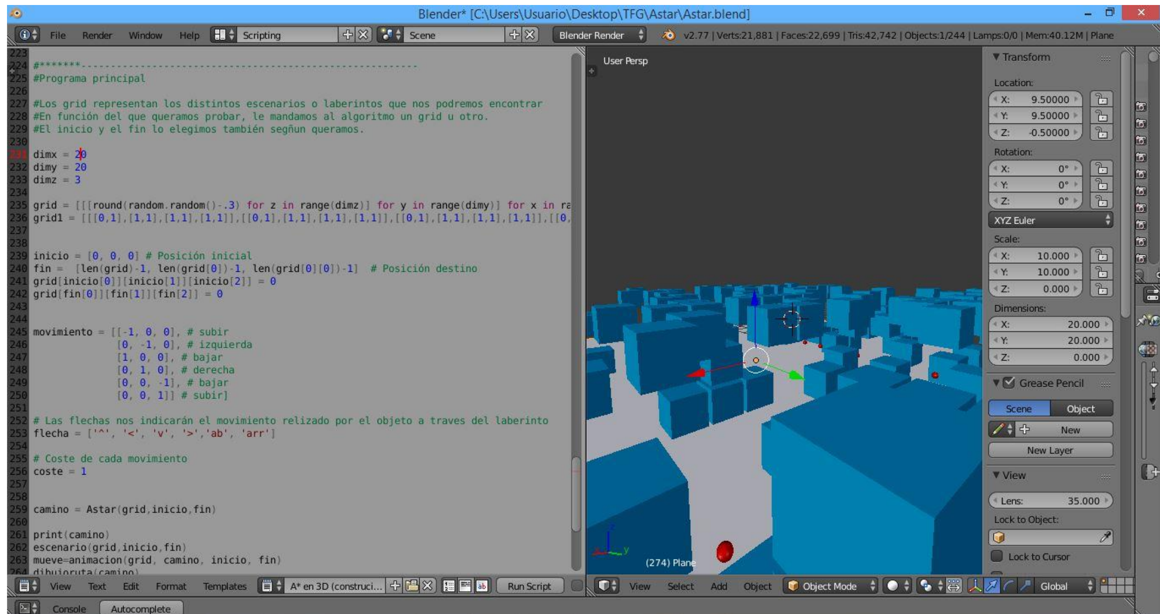
5.1. Resultados con Blender

En dos dimensiones podemos representar escenarios como los vistos en los ejemplos del capítulo 2. Tras el cálulo del camino más corto desde un punto hasta otro, tenemos animaciones en las que vemos como nuestro sujeto, el monito, recorre el laberinto esquivando obstáculos.



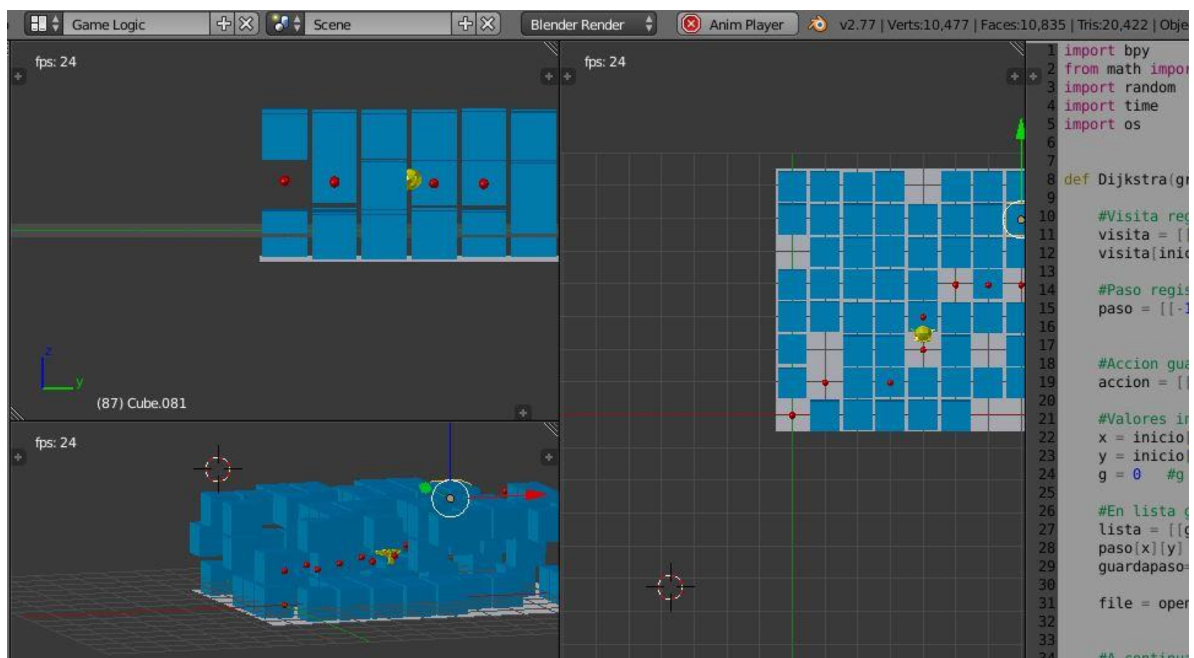
Animación 5.1.1

Aquí vemos la escena junto a la ventana de scripting, en la que mediante código escrito controlamos todos los parámetros de la animación.

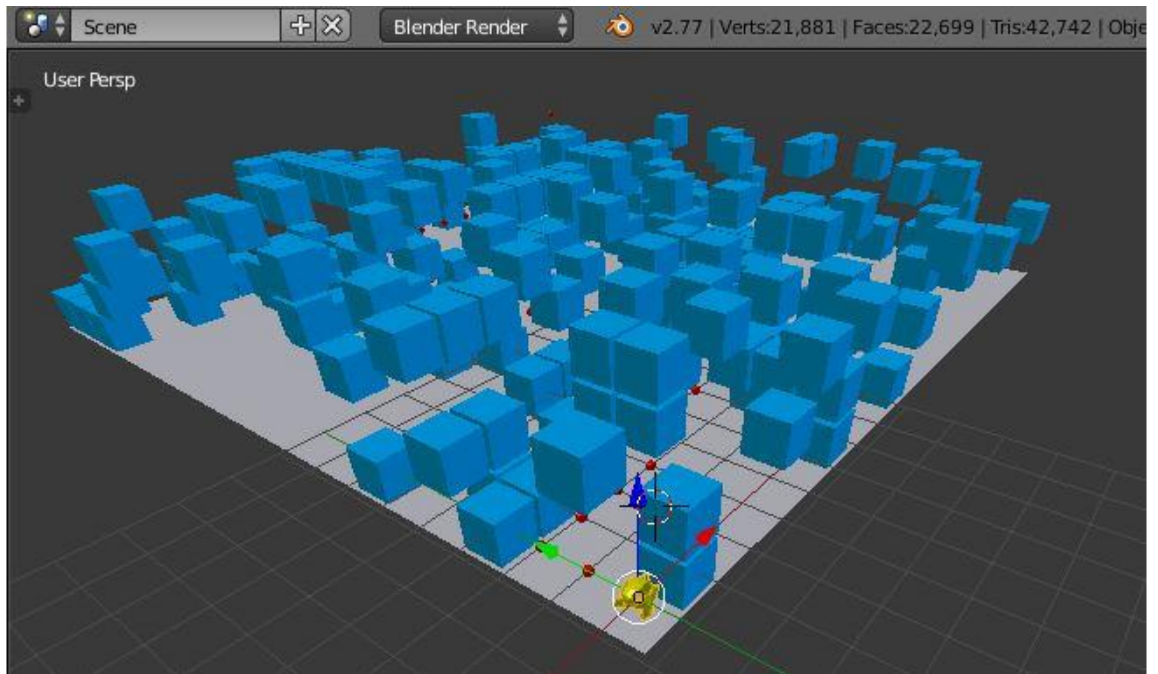


Animación 5.1.2

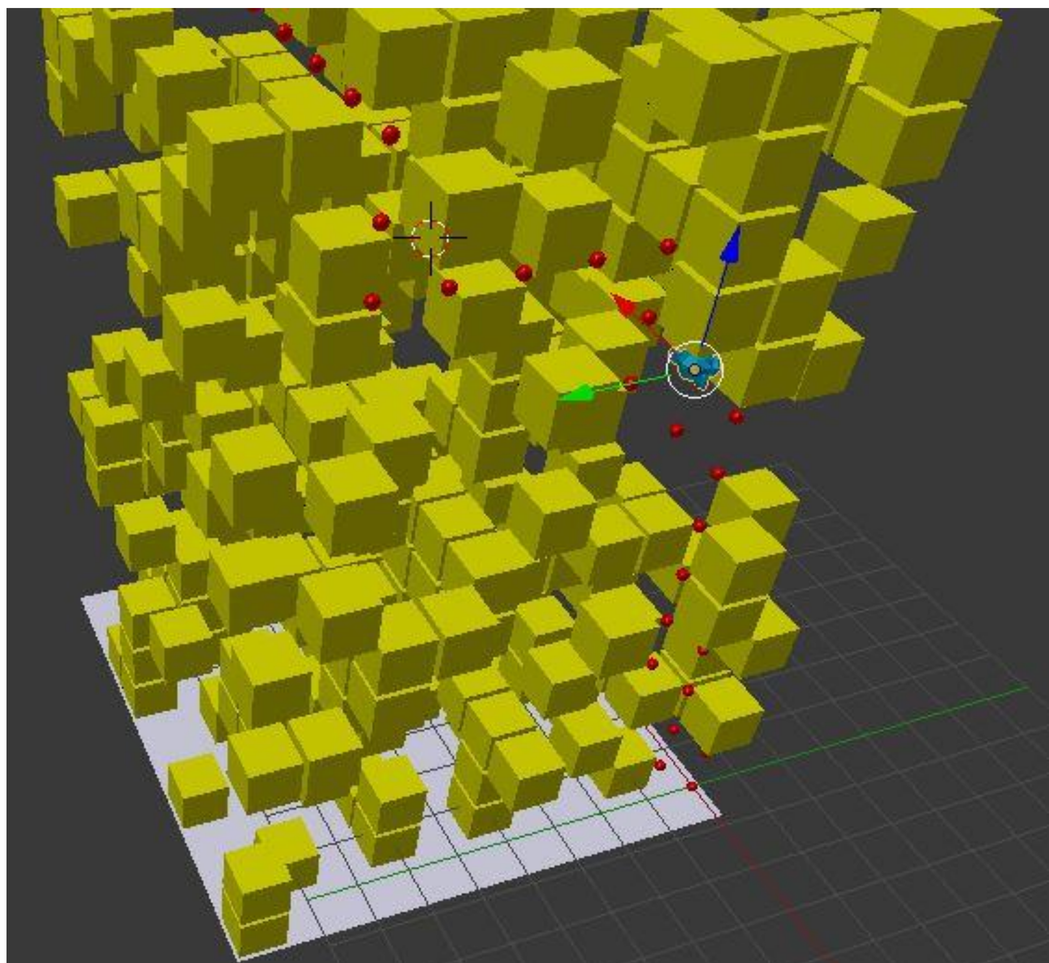
Pasando a una escena tridimensional, le vamos metiendo altura al laberinto creado y comprobando como el comportamiento es el adecuado. Para ello podemos ayudarnos de un espacio predefinido de trabajo, *Game Logic*, el cual nos permite hacer un seguimiento de la animación desde distintas perspectivas.



Animación 5.1.3

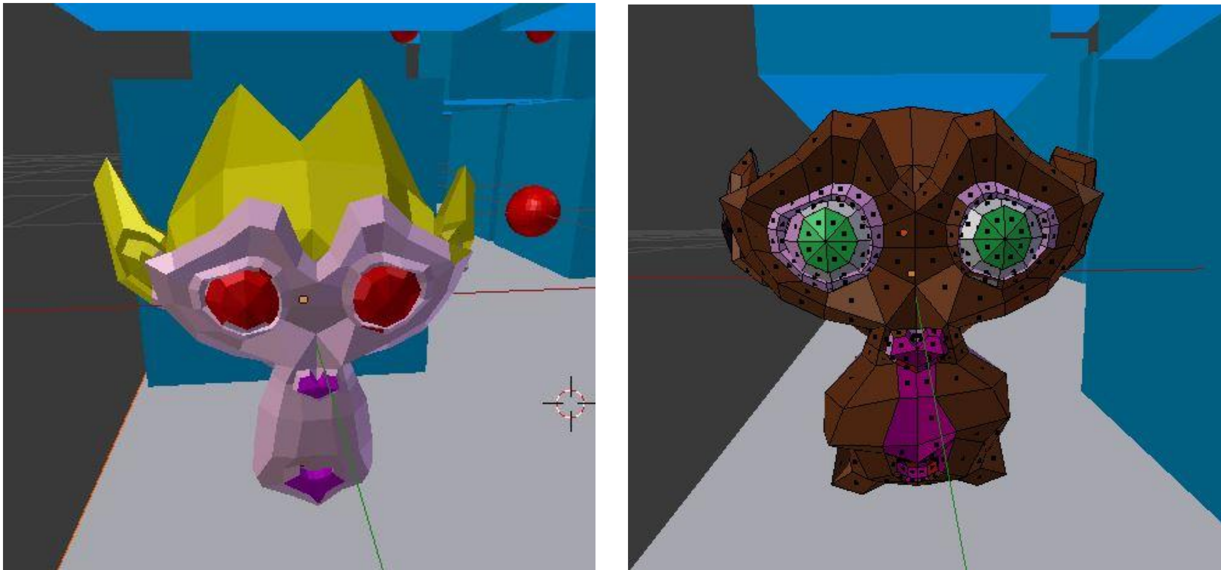


Animación 5.1.4



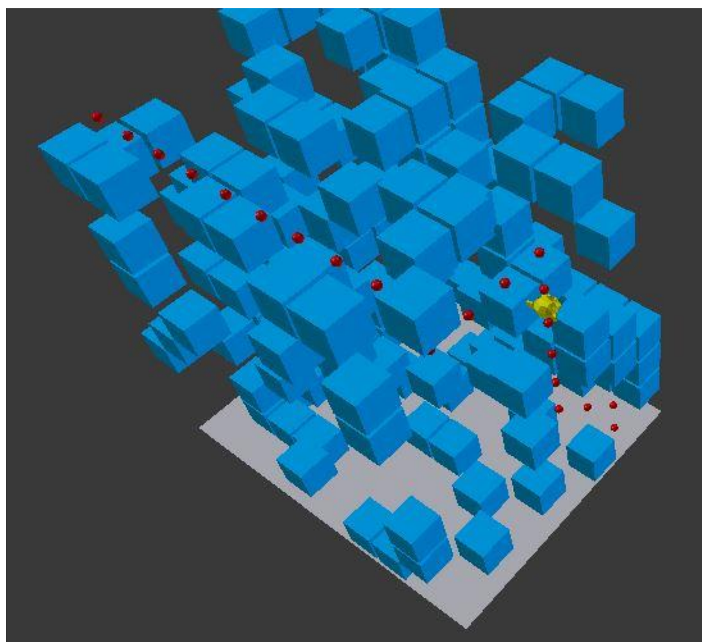
Animación 5.1.5

Blender tiene herramientas de modelado muy potentes, por lo que si quisiéramos podríamos retocar el objeto que recorre el laberinto de muchas maneras. Aquí vemos cómo podemos modificar la figura que recorrerá el laberinto, modelando y asignándole materiales a sus distintos componentes.

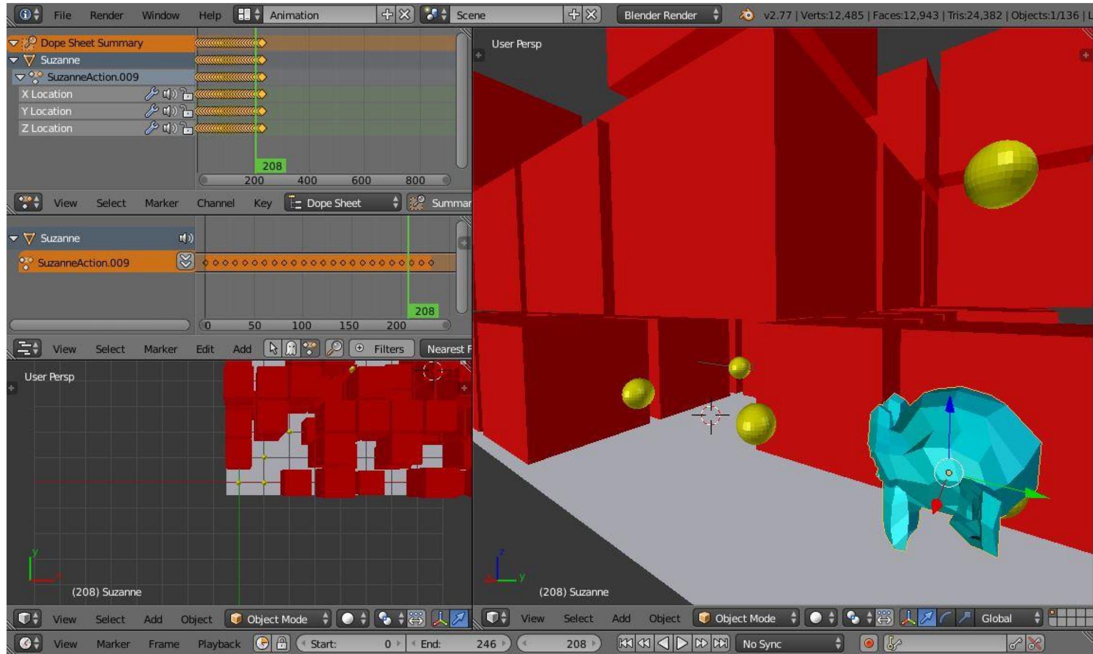


Animación 5.1.6

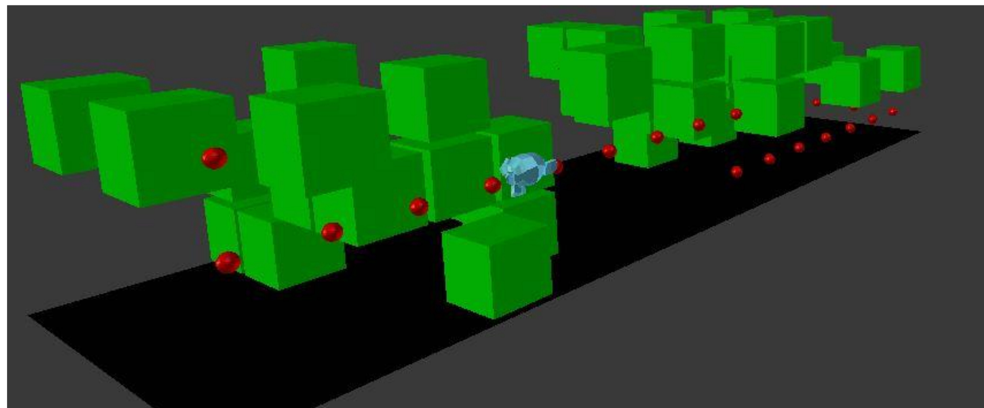
Al igual que el mono, podríamos retocar cada componente de la animación a nuestro gusto. Aunque este proyecto no está enfocado al modelado de objetos ni a a la representación de escenas, sí que es bueno saber que el programa tiene herramientas de alta complejidad con las que se puede hacer prácticamente de todo. En nuestro caso, podemos modificar los escenarios a nuestro gusto simplemente asignándoles colores distintos a los materiales de cada elemento. Esto se puede hacer directamente desde el módulo de scripting.



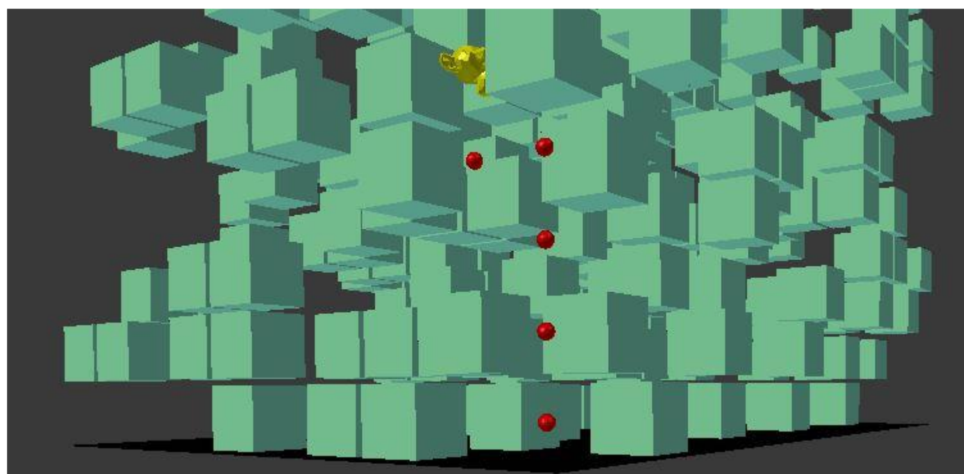
Animación 5.1.7



Animación 5.1.8



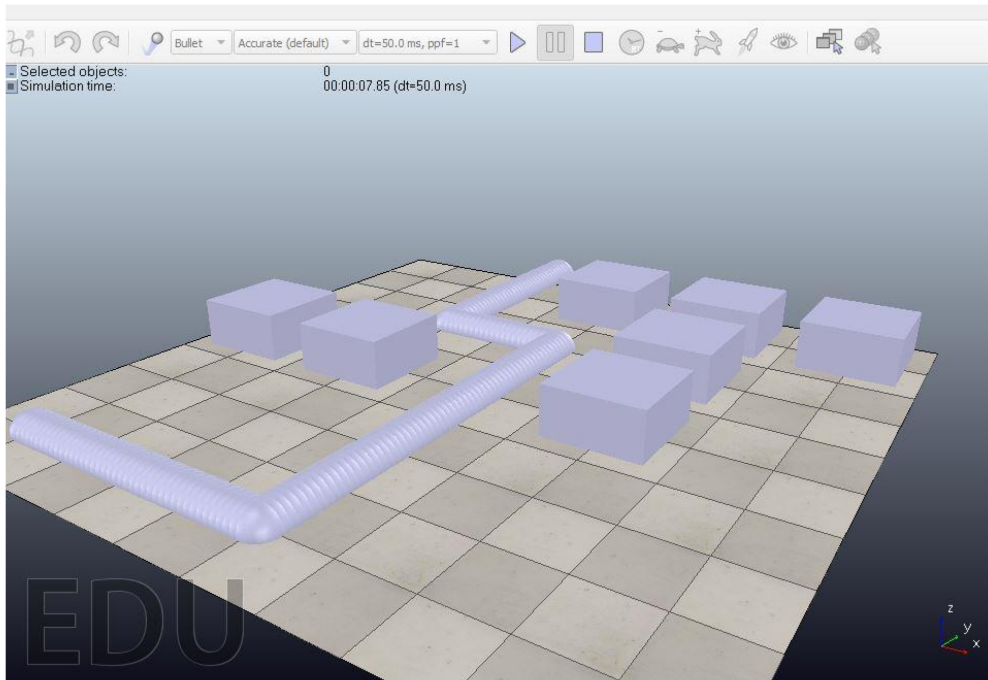
Animación 5.1.9



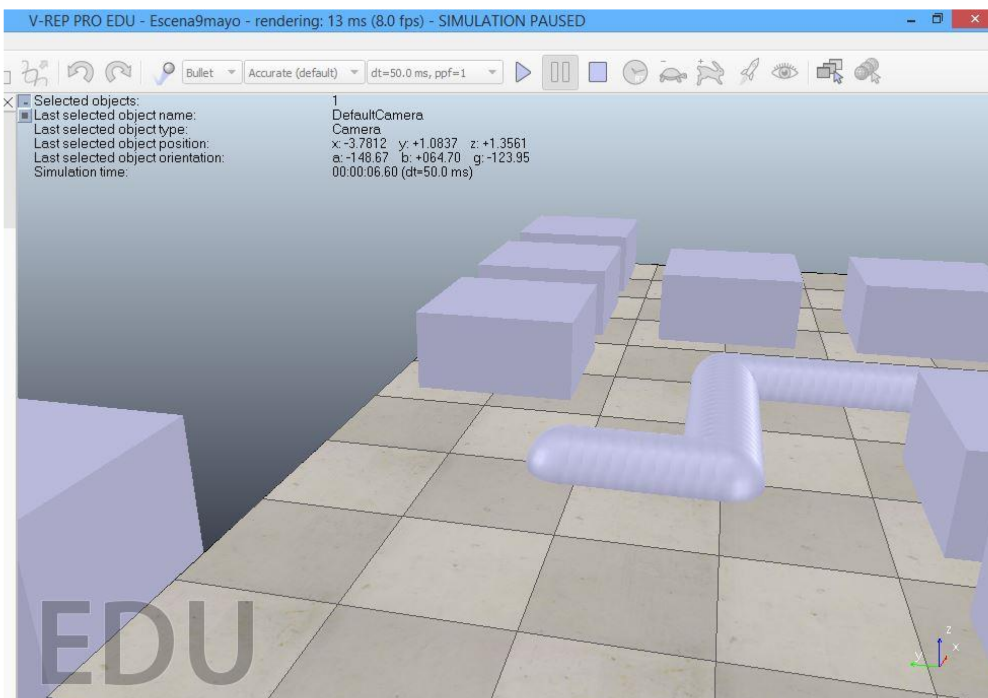
Animación 5.1.10

5.2. Resultados con V-REP

Como se vio en el capítulo dedicado a V-REP, las escenas en este software traen implementado un tablero por defecto sobre el que podríamos representar la escena. En un principio, la resolución de algoritmos en 2D se realizó sobre este tablero.

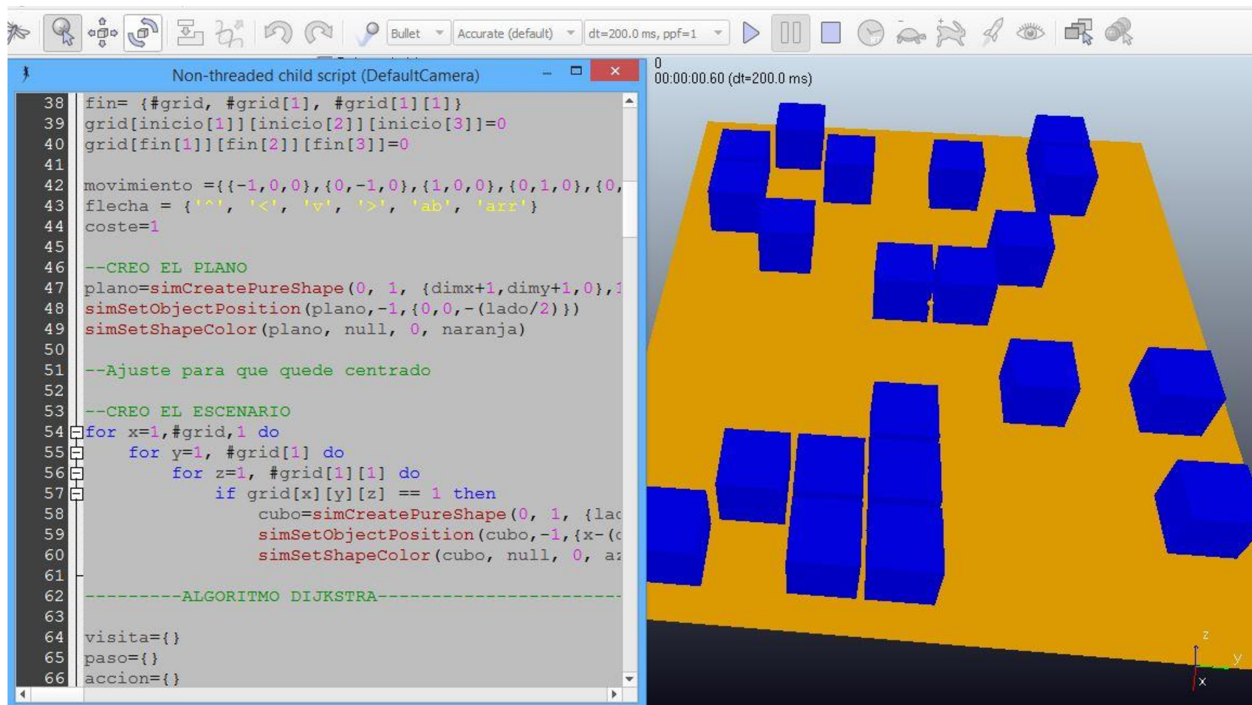


Animación 5.2.1



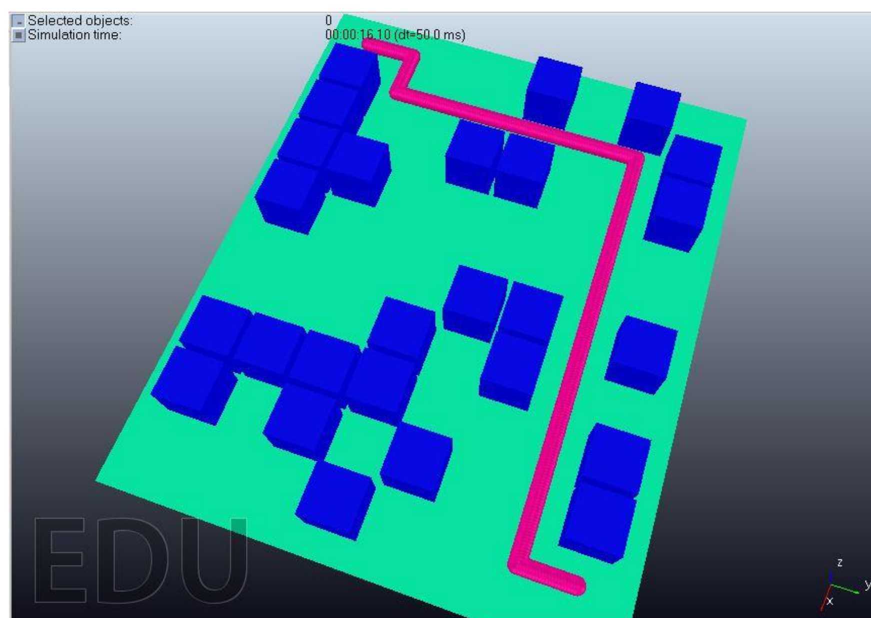
Animación 5.2.2

Sin embargo, para tener un escenario íntegramente creado por nuestro script, decidimos quitar este tablero y crear uno que se adaptara a cualquier posible tamaño del laberinto. De esta forma cada vez que ejecutemos el script se creará un tablero con las medidas específicas del escenario.



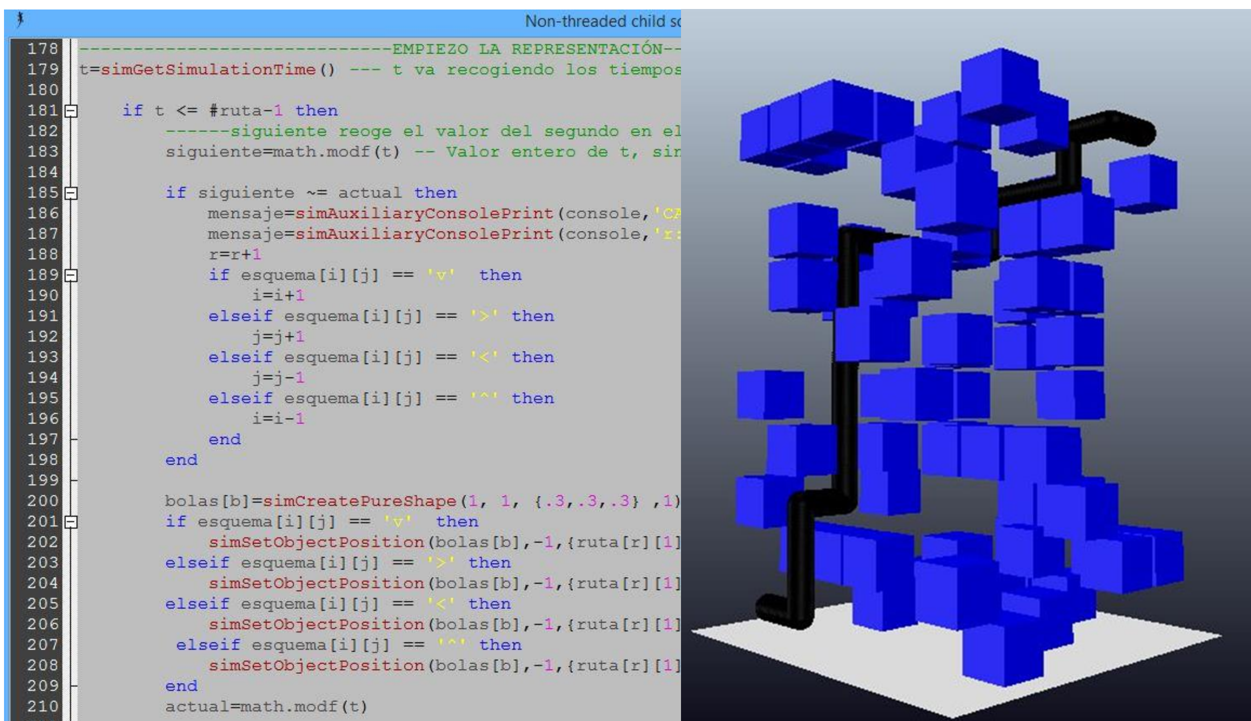
Animación 5.2.3

Para una primera implementaión del algoritmo, primero lo hacemos en 2D y comprobamos la correcta resolución de la búsqueda de caminos. Posteriormene lo implementamos también sobre el eje Z para hacer que esta búsqueda sea tridimensional.



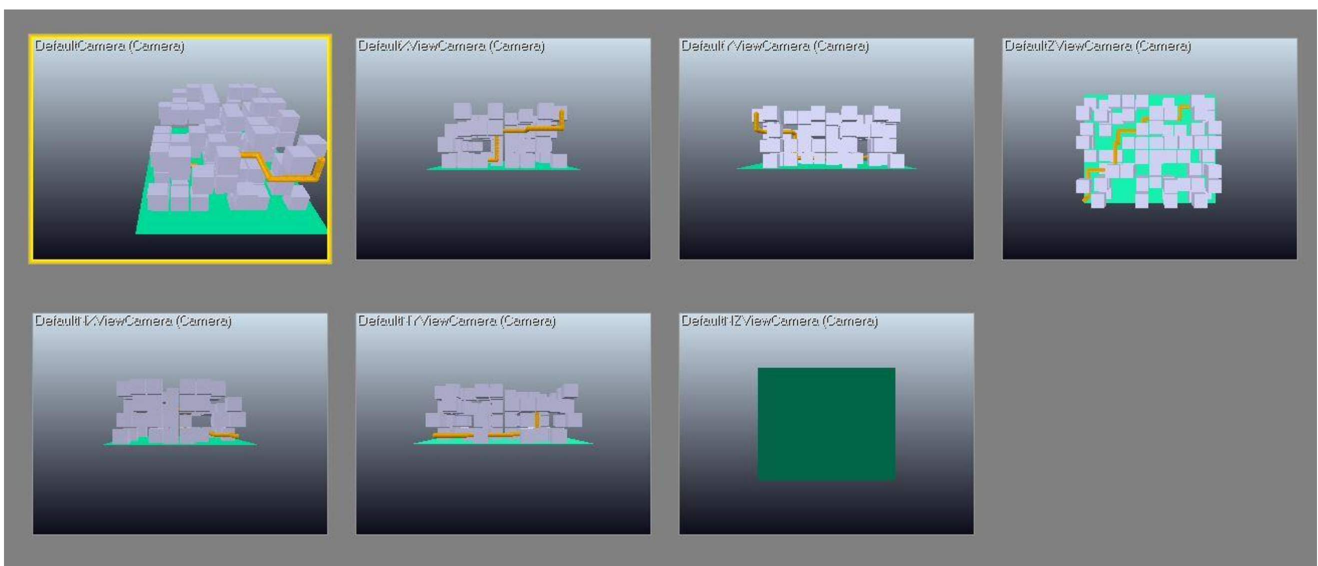
Animación 5.2.4

Mediante el script podemos ir definiendo la búsqueda de caminos y la animación, controlando todos los parámetros que van apareciendo en la escena. En el Anexo B aparece todo el código implementado.



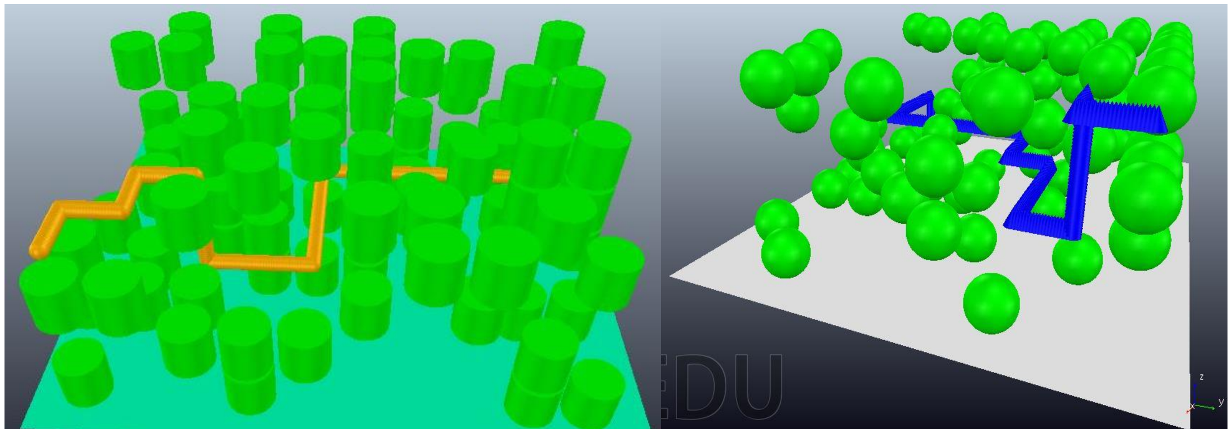
Animación 5.2.5

Además, en V-REP también tenemos un espacio de trabajo desde el que seguir la representación de la escena desde distintas perspectivas.

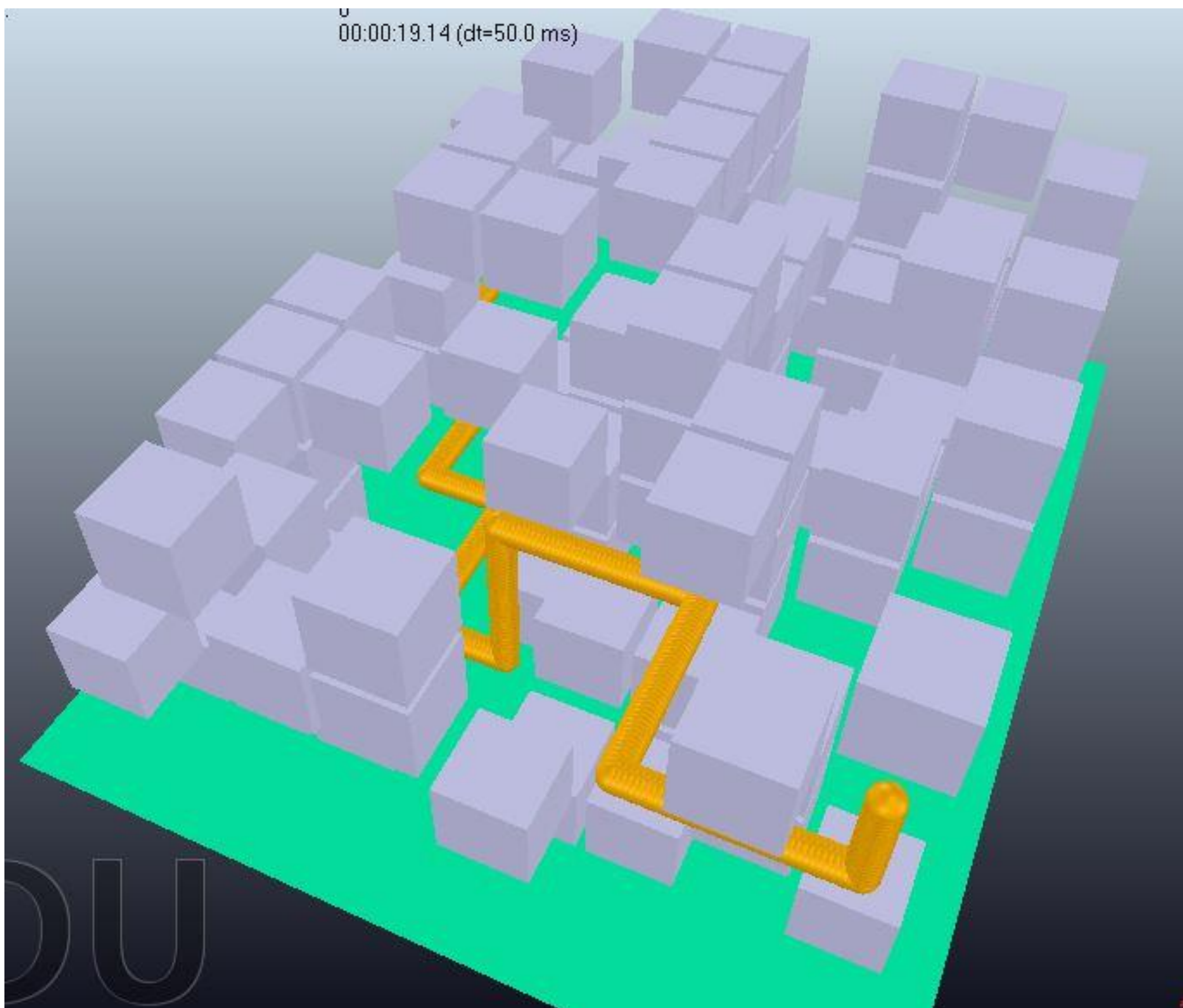


Animación 5.2.6

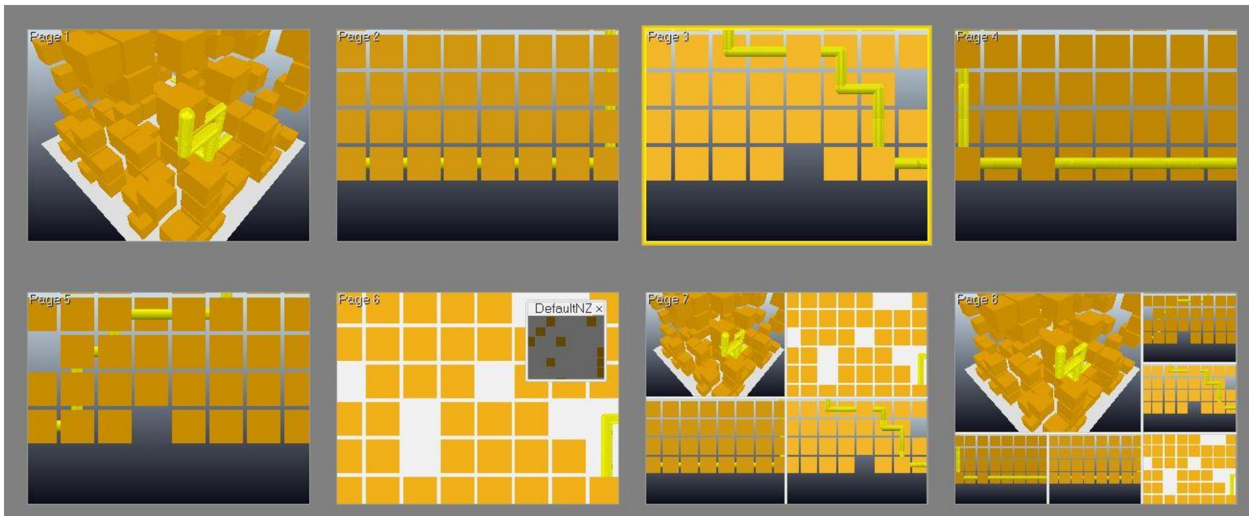
Definiendo colores al principio del código podremos moldear el escenario como vayamos queriendo. También podemos elegir distintas figuras para moldear el escenario a nuestro gusto. Aunque lo más lógico es representar un obstáculo con un cubo, podríamos seleccionar otro tipo de figura para definir el laberinto.



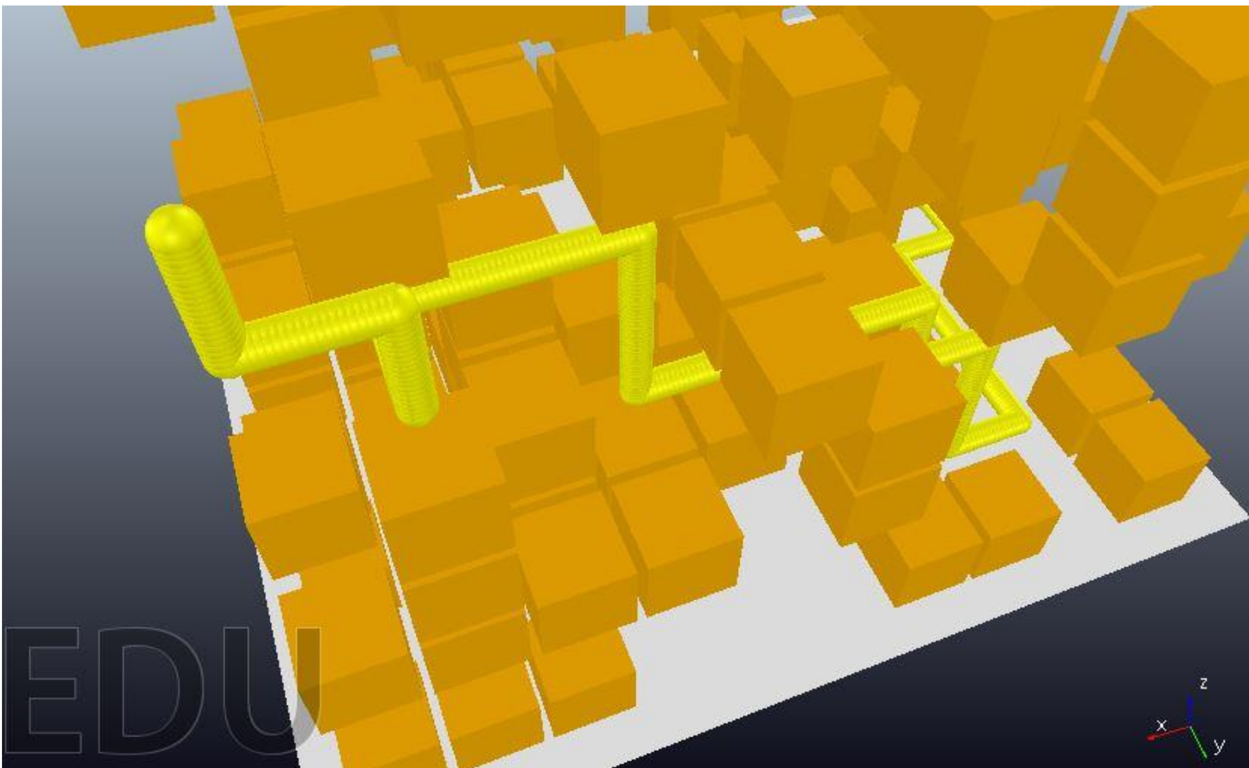
Animación 5.2.7



Animación 5.2.8



Animación 5.2.9



Animación 5.2.10

6 CONCLUSIONES FINALES Y FUTURAS LÍNEAS DE INVESTIGACIÓN

En este último capítulo se exponen las conclusiones finales extraídas tras la realización del proyecto, así como las dificultades surgidas durante el mismo. Por otro lado, se enumeran una serie de líneas de investigación sobre las que se podría seguir investigando y desarrollando este trabajo en el futuro.

6.1. Conclusiones

Este proyecto ha tratado de abordar la implementación de los algoritmos de búsqueda que sirven como herramienta para generar animaciones en entornos 3D. El objetivo principal era estudiar y comprender la estructura de los algoritmos para posteriormente modelarlos e implementarlos en softwares de animación tridimensionales.

A lo largo del trabajo se han ido encontrado dificultades fruto del desconocimiento inicial de ambos entornos de creación tridimensional, en especial con Blender. En un principio parece un programa demasiado complejo como para empezar desde cero a implementar un proyecto. Sin embargo, una vez entendido el funcionamiento básico y tras tiempo de práctica, se puede empezar a manejar con relativa soltura. Además, el lenguaje Python, con el que apenas había trabajo previamente, es bastante sencillo de entender y tiene una sintaxis simple y clara.

Por otro lado con V-REP las dudas han sido bastante más difíciles de resolver. Aunque tiene un foro en el que empleados de *Coppelia Robtotics* atienden a cualquier tipo de consulta, la comunidad que usa este programa es muy reducida. De hecho, en internet apenas se encuentran modelos o trabajos implementados dentro de este software, por lo que para resolver ciertos problemas les he tenido que dar muchas vueltas. Además, el lenguaje Lua me resultó bastante más incómodo que Python en muchos aspectos.

Con la resolución de algoritmos en 2D he conseguido entender el funcionamiento y la estructura de los distintos algoritmos de búsqueda. Entre estos, el más fiable tras las pruebas realizadas parece el A*. Sin embargo, para llevarse a cabo hace falta definir la función heurística, por lo que no siempre será posible implementarlo. En muchos casos, no será posible desarrollar esta función, por lo que tendremos que elegir otro algoritmo. El Dijkstra parece mejor en espacios de dimensiones reducidas, mientras que el RRT da resultados muchos más rápidos y eficientes cuanto mayor se hace el escenario.

Por el camino, no solo he aprendido a implementar algoritmos en 2D antes de pasar a la parte tridimensional. Ante la necesidad de crear numerosas figuras y esquemas para la explicación y desarrollo de cada uno de ellos, he tenido que aprender a usar la plataforma Pygame. Esta herramienta permite, mediante el lenguaje Python, crear juegos en 2D de manera sencilla. Para ello, combinando el lenguaje Python y el modulo de scripting de Blender podemos exportar las variables necesarias para posteriormente realizar la animación llamando a Pygame. En el Anexo C está todo el código implementado para la realización de las distintas animaciones en

2D, las cuales se han usado para ilustrar todo el Capítulo 2.

Otra de las conclusiones que he extraído del trabajo es la enorme utilidad que pueden tener las APIs para distintos programas informáticos. Si se van entendiendo los conceptos básicos, así como el uso de las librerías específicas de cada API, la interactuación con las plataformas de animación 3D se hace mucho más sencilla. Además, tanto en V-REP como Blender, el abanico de posibilidades que nos ofrecen estas interfaces de programación es enorme.

6.2. Líneas futuras

A continuación se sugieren algunos aspectos sobre los que continuar con el desarrollo del proyecto. Utilizando como punto de partida el trabajo expuesto a lo largo de esta memoria, podrían ser objeto de interés las siguientes líneas de investigación:

- Se podrían diseñar animaciones más vistosas y profesionales, añadiendo elementos que, aunque no fueran estrictamente necesarios, le fueran dando forma a la escena. Con Blender y V-REP podemos conseguir resultados totalmente realistas. El modelado de escenarios es un ámbito en el que podríamos seguir practicando e investigando todo lo que quisiéramos.

- Tras haber explicado los tres algoritmos de búsqueda, vimos una comparación de los mismos. Una línea de investigación que sería conveniente continuar es la eficiencia de los algoritmos en según qué escenarios. En este proyecto se han comparado escenarios del rango de las 100 a las 10.000 celdas. Sin embargo, podría seguirse esta investigación para escenarios muchos mayores, del rango de las 100.000 celdas o incluso mayores.

- Sería interesante trabajar en el diseño de escenarios mucho más complejos, organizando todo el espacio a examinar en secciones o bloques. Así, en vez de usar una matriz de datos, trabajaríamos con varias matrices de distintas dimensiones que funcionaran como un solo bloque. De esta forma, una vivienda por ejemplo vendría representada por un conjunto de matrices, en el que cada matriz representaría por una determinada habitación o espacio. El conjunto de varias matrices a su vez representaría un plano completo de la casa, y nuestro sujeto sería capaz de buscar caminos a lo largo de las distintas secciones del mapa.

- Otra posible línea de trabajo sería el modelado con el editor de video. Podría crearse una escena en la que mediante cámaras e iluminación pudiéramos hacer un seguimiento en primera persona del sujeto que va recorriendo el laberinto. Además, Blender nos permite implementar y trabajar con el equipo de sonido, por lo que a las animaciones vendrían acompañadas de banda sonora si así lo quisiéramos.

- Puesto que ya están explicados los algoritmos Dijkstra, A* y RRT, sería razonable continuar con otros algoritmos de búsqueda distintos. A parte de algunos ya mencionados durante el trabajo como pueden ser el D* o las variantes del RRT, hay otros muchos algoritmos de búsqueda que podrían ser abordados. El algoritmo de Bellman – Ford o el algoritmo de Viterbi son ejemplos con los que podríamos seguir ampliando el proyecto.

- Podría ser interesante implementar esos algoritmos de búsqueda en robots dentro de V-REP. Como vimos, V-REP viene con prototipos de robots móviles controlados mediante scripts. Podríamos investigar en la modificación de estos scripts de forma que estos robots, en vez de hacer el movimiento que traen por defecto, hicieron el camino calculado previamente con el algoritmo de búsqueda implementado.

- Respecto al código desarrollado con la herramienta Pygame, parece una buena idea el seguir trabajando en él hasta convertirlo en una especie de aplicación o juego. La aplicación consistiría en que, tras seleccionar con nuestro ratón las posiciones origen y destino dentro del laberinto, se desarrollara el algoritmo de búsqueda que especificáramos y lo viéramos interactivamente.

- El planeamiento de rutas autónomo para vehículos es algo en lo que muchas marcas de coche están trabajando actualmente. La idea del coche autónomo no es una utopía, y hoy en día se pueden encontrar prototipos de prueba en ciertas marcas del mercado automovilístico. Esto está relacionado directamente con la resolución de algoritmos en tiempo real. Aunque parece un campo más complejo de abarcar, podríamos investigar situaciones en las que el entorno o mapa va cambiando mientras se ejecuta el algoritmo.

- Otra idea interesante sobre la que seguir trabajando es la de añadirle movimientos diagonales al sujeto que recorre el laberinto. Así, en 2D por ejemplo, en vez de 4 movimientos podríamos realizar hasta 8. Otra variación podría ser la de variar los costes de las diferentes direcciones y apreciar cómo se ejecutarían los algoritmos en ese caso.

6.3. Comentarios finales

Abordar este proyecto ha supuesto un gran reto para mí, y aunque algunos resultados podrían haber sido mejores, se han cumplido los objetivos fijados.

Gracias a este trabajo he aprendido a estudiar y desarrollar conocimientos de forma autodidacta, aprendiendo a manejar dos herramientas que hasta entonces eran desconocidas para mí. A su vez, he aprendido documentarme mucho mejor, buscando y filtrando información tanto en libros como en internet.

Esta aplicación puede ayudar a entender el funcionamiento de los algoritmos de búsqueda, además de servir como primera toma de contacto para aquellos que quieran implementarlos en algún entorno de desarrollo tridimensional.

Aunque en este trabajo no se ha entrado en los detalles complejos de cada lenguaje, si que sirve como primera toma de contacto con Python y Lua, ya que proporciona cierta base a la hora de trabajar con ellos.

En cualquier caso, considero que los resultados del proyecto en general han sido buenos. He adquirido una gran cantidad de conocimientos respecto a Blender y V-REP, así como los lenguajes de programación empleados por estos programas. Estoy seguro de que estos conocimientos me serán de utilidad en el futuro.

REFERENCIAS

[1] Steven M. LaValle. '*Planning algorithms*'. Published by Cambridge University Press, 2006

[2] Steven M. LaValle. *'Tutotial Motion Planning - Part I: The Essentials*'. IEEE Robotics and Automation Magazine, marzo 2011

[3] Nikolaus Correll. '*Introduction to Autonomous Robots. Lesson 4: Path planning'.* First edition, abril 2006.

[4] Página web de Blender: https://www.blender.org/

[5] Página web de Coppelia Robotics: http://www.coppeliarobotics.com/

[6] Manual de referencia de Lua: https://www.lua.org/manual/5.1/es/manual.html

[7] Dave Ferguson, Maxim Likhachev and Anthony Stentz. *'A Guide to Heuristic-based Path Planning'.* School of Computer Science Carnegie Mellon University Pittsburgh, PA, USA 2005.

[8] Amit Patel, blog personal: http://www-cs-students.stanford.edu/~amitp/, así como su página web: http://www.redblobgames.com/

[9] Luis Miguel Sánchez Brea. '*Manual python: PIMCD2013-python*'

[10] Tutoriales en la plataforma *youtube.com:*
https://www.youtube.com/user/entivoo/playlists
https://www.youtube.com/watch?v=OfpB87pRoUk

[11] Steven M. LaValle, *Rapidly-exploring random trees: A new tool for path planning*. Iowa State University, Octubre 1998

[12] Buniyamin N., Wan Ngah W.A.J., Sariff N., Mohamad Z. '*A Simple Local Path Planning Algorithm for Autonomous Mobile Robots'*. Internationa journal of systems applications, engineering & developmente, volume 5, 2011

[13] Sebastian Thrun. '*Artificial intelligence for robotics, programming a robotic car. Lesson 4: search*'

Anexo A. Códigos para Blender: Algoritmos y animaciones

A.1. Algoritmo Dijkstra en 3D

```
def Dijkstra(grid, inicio, fin):

    visita = [[[0 for z in range(len(grid[0][0]))] for y in range(len(grid[0]))] for x in range(len(grid))]
    visita[inicio[0]][inicio[1]][inicio[2]] = 1
    paso = [[[-1 for z in range(len(grid[0][0]))] for y in range(len(grid[0]))] for x in range(len(grid))]
    accion = [[[-1 for z in range(len(grid[0][0]))] for y in range(len(grid[0]))] for x in range(len(grid))]

    x = inicio[0]
    y = inicio[1]
    z = inicio[2]
    g = 0

    lista = [[g, x, y, z]]
    paso[x][y][z]= g

    completada = False
    fracasada= False
    contador = 0

    while not completada and not fracasada:
        if len(lista) == 0:
            fracasada = True
            print("Camino no encontrado")
        else:
            lista.sort()
            lista.reverse()
            siguiente = lista.pop()
            x = siguiente[1]
```

```
        y = siguiente[2]
        z = siguiente[3]
        g = siguiente[0]

        paso[x][y][z]= contador
        contador += 1

        if x == fin[0] and y == fin[1] and z == fin[2]:
            completada = True
        else:
            for i in range(len(movimiento)):
                x2 = x + movimiento[i][0]
                y2 = y + movimiento[i][1]
                z2 = z + movimiento[i][2]

                if x2 >= 0 and x2 < len(grid) and y2 >=0 and y2 <len(grid[0]) and z2 >=0 and z2 <len(grid[0][0]) :
                    if visita[x2][y2][z2] == 0 and grid[x2][y2][z2] == 0:
                        g2 = g + coste
                        lista.append([g2, x2, y2, z2])
                        visita[x2][y2][z2] = 1
                        accion[x2][y2][z2] = i

esquema = [[['' for z in range(len(grid[0][0]))] for y in range(len(grid[0]))] for x in range(len(grid))]
esquema[fin[0]][fin[1]][fin[2]]='*'

ruta=[]
ruta.append([fin[0], fin[1], fin[2]])

while x != inicio[0] or y != inicio[1] or z != inicio[2]:
    x2 = x - movimiento[accion[x][y][z]][0]
    y2 = y - movimiento[accion[x][y][z]][1]
    z2 = z - movimiento[accion[x][y][z]][2]
    esquema[x2][y2][z2] = flecha[accion[x][y][z]]
    x=x2
    y=y2
    z=z2
    ruta.append([x2, y2, z2])
```

```
    ruta.reverse()

    return ruta, fracasada
```

A.2. Algoritmo A* en 3D

```
def Astar(grid, inicio, fin):

    visita = [[[0 for z in range(len(grid[0][0]))] for y in range(len(grid[0]))] for x in range(len(grid))]
    visita[inicio[0]][inicio[1]][inicio[2]] = 1
    paso = [[[-1 for z in range(len(grid[0][0]))] for y in range(len(grid[0]))] for x in range(len(grid))]
    accion = [[[-1 for z in range(len(grid[0][0]))] for y in range(len(grid[0]))] for x in range(len(grid))]

    heuristic = [[[0 for z in range(len(grid[0][0]))] for y in range(len(grid[0]))] for x in range(len(grid))]
    for x in range(len(grid)):
        for y in range(len(grid[0])):
            for z in range(len(grid[0][0])):
                dist=abs(x-fin[0])+abs(y-fin[1])+abs(z-fin[2])
                heuristic[x][y][z]=dist

    x = inicio[0]
    y = inicio[1]
    z = inicio[2]
    g = 0
    h = heuristic[x][y][z]
    f = g+h

    lista = [[f, g, h, x, y, z]]
    paso[x][y][z]= g

    completada = False
    fracasada= False
    contador = 0

    while not completada and not fracasada:
        if len(lista) == 0:
            fracasada = True
```

```
        print("Camino no encontrado")
    else:
        lista.sort()
        lista.reverse()

        siguiente = lista.pop()
        x = siguiente[3]
        y = siguiente[4]
        z = siguiente[5]
        g = siguiente[1]

        paso[x][y][z]= contador
        contador += 1

        if x == fin[0] and y == fin[1] and z == fin[2]:
            completada = True
        else:
            for i in range(len(movimiento)):
                x2 = x + movimiento[i][0]
                y2 = y + movimiento[i][1]
                z2 = z + movimiento[i][2]

                if x2 >= 0 and x2 < len(grid) and y2 >=0 and y2 <len(grid[0]) and z2 >=0 and z2 <len(grid[0][0]) :
                    if visita[x2][y2][z2] == 0 and grid[x2][y2][z2] == 0:
                        g2 = g + coste
                        h2 = heuristic[x2][y2][z2]
                        f2 = g2 + h2
                        lista.append([f2, g2, h2, x2, y2, z2])
                        visita[x2][y2][z2] = 1
                        accion[x2][y2][z2] = i

esquema = [[['' for z in range(len(grid[0][0]))] for y in range(len(grid[0]))] for x in range(len(grid))]
esquema[fin[0]][fin[1]][fin[2]]='*'

ruta=[]
ruta.append([fin[0], fin[1], fin[2]])

while x != inicio[0] or y != inicio[1] or z != inicio[2]:
```

```
        x2 = x - movimiento[accion[x][y][z]][0]
        y2 = y - movimiento[accion[x][y][z]][1]
        z2 = z - movimiento[accion[x][y][z]][2]
        esquema[x2][y2][z2] = flecha[accion[x][y][z]]
        x=x2
        y=y2
        z=z2
        ruta.append([x2, y2, z2])

    ruta.reverse()

    return ruta, fracasada
```

A.3. Algoritmo RRT en 3D

```
visita = [[[0 for z in range(len(grid[0][0]))] for y in range(len(grid[0]))] for x in range(len(grid))]
visita[inicio[0]][inicio[1]][inicio[2]] = 1    #matriz con la posición inicial = 1
paso = [[[-1 for z in range(len(grid[0][0]))] for y in range(len(grid[0]))] for x in range(len(grid))] #
accion = [[[-1 for z in range(len(grid[0][0]))] for y in range(len(grid[0]))] for x in range(len(grid))]

alcance=9
completada=False

def rama(p1,p2):

    x = p1[0]
    y = p1[1]
    z = p1[2]
    movx = 0
    movy = 0
    movz = 0
    pasosx = abs(p2[0]-p1[0])   #Pasos que tengo que recorrer en vertical y en horizontal
    pasosy = abs(p2[1]-p1[1])
    pasosz = abs(p2[2]-p1[2])

    #Veo si me tengo que mover hacia arriba o abajo y hacia derecha o izquierda
```

```
if (p2[0]-p1[0]) >= 0 :
  siguientex = 1
  accionx='^'
else:
  siguientex = -1
  accionx='v'
if p2[1]-p1[1] >=0 :
   siguientey = 1
   acciony='<'
else:
    siguientey=-1
    acciony='>'
if p2[2]-p1[2] >=0 :
   siguientez = 1
   accionz='ab'
else:
    siguientez=-1
    accionz='arr'

#Banderas para ver cual de los 6 caminos me vale
camino1= True
camino2= True
camino3= True
camino4= True
camino5= True
camino6= True
completada=False

#Intento CAMINO1: primero desplazarme en el eje x, luego y, luego z
while (movx < pasosx) and camino1 == True and completada==False:
   x=x+siguientex      #La coordenada x aumenta o disminuye
   movx=movx+1         #Asi se cuantos pasos tengo que dar hacia derecha o izquierda
   if grid[x][y][z] == 1:
      camino1= False
   else:
      if x == fin[0] and y == fin[1] and z == fin[2]:
         completada = True
         accion[x][y][z]=accionx
```

```
            print("Camino encontrado")
        else:
            if accion[x][y][z]==-1:
                accion[x][y][z]=accionx
            visita[x][y][z] = 1
while (movy < pasosy) and camino1 == True and completada==False:
    y=y+siguientey
    movy=movy+1
    if grid[x][y][z] == 1:
        camino1= False
    else:
        if x == fin[0] and y == fin[1] and z == fin[2]:
            completada = True
            accion[x][y][z]=acciony
            print("Camino encontrado")
        else:
            if accion[x][y][z] == -1:
                accion[x][y][z]=acciony
            visita[x][y][z] = 1
while (movz < pasosz) and camino1 == True and completada==False:
    z=z+siguientez
    movz=movz+1
    if grid[x][y][z] == 1:
        camino1= False
    else:
        if x == fin[0] and y == fin[1] and z == fin[2]:
            completada = True
            accion[x][y][z]=accionz
            print("lo encontreeeeeee")
        else:
            if accion[x][y][z] == -1:
                accion[x][y][z]=accionz
            visita[x][y][z] = 1

#Si no, intento CAMINO2, pruebo el camino 2:  x, z, y
if camino1 == False:
    x = p1[0]
    y = p1[1]
```

```
z = p1[2]
movx=0
movy=0
movz=0

while (movx < pasosx) and camino2 == True and completada==False:
    x=x+siguientex  #la coordenada x aumenta o disminuye
    movx=movx+1     #asi se cuantos pasos tengo que dar haia derecha o izquierda
    if grid[x][y][z] == 1:
        camino2= False
    else:
        if x == fin[0] and y == fin[1] and z == fin[2]:
            completada = True
            accion[x][y][z]=accionx
            print("Camino encontrado")
        else:
            if accion[x][y][z]==-1:
                accion[x][y][z]=accionx
            visita[x][y][z] = 1
while (movz < pasosz) and camino2 == True and completada==False:
    z=z+siguientez
    movz=movz+1
    if grid[x][y][z] == 1:
        camino2= False
    else:
        if x == fin[0] and y == fin[1] and z == fin[2]:
            completada = True
            accion[x][y][z]=accionz
            print("Camino encontrado")
        else:
            if accion[x][y][z]==-1:
                accion[x][y][z]=accionz
            visita[x][y][z] = 1
while (movy < pasosy) and camino2 == True and completada==False:
    y=y+siguientey
    movy=movy+1
    if grid[x][y][z] == 1:
        camino2= False
```

```
        else:
            if x == fin[0] and y == fin[1] and z == fin[2]:
                completada = True
                accion[x][y][z]=acciony
                print("Camino encontrado")
            else:
                if accion[x][y][z] == -1:
                    accion[x][y][z]=acciony
                visita[x][y][z] = 1

    #Si no, pruebo el CAMINO3, y, x, z,
    if camino2 == False:
        x = p1[0]
        y = p1[1]
        z = p1[2]
        movx=0
        movy=0
        movz=0

        while (movy < pasosy) and camino3 == True and completada==False:
            y=y+siguientey
            movy=movy+1
            if grid[x][y][z] == 1:
                camino3= False
            else:
                if x == fin[0] and y == fin[1] and z == fin[2]:
                    completada = True
                    accion[x][y][z]=acciony
                    print("Camino encontrado")
                else:
                    if accion[x][y][z]==-1:
                        accion[x][y][z]=acciony
                    visita[x][y][z] = 1

        while (movx < pasosx) and camino3 == True and completada==False:
            x=x+siguientex  #la coordenada x aumenta o disminuye
            movx=movx+1     #asi se cuantos pasos tengo que dar haia derecha o izquierda
            if grid[x][y][z] == 1:
```

```
        camino3= False
    else:
        if x == fin[0] and y == fin[1] and z == fin[2]:
            completada = True
            accion[x][y][z]=accionx
            print("Camino encontrado")
        else:
            if accion[x][y][z]==-1:
                accion[x][y][z]=accionx
            visita[x][y][z] = 1
while (movz < pasosz) and camino3 == True and completada==False:
    z=z+siguientez
    movz=movz+1
    if grid[x][y][z] == 1:
        camino3= False
    else:
        if x == fin[0] and y == fin[1] and z == fin[2]:
            completada = True
            accion[x][y][z]=accionz
            print("Camino encontrado")
        else:
            if accion[x][y][z] == -1:
                accion[x][y][z]=accionz
            visita[x][y][z] = 1

#Si no, pruebo el cCAMINO4, y, z,  x
if camino3==False:
    x = p1[0]
    y = p1[1]
    z = p1[2]
    movx=0
    movy=0
    movz=0

    while (movy < pasosy) and camino4 == True and completada==False:
        y=y+siguientey
        movy=movy+1
        if grid[x][y][z] == 1:
```

```
            camino4= False
        else:
            if x == fin[0] and y == fin[1] and z == fin[2]:
                completada = True
                accion[x][y][z]=acciony
                print("Camino encontrado")
            else:
                if accion[x][y][z] == -1:
                    accion[x][y][z]=acciony
                visita[x][y][z] = 1
    while (movz < pasosz) and camino4 == True and completada==False:
        z=z+siguientez
        movz=movz+1
        if grid[x][y][z] == 1:
            camino4= False
        else:
            if x == fin[0] and y == fin[1] and z == fin[2]:
                completada = True
                accion[x][y][z]=accionz
                print("Camino encontrado")
            else:
                if accion[x][y][z] == -1:
                    accion[x][y][z]=accionz
                visita[x][y][z] = 1
    while (movx < pasosx) and camino4 == True and completada==False:
        x=x+siguientex  #la coordenada x aumenta o disminuye
        movx=movx+1     #asi se cuantos pasos tengo que dar haia derecha o izquierda
        if grid[x][y][z] == 1:
            camino4= False
        else:
            if x == fin[0] and y == fin[1] and z == fin[2]:
                completada = True
                accion[x][y][z]=accionx
                print("Camino encontrado")
            else:
                if accion[x][y][z] == -1:
                    accion[x][y][z]=accionx
                visita[x][y][z] = 1
```

```
#Probamos ahora por el CAMINO5:  z, x, y
if camino4==False:
   x = p1[0]
   y = p1[1]
   z = p1[2]
   movx=0
   movy=0
   movz=0

   while (movz < pasosz) and camino5 == True and completada==False:
      z=z+siguientez
      movz=movz+1
      if grid[x][y][z] == 1:
         camino5= False
      else:
         if x == fin[0] and y == fin[1] and z == fin[2]:
            completada = True
            accion[x][y][z]=accionz
            print("Camino encontrado")
         else:
            if accion[x][y][z]==-1:
               accion[x][y][z]=accionz
            visita[x][y][z] = 1
   while (movx < pasosx) and camino5 == True and completada==False:
      x=x+siguientex  #la coordenada x aumenta o disminuye
      movx=movx+1     #asi se cuantos pasos tengo que dar haia derecha o izquierda
      if grid[x][y][z] == 1:
         camino5= False
      else:
         if x == fin[0] and y == fin[1] and z == fin[2]:
            completada = True
            accion[x][y][z]=accionx
            print("Camino encontrado")
         else:
            if accion[x][y][z]==-1:
               accion[x][y][z]=accionx
```

```
            visita[x][y][z] = 1
    while (movy < pasosy) and camino5 == True and completada==False:
        y=y+siguientey
        movy=movy+1
        if grid[x][y][z] == 1:
            camino5= False
        else:
            if x == fin[0] and y == fin[1] and z == fin[2]:
                completada = True
                accion[x][y][z]=acciony
                print("Camino encontrado")
            else:
                if accion[x][y][z]==-1:
                    accion[x][y][z]=acciony
                visita[x][y][z] = 1

    #Pruebo por el último, CAMINO6: z, y, x
    if camino5==False:
        x = p1[0]
        y = p1[1]
        z = p1[2]
        movx=0
        movy=0
        movz=0
        while (movz < pasosz) and camino6 == True and completada==False:
            z=z+siguientez
            movz=movz+1
            if grid[x][y][z] == 1:
                camino6= False
            else:
                if x == fin[0] and y == fin[1] and z == fin[2]:
                    completada = True
                    accion[x][y][z]=accionz
                    print("Camino encontrado")
                else:
                    if accion[x][y][z] == -1:
                        accion[x][y][z]=accionz
                    visita[x][y][z] = 1
```

```
            while (movy < pasosy) and camino6 == True and completada==False:
                y=y+siguientey
                movy=movy+1
                if grid[x][y][z] == 1:
                    camino6= False
                else:
                    if x == fin[0] and y == fin[1] and z == fin[2]:
                        completada = True
                        accion[x][y][z]=acciony
                        print("Camino encontrado")
                    else:
                        if accion[x][y][z] == -1:
                            accion[x][y][z]=acciony
                        visita[x][y][z] = 1
            while (movx < pasosx) and camino6 == True and completada==False:
                x=x+siguientex  #la coordenada x aumenta o disminuye
                movx=movx+1     #asi se cuantos pasos tengo que dar haia derecha o izquierda
                if grid[x][y][z] == 1:
                    camino6= False
                else:
                    if x == fin[0] and y == fin[1] and z == fin[2]:
                        completada = True
                        accion[x][y][z]=accionx
                        print("Camino encontrado")
                    else:
                        if accion[x][y][z] == -1:
                            accion[x][y][z]=accionx
                        visita[x][y][z] = 1

            if camino6==False:
                print("Ningún camino es posible")
                return [x,y,z], completada
            else:
                return[x,y,z], completada
        else:
            return[x,y,z], completada
    else:
        return[x,y,z], completada
```

```
            else:
                return[x,y,z], completada
        else:
            return[x,y,z], completada
    else:
        return [x,y,z], completada



def distancia(p1,p2):
    return abs(p1[0]-p2[0])+abs(p1[1]-p2[1])+abs(p1[2]-p2[2])



def siguientepaso(p1,p2):
    if dist(p1,p2) <= alcance:
        return p2
    else:
        difx = p2[0]-p1[0]
        dify = p2[1]-p1[1]
        difz = p2[2]-p1[2]
        dif= abs(difx) + abs(dify) + abs(difz)
        siguiente = p1[0] + round((difx/dif)*alcance), p1[1] + round((dify/dif)*alcance), p1[2] +
round((difz/dif)*alcance)
        return siguiente



#Programa principal
nodos = []
nodos.append(inicio)
numnodos=dimx*dimy*dimz
i=0
alcanzado =False

while i<numnodos and alcanzado==False:
    i=i+1
    rand = round(random.random()*(len(grid)-1)), round(random.random()*(len(grid[0])-1)),
round(random.random()*(len(grid[0][0])-1))

  while grid[rand[0]][rand[1]][rand[2]] == 1 and visita[rand[0]][rand[1]][rand[2]] == 1:
        rand = round(random.random()*(len(grid)-1)), round(random.random()*(len(grid[0])-1)),
```

```
round(random.random()*(len(grid[0][0])-1))

 nn = nodos[0]
 for p in nodes:
   if distancia(p,rand) < distancia(nn,rand):
       nn = p
 newnode = siguientepaso(nn,rand)
 nuevo, alcanzado =camino(nn,newnode)
 nodos.append(nuevo)

j=0
ruta=[]
x=nuevo[0]
y=nuevo[1]
z=nuevo[2]
ruta.append([x,y,z])

if alcanzado==True:
  while (x != inicio[0] or y != inicio[1] or z != inicio[2]) :
    j=j+1
    if accion[x][y][z] == '^':
      x=x-1
    else:
      if accion[x][y][z] == 'v':
        x=x+1
      else:
        if accion[x][y][z] == '<':
          y=y-1
        else:
          if accion[x][y][z] == '>':
            y=y+1
          else:
            if accion[x][y][z] == 'arr':
              z=z+1
            else:
              if accion[x][y][z] == 'ab':
                z=z-1
```

```
        ruta.append([x,y,z])

    ruta.reverse()

#Ruta nos devuelve el laberinto encontrado. Programa principal
escenario(grid, inicio, fin)
animacion(grid, ruta, inicio, fin)
dibujoruta(ruta)
```

A.4. Funciones comunes, variables y llamada a las funciones

```
#Importación de librerías
import bpy
from math import *
import random
import time

#Funicón escenario
def escenario(grid,inicio,fin):

    bpy.ops.object.select_by_type(type='MESH')
    bpy.ops.object.delete(use_global=False)

    bpy.ops.mesh.primitive_plane_add(location=((len(grid)/2)-0.5,(len(grid[0])/2)-0.5,-0.5))
    bpy.ops.transform.resize(value=(len(grid)/2, len(grid[0])/2, 0))

     plano=bpy.context.scene.objects.active

    materialplano= bpy.data.materials.new(plano.name)
    materialplano.diffuse_color = (1,1,1)

    materialcubo= bpy.data.materials.new(plano.name)
    materialcubo.diffuse_color = (0,0.5,1)

    plano.data.materials.append(materialplano)
```

```
for x in range(len(grid)):
    for y in range(len(grid[0])):
        for z in range(len(grid[0][0])):
            if grid[x][y][z] == 1:
                bpy.ops.mesh.primitive_cube_add(location=(x,y,z), radius=0.45)
                bpy.context.scene.objects.active.data.materials.append(materialcubo)

#Función animación
def animación(grid, ruta, inicio, fin):

    x = inicio[0]
    y = inicio[1]
    z = inicio[2]
    frame = 1
    i = 0

    bpy.ops.mesh.primitive_monkey_add(radius=0.3, view_align=False, enter_editmode=False, location=(x, y,
        z), layers=(True, False, False, False, False, False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False))
    mono = bpy.context.object

    materialmono= bpy.data.materials.new(mono.name)
    materialmono.diffuse_color = (1,1,0)
    mono.data.materials.append(materialmono)

    while x != fin[0] or y != fin[1] or z != fin[2]:
        mono.keyframe_insert(data_path="location", frame=frame)
        x=ruta[i][0]
        y=ruta[i][1]
        z=ruta[i][2]

        mono.location[2] = z
        mono.location[1] = y
        mono.location[0] = x
        i+=1
        frame += 10
        mono.keyframe_insert(data_path='location', frame=frame)
```

```
    bpy.data.scenes["Scene"].frame_end = frame+15
    bpy.data.scenes["Scene"].frame_current = 0



#Función dibujoruta
def dibujoruta(ruta):

    for i in range(len(ruta)):
        bpy.ops.mesh.primitive_uv_sphere_add(size=.1,location=(ruta[i][0],ruta[i][1],ruta[i][2]))
        materialbolas = bpy.data.materials.new("Nombre del material")
        materialbolas.diffuse_color=(1,0,0)
        bpy.context.scene.objects.active.data.materials.append(materialbolas)

#--------------------------------------------------------
#Programa principal para Dijkstra y A*

dimx = 10
dimy = 8
dimz = 3

grid = [[[round(random.random()-.3) for z in range(dimz)] for y in range(dimy)] for x in range(dimx)]
inicio = [0, 0, 0] # Posición inicial
fin =  [len(grid)-1, len(grid[0])-1, len(grid[0][0])-1]  # Posición destino

grid[inicio[0]][inicio[1]][inicio[2]] = 0
grid[fin[0]][fin[1]][fin[2]] = 0

movimiento = [[-1, 0, 0], # subir
              [0, -1, 0], # izquierda
              [1, 0, 0], # bajar
              [0, 1, 0], # derecha
              [0, 0, -1], # bajar
              [0, 0, 1]] # subir

flecha = ['^', '<', 'v', '>','ab', 'arr']
coste = 1

escenario(grid, inicio, fin)
```

```
camino, fracaso = Dijkstra(grid, inicio, fin)
camino, fracaso = Astar(grid, inicio, fin)

if fracaso == True:
    print("Camino no encontrado")
else:
   animacion(grid, camino, inicio, fin)
    dibujoruta(camino)
```

ANEXO B. CÓDIGOS V-REP: ALGORITMOS Y ANIMACIONES

B.1. Algoritmo Dijkstra en 3D

```
--INICIALIZACIÓN DEL SCRIPT: DEFINICIÓN DEL ESCENARIO Y BÚSQUEDA DEL CAMINO
if (sim_call_type==sim_childscriptcall_initialization) then

console=simAuxiliaryConsoleOpen('titulo',100,1)
mensaje=simAuxiliaryConsolePrint(console,'Hago el laberinto e inicio del Script \n')

--COlORES PARA LA REPRESENTACIÓN
white = {1,1,1}
black = {0,0,0}
blue = {0,0,1}
red = {1,0,0}
green = {0,1,0}

--DIMENSIONES DEL LABERINTO
dimx=10
dimy=8
dimz=6
grid = {}

--CREO EL LABERINTO EN LA MATRIZ GRID
for x=1,dimx,1 do
   grid[x]={}
   for y=1,dimy,1 do
      grid[x][y]={}
      for z=1,dimz,1 do
         grid[x][y][z]=math.modf(math.random()+0.2) end end end


lado=0.9
```

```
inicio= {1,1,1}
fin= {#grid, #grid[1], #grid[1][1]}
grid[inicio[1]][inicio[2]][inicio[3]]=0
grid[fin[1]][fin[2]][fin[3]]=0

movimiento ={{-1,0,0},{0,-1,0},{1,0,0},{0,1,0},{0,0,-1},{0,0,1}}
flecha = {'^', '<', 'v', '>', 'ab', 'arr'}
coste=1

--CREO EL PLANO
plano=simCreatePureShape(0, 1, {dimx+1,dimy+1,0},1)
simSetObjectPosition(plano,-1,{0,0,-(lado/2)})
simSetShapeColor(plano, null, 0, white)

--CREO EL ESCENARIO
for x=1,#grid,1 do
   for y=1, #grid[1] do
      for z=1, #grid[1][1] do
         if grid[x][y][z] == 1 then
            cubo=simCreatePureShape(0, 1, {lado,lado,lado},1)
            simSetObjectPosition(cubo,-1,{x-(dimx/2)-lado/2,y-(dimy/2)-lado/2,z-1})
            simSetShapeColor(cubo, null, 0, blue) end end end end

---------ALGORITMO DIJKSTRA------------------------
visita={}
paso={}
accion={}

for x=1,#grid,1 do
   visita[x]={}
   for y=1,#grid[1],1 do
      visita[x][y]={}
      for z=1,#grid[1][1],1 do
         visita[x][y][z]=0 end end end

visita[inicio[1]][inicio[2]][inicio[3]]=1
```

```
for x=1,#grid,1 do
   paso[x]={}
   for y=1,#grid[1],1 do
      paso[x][y]= {}
      for z=1, #grid[1][1], 1 do
         paso[x][y][z]=-1   end end end

for x=1,#grid,1 do
   accion[x]={}
   for y=1,#grid[1],1 do
      accion[x][y]= {}
      for z=1, #grid[1][1], 1 do
         accion[x][y][z]=-1   end  end end

x=inicio[1]
y=inicio[2]
z=inicio[3]
g=0

lista ={{g,x,y,z}}
paso[x][y][z]= g

contador=0
completada = false
fracasada= false

while (completada == false) and (fracasada == false)  do
   if #lista==0 then
      fracasada = true
      mensaje=simAuxiliaryConsolePrint(console,'Ruta no encontrada')
   else

      -----sort y reverse en estas lineas
      for iteraciones=1,#lista,1 do
         min=lista[1][1]
         pos=1
         for i=1,#lista+1-iteraciones, 1 do
            if lista[i][1]< min then
```

```
            min=lista[i][1]
            pos=i   end   end
        aux=lista[table.maxn(lista)+1-iteraciones]
        lista[table.maxn(lista)+1-iteraciones]=lista[pos]
        lista[pos]=aux
        end

        siguiente=lista[table.maxn(lista)]
        table.remove(lista)

        g = siguiente[1]
        x = siguiente[2]
        y = siguiente[3]
        z = siguiente[4]
        paso[x][y][z]=contador
        contador=contador+1

        mensaje=simAuxiliaryConsolePrint(console, x .. y .. z .."\n")

        if x == fin[1] and y == fin[2] and z == fin[3] then
          completada = true
          mensaje=simAuxiliaryConsolePrint(console,'encontrado \n')
        else
         for d=1,#movimiento,1 do
           x2 = x + movimiento[d][1]
           y2 = y + movimiento[d][2]
           z2 = z + movimiento[d][3]

           if x2 >= 1 and x2 <= #grid and y2 >=1 and y2 <= #grid[1] and z2 >=1 and z2 <= #grid[1][1] then
             if visita[x2][y2][z2] == 0 and grid[x2][y2][z2] == 0 then
               g2 = g +coste
               visita[x2][y2][z2] = 1
               accion[x2][y2][z2] = d
               table.insert(lista,{g2, x2, y2, z2})
             end
           end
         end
        end
```

```
  end
end

esquema = {}
for x=1,#grid,1 do
  esquema[x]={}
  for y=1,#grid[1],1 do
    esquema[x][y]={}
    for z=1,#grid[1][1],1 do
      esquema[x][y][z]=' '  end  end  end

esquema[fin[1]][fin[2]][fin[3]]='*'

ruta = {}
table.insert(ruta,1 ,{fin[1], fin[2], fin[3]}) ------- el 1 hace que las vaya insertando al principio
num=0
while x ~= inicio[1] or y ~= inicio[2] or z ~= inicio[3]  do
  num=num+1
  x2 = x - movimiento[accion[x][y][z]][1]
  y2 = y - movimiento[accion[x][y][z]][2]
  z2 = z - movimiento[accion[x][y][z]][3]
  esquema[x2][y2][z2] = flecha[accion[x][y][z]]
  x=x2
  y=y2
  z=z2
  table.insert(ruta,1,{x2, y2, z2})
end

---------Índice en la ruta y en las bolas
i=ruta[1][1]
j=ruta[1][2]
k=ruta[1][3]
b=1
r=1
--------Segundo actual
actual=0

bolas = {}
```

```
end

--EN ESTA PARTE DEL SCRIPT SE REALIZA LA ANIMACIÓN
if (sim_call_type==sim_childscriptcall_actuation) then

t=simGetSimulationTime() --- t va recogiendo los tiempos. pasa por aqui cada 'dt'

  if t <= #ruta-1 then
    ------siguiente recoge el valor del segundo en el que estamos, actual en el que estábamos
    siguiente=math.modf(t) -- Valor entero de t, sin la parte decimal

    if siguiente ~= actual then
      mensaje=simAuxiliaryConsolePrint(console,'CAMBIO DE SEGUNDO \n')
      r=r+1
      if esquema[i][j][k] == 'v'  then
        i=i+1
      elseif esquema[i][j][k] == '>' then
        j=j+1
      elseif esquema[i][j][k] == '<' then
        j=j-1
      elseif esquema[i][j][k] == '^' then
        i=i-1
      elseif esquema[i][j][k] == 'arr' then
        k=k+1
      elseif esquema[i][j][k] == 'ab' then
        k=k-1
      end
    end

    bolas[b]=simCreatePureShape(1, 1, {.5,.5,.5} ,1)
    if esquema[i][j][k] == 'v'  then
      simSetObjectPosition(bolas[b],-1,{ruta[r][1]+t-math.modf(t)-(dimx/2)-lado/2,ruta[r][2]-(dimy/2)-
       lado/2,ruta[r][3]-1})
    elseif esquema[i][j][k] == '>' then
      simSetObjectPosition(bolas[b],-1,{ruta[r][1]-(dimx/2)-lado/2,ruta[r][2]+t-math.modf(t)-(dimy/2)-
       lado/2,ruta[r][3]-1})
```

```
elseif esquema[i][j][k] == '<' then
    simSetObjectPosition(bolas[b],-1,{ruta[r][1]-(dimx/2)-lado/2,ruta[r][2]-(t-math.modf(t))-(dimy/2)-
    lado/2,ruta[r][3]-1})
elseif esquema[i][j][k] == '^' then
    simSetObjectPosition(bolas[b],-1,{ruta[r][1]-(t-math.modf(t))-(dimx/2)-lado/2,ruta[r][2]-(dimy/2)-
    lado/2,ruta[r][3]-1})
elseif esquema[i][j][k] == 'arr' then
    simSetObjectPosition(bolas[b],-1,{ruta[r][1]-(dimx/2)-lado/2,ruta[r][2]-(dimy/2)-lado/2,ruta[r][3]-1+t-
    math.modf(t)})
elseif esquema[i][j][k] == 'ab' then
    simSetObjectPosition(bolas[b],-1,{ruta[r][1]-(dimx/2)-lado/2,ruta[r][2]-(dimy/2)-lado/2,ruta[r][3]-1-
    t+math.modf(t)})
end

actual=math.modf(t)
simSetShapeColor(bolas[b], null, 0, black)
b=b+1
else
    simPauseSimulation()    ---- Cuando se llega al final de la ruta, se detiene
end
    --------------------------------------------------------------------
end


if (sim_call_type==sim_childscriptcall_sensing) then

end
```

B.2. Algoritmo A* en 3D

```
--INICIALIZACIÓN DEL SCRIPT: DEFINICIÓN DEL ESCENARIO Y BÚSQUEDA DEL CAMINO
if (sim_call_type==sim_childscriptcall_initialization) then

consola=simAuxiliaryConsoleOpen(‘Algritmo A*’,100,1)
mensaje=simAuxiliaryConsolePrint(consola,'Hago el laberinto e inicio del Script \n')

--COlORES PARA LA REPRESENTACIÓN
blanco = {1,1,1}
```

```
negro = {0,0,0}
azul = {0,0,1}
rojo = {1,0,0}
verde = {0,1,0}

--DIMENSIONES DEL LABERINTO
dimx=10
dimy=8
dimz=6
grid = {}

--CREO EL LABERINTO EN LA MATRIZ GRID
for x=1,dimx,1 do
   grid[x]={}
   for y=1,dimy,1 do
      grid[x][y]={}
      for z=1,dimz,1 do
         grid[x][y][z]=math.modf(math.random()+0.2) end end end


lado=0.9

inicio= {1,1,1}
fin= {#grid, #grid[1], #grid[1][1]}
grid[inicio[1]][inicio[2]][inicio[3]]=0
grid[fin[1]][fin[2]][fin[3]]=0

movimiento ={{-1,0,0},{0,-1,0},{1,0,0},{0,1,0},{0,0,-1},{0,0,1}}
flecha = {'^', '<', 'v', '>', 'ab', 'arr'}
coste=1

--CREO EL PLANO
plano=simCreatePureShape(0, 1, {dimx+1,dimy+1,0},1)
simSetObjectPosition(plano,-1,{0,0,-(lado/2)})
simSetShapeColor(plano, null, 0, blanco)

--CREO EL ESCENARIO
for x=1,#grid,1 do
```

```
    for y=1, #grid[1] do
        for z=1, #grid[1][1] do
            if grid[x][y][z] == 1 then
                cubo=simCreatePureShape(0, 1, {lado,lado,lado},1)
                simSetObjectPosition(cubo,-1,{x-(dimx/2)-lado/2,y-(dimy/2)-lado/2,z-1})
                simSetShapeColor(cubo, null, 0, azul  )end end end end

---------ALGORITMO A*-------------------------
visita={}
paso={}
accion={}

for x=1,#grid,1 do
    visita[x]={}
    for y=1,#grid[1],1 do
        visita[x][y]={}
        for z=1,#grid[1][1],1 do
            visita[x][y][z]=0 end end end

visita[inicio[1]][inicio[2]][inicio[3]]=1

for x=1,#grid,1 do
    paso[x]={}
    for y=1,#grid[1],1 do
        paso[x][y]= {}
        for z=1, #grid[1][1], 1 do
            paso[x][y][z]=-1   end end end

for x=1,#grid,1 do
    accion[x]={}
    for y=1,#grid[1],1 do
        accion[x][y]= {}
        for z=1, #grid[1][1], 1 do
            accion[x][y][z]=-1   end  end end


--FUNCIÓN HEURÍSTICA
heuristic={}
```

```
for x=1,#grid,1 do
   heuristic[x]={}
   for y=1,#grid[1],1 do
      heuristic[x][y]= {}
      for z=1, #grid[1][1], 1 do
         dist=math.abs(x-fin[1])+math.abs(y-fin[2])+math.abs(z-fin[3])
         heuristic[x][y][z] = dist   end  end end

x=inicio[1]
y=inicio[2]
z=inicio[3]
g=0
h = heuristic[x][y][z]
f=g+h

lista ={{f,g,h,x,y,z}}
paso[x][y][z]= g

contador=0
completada = false
fracasada= false

while (completada == false) and (fracasada == false)  do
   if #lista==0 then
      fracasada = true
      mensaje=simAuxiliaryConsolePrint(consola,'Ruta no encontrada')
   else

      -----sort y reverse en estas lineas
      for iteraciones=1,#lista,1 do
         min=lista[1][1]
         pos=1
         for i=1,#lista+1-iteraciones, 1 do
            if lista[i][1]< min then
               min=lista[i][1]
               pos=i   end   end
      aux=lista[table.maxn(lista)+1-iteraciones]
      lista[table.maxn(lista)+1-iteraciones]=lista[pos]
```

```
      lista[pos]=aux
    end

    siguiente=lista[table.maxn(lista)]
    table.remove(lista)

    g = siguiente[2]
    x = siguiente[4]
    y = siguiente[5]
    z = siguiente[6]
    paso[x][y][z]=contador
    contador=contador+1

    mensaje=simAuxiliaryConsolePrint(console, x .. y .. z .."\n")

    if x == fin[1] and y == fin[2] and z == fin[3] then
      completada = true
      mensaje=simAuxiliaryConsolePrint(console,'encontrado \n')
    else
     for d=1,#movimiento,1 do
        x2 = x + movimiento[d][1]
        y2 = y + movimiento[d][2]
        z2 = z + movimiento[d][3]

        if x2 >= 1 and x2 <= #grid and y2 >=1 and y2 <= #grid[1] and z2 >=1 and z2 <= #grid[1][1] then
          if visita[x2][y2][z2] == 0 and grid[x2][y2][z2] == 0 then
            g2 = g +coste
            h2 = heuristic[x2][y2][z2]
            f2 = g2 + h2
            visita[x2][y2][z2] = 1
            accion[x2][y2][z2] = d
            table.insert(lista,{f2,g2,h2, x2, y2, z2})
          end
        end
      end
    end
  end
end
```

```
esquema = {}
for x=1,#grid,1 do
    esquema[x]={}
    for y=1,#grid[1],1 do
        esquema[x][y]={}
        for z=1,#grid[1][1],1 do
            esquema[x][y][z]=' '  end  end  end

esquema[fin[1]][fin[2]][fin[3]]='*'

ruta = {}
table.insert(ruta,1 ,{fin[1], fin[2], fin[3]}) ------- el 1 hace que las vaya insertando al principio
num=0
while x ~= inicio[1] or y ~= inicio[2] or z ~= inicio[3]  do
    num=num+1
   x2 = x - movimiento[accion[x][y][z]][1]
    y2 = y - movimiento[accion[x][y][z]][2]
    z2 = z - movimiento[accion[x][y][z]][3]
    esquema[x2][y2][z2] = flecha[accion[x][y][z]]
    x=x2
    y=y2
    z=z2
    table.insert(ruta,1,{x2, y2, z2})
end

---------Índice en la ruta y en las bolas
i=ruta[1][1]
j=ruta[1][2]
k=ruta[1][3]
b=1
r=1
--------Segundo actual
actual=0

bolas = {}

end
```

```
--EN ESTA PARTE DEL SCRIPT SE REALIZA LA ANIMACIÓN
if (sim_call_type==sim_childscriptcall_actuation) then

t=simGetSimulationTime() --- t va recogiendo los tiempos. pasa por aqui cada 'dt'

  if t <= #ruta-1 then
    ------siguiente recoge el valor del segundo en el que estamos, actual en el que estábamos
    siguiente=math.modf(t) -- Valor entero de t, sin la parte decimal

    if siguiente ~= actual then
      mensaje=simAuxiliaryConsolePrint(console,'CAMBIO DE SEGUNDO \n')
      r=r+1
      if esquema[i][j][k] == 'v'  then
        i=i+1
      elseif esquema[i][j][k] == '>' then
        j=j+1
      elseif esquema[i][j][k] == '<' then
        j=j-1
      elseif esquema[i][j][k] == '^' then
        i=i-1
      elseif esquema[i][j][k] == 'arr' then
        k=k+1
      elseif esquema[i][j][k] == 'ab' then
        k=k-1
      end
    end

    bolas[b]=simCreatePureShape(1, 1, {.5,.5,.5} ,1)
    if esquema[i][j][k] == 'v'  then
      simSetObjectPosition(bolas[b],-1,{ruta[r][1]+t-math.modf(t)-(dimx/2)-lado/2,ruta[r][2]-(dimy/2)-
       lado/2,ruta[r][3]-1})
    elseif esquema[i][j][k] == '>' then
      simSetObjectPosition(bolas[b],-1,{ruta[r][1]-(dimx/2)-lado/2,ruta[r][2]+t-math.modf(t)-(dimy/2)-
       lado/2,ruta[r][3]-1})
    elseif esquema[i][j][k] == '<' then
      simSetObjectPosition(bolas[b],-1,{ruta[r][1]-(dimx/2)-lado/2,ruta[r][2]-(t-math.modf(t))-(dimy/2)-
       lado/2,ruta[r][3]-1})
```

```
        elseif esquema[i][j][k] == '^' then
          simSetObjectPosition(bolas[b],-1,{ruta[r][1]-(t-math.modf(t))-(dimx/2)-lado/2,ruta[r][2]-(dimy/2)-
           lado/2,ruta[r][3]-1})
        elseif esquema[i][j][k] == 'arr' then
          simSetObjectPosition(bolas[b],-1,{ruta[r][1]-(dimx/2)-lado/2,ruta[r][2]-(dimy/2)-lado/2,ruta[r][3]-1+t-
           math.modf(t)})
        elseif esquema[i][j][k] == 'ab' then
          simSetObjectPosition(bolas[b],-1,{ruta[r][1]-(dimx/2)-lado/2,ruta[r][2]-(dimy/2)-lado/2,ruta[r][3]-1-
           t+math.modf(t)})
        end


        actual=math.modf(t)
        simSetShapeColor(bolas[b], null, 0, black)
        b=b+1
      else
         simPauseSimulation()    ---- Cuando se llega al final de la ruta, se detiene
      end
           ------------------------------------------------------------------
end



if (sim_call_type==sim_childscriptcall_sensing) then


end
```

ANEXO C. CÓDIGOS EN BLENDER Y PYGAME: ANIMACIONES EN 2D

C.1. Algoritmo Dijkstra en 2D

Código en Blender:

```
import bpy
from math import *
import random
import time
import os

def Dijkstra(grid, inicio, fin):

    visita = [[0] *len(grid[0]) for i in grid]
    visita[inicio[0]][inicio[1]] = 1
    paso = [[-1] *len(grid[0]) for i in grid]
    accion = [[-1] *len(grid[0]) for i in grid]

    x = inicio[0]
    y = inicio[1]
    g = 0

    lista = [[g, x, y]]
    paso[x][y] = g
    guardapaso=[]

    file = open(os.path.splitext(bpy.data.filepath)[0] + "variables.py", 'w')

    completada = False
    fracasada= False
    contador = 0
```

```
while not completada and not fracasada:
    if len(lista) == 0:
        fracasada = True
        print("Fail")
    else:
        lista.sort()
        lista.reverse()

        siguiente = lista.pop()
        x = siguiente[1]
        y = siguiente[2]
        g = siguiente[0]

        paso[x][y]= contador
        guardapaso.append([x,y])
        contador += 1

        if x == fin[0] and y == fin[1]:
            completada = True
        else:
            for i in range(len(movimiento)):
                x2 = x + movimiento[i][0]
                y2 = y + movimiento[i][1]

                if x2 >= 0 and x2 < len(grid) and y2 >=0 and y2 <len(grid[0]):
                    if visita[x2][y2] == 0 and grid[x2][y2] == 0:
                        g2 = g + coste
                        lista.append([g2, x2, y2])
                        visita[x2][y2] = 1
                        accion[x2][y2] = i

esquema = [[''] *len(grid[0]) for i in grid]
esquema[fin[0]][fin[1]]='*'

ruta=[]
ruta.append([fin[0], fin[1]])
```

```
    while x != inicio[0] or y != inicio[1]:
        x2 = x - movimiento[accion[x][y]][0]
        y2 = y - movimiento[accion[x][y]][1]
        esquema[x2][y2] = flecha[accion[x][y]]
        x=x2
        y=y2

        ruta.append([x2, y2])

    ruta.reverse()

    return ruta, guardapaso, fracasada


def escenario(grid,inicio,fin):

    bpy.ops.object.select_by_type(type='MESH')
    bpy.ops.object.delete(use_global=False)

    bpy.ops.mesh.primitive_plane_add(location=((len(grid)/2)-0.5,(len(grid[0])/2)-0.5,-0.5))
    bpy.ops.transform.resize(value=(len(grid)/2, len(grid[0])/2, 0))

    plano=bpy.context.scene.objects.active

    materialplano= bpy.data.materials.new(plano.name)
    materialplano.diffuse_color = (1,1,1)



    materialcubo= bpy.data.materials.new(plano.name)
    materialcubo.diffuse_color = (0,0.5,1)

    plano.data.materials.append(materialplano)

    for x in range(len(grid)):
        for y in range(len(grid[0])):
            if grid[x][y] == 1:
                bpy.ops.mesh.primitive_cube_add(location=(x,y,0), radius=0.5)
                bpy.context.scene.objects.active.data.materials.append(materialcubo)
```

```
def animacion(grid, ruta, inicio, fin):

    x = inicio[0]
    y = inicio[1]
    z = 0
    frame = 1
    i = 0

    bpy.ops.mesh.primitive_monkey_add(radius=0.3, enter_editmode=False, location=(x, y, 0),)
    mono = bpy.context.object

    materialmono= bpy.data.materials.new(mono.name)
    materialmono.diffuse_color = (1,0,0)
    mono.data.materials.append(materialmono)

    while x!= fin[0] or y != fin[1]:

        mono.keyframe_insert(data_path="location", frame=frame)
        x=ruta[i][0]
        y=ruta[i][1]
        z=0

        mono.location[2] = z
        mono.location[1] = y
        mono.location[0] = x
        i+=1
        frame += 10
        mono.keyframe_insert(data_path='location', frame=frame)

    print("Creando el monito")
    bpy.data.scenes["Scene"].frame_end = frame+15


def dibujoruta(ruta):

    for i in range(len(ruta)):
```

```
    bpy.ops.mesh.primitive_uv_sphere_add(size=.1,location=(ruta[i][0],ruta[i][1],0))
    materialbolas = bpy.data.materials.new("Nombre del material")
    materialbolas.diffuse_color=(0,1,0)
    bpy.context.scene.objects.active.data.materials.append(materialbolas)


#*******-------------------------------------------------------
#Programa principal

dimx=20
dimy=10

grid = [[round(random.random()-.4) for y in range(dimy)] for x in range(dimx)]
grid2 = [[0 for y in range(dimy)] for x in range(dimx)]

inicio=[0,0]
fin = [8,9]

grid[inicio[0]][inicio[1]] = 0
grid[fin[0]][fin[1]]= 0

movimiento = [[-1, 0],
              [0, -1],
              [1, 0],
              [0, 1]]

flecha = ['^', '<', 'v', '>']
coste = 1

escenario(grid,inicio,fin)
camino, busqueda, fracaso  = Dijkstra(grid,inicio,fin)
if fracaso == True:
  print("Camino no encontrado")
else:
  mueve=animacion(grid, camino, inicio, fin)
  dibujoruta(camino)

file = open(os.path.splitext(bpy.data.filepath)[0] + "variables.py", 'w')
```

```
file.write('inicio =' + str(inicio).strip('') + '\n')
file.write('fin =' + str(fin).strip('') + '\n')
file.write('grid =' + str(grid).strip('') + '\n')
file.write('busqueda =' + str(busqueda).strip('') + '\n')
file.write('iteraciones=' + str(len(busqueda)).strip('') + '\n')
file.write('camino =' + str(camino).strip('') + '\n')
file.close()
```

Código en Pygame:

```
import pygame
import Dijkstravariables

# Definimos algunos colores
NEGRO = (0, 0, 0)
BLANCO = (255, 255, 255)
VERDE = ( 0, 255, 0)
ROJO = (255, 0, 0)
AMARILLO = (255, 255,0)
AZUL = (0, 255, 255)

ini = Dijkstravariables.inicio
fin = Dijkstravariables.fin
grid = Dijkstravariables.grid

fil=len(grid)
col=len(grid[0])

# Establecemos el margen entre las celdas y el tamaño del lado
MARGEN = 5
lado = 45
# Inicializamos pygame
pygame.init()

# Establecemos el LARGO y ALTO de la pantalla, DEJARLO FIJO
DIMENSION_VENTANA = [lado*col,lado*fil]
pantalla = pygame.display.set_mode(DIMENSION_VENTANA)
```

```
# Establecemos el LARGO y ALTO de cada celda de la retícula.
LARGO  = lado-MARGEN
ALTO = lado-MARGEN

# Establecemos el título de la pantalla.
pygame.display.set_caption("Mapa")

# Iteramos hasta que el usuario pulse el botón de salir.
hecho = False

# Dibujamos la retícula
pantalla.fill(NEGRO)
for fila in range(fil):
   for columna in range(col):
      color = BLANCO
      if grid[fila][columna]==1:
         color = NEGRO
      pygame.draw.rect(pantalla,color,[(MARGEN+LARGO) * columna + MARGEN,(MARGEN+ALTO) *
fila + MARGEN,  LARGO, ALTO])

pygame.draw.rect(pantalla,AZUL,[(MARGEN+LARGO) * ini[1] + MARGEN,(MARGEN+ALTO) * ini[0]+
MARGEN, LARGO, ALTO])
pygame.draw.rect(pantalla,ROJO,[(MARGEN+LARGO) * fin[1] + MARGEN,(MARGEN+ALTO) * fin[0]+
MARGEN, LARGO, ALTO])
pygame.time.wait(500)

# Lo usamos para establecer cuán rápido de refresca la pantalla.
reloj = pygame.time.Clock()
x=0
tabla=Dijkstravariables.busqueda
contador=0

# -------- Bucle Principal del Programa-----------
while not hecho and x<len(tabla):
   for evento in pygame.event.get():
      if evento.type == pygame.QUIT:
         hecho = True

  # Establecemos el fondo de pantalla.
```

```
    contador=contador+1
    pygame.time.wait(1)
    fila = tabla[x][0]
    columna = tabla[x][1]
    pygame.draw.rect(pantalla,VERDE,[(MARGEN+LARGO) * columna + MARGEN,(MARGEN+ALTO) *
fila + MARGEN, LARGO, ALTO])
    x=x+1

    # Limitamos a 20 fotogramas por segundo.
    reloj.tick(1000)

    # Avanzamos y actualizamos la pantalla con lo que hemos dibujado.
    pygame.display.flip()

pygame.draw.rect(pantalla,ROJO,[(MARGEN+LARGO) * ini[1] + MARGEN,(MARGEN+ALTO) * ini[0]+
MARGEN,  LARGO, ALTO])

print("DIBUJO EL CAMINO")
camino=Dijkstravariables.camino
x=0
contador=0

# -------- Bucle Para hacer el camino-----------
while not hecho and x<len(camino):

    for evento in pygame.event.get():
        if evento.type == pygame.QUIT:
            hecho = True

   # Establecemos el fondo de pantalla.
    contador= contador+1
    pygame.time.wait(5)
    fila = camino[x][0]
    columna = camino[x][1]
    pygame.draw.rect(pantalla,AMARILLO,[(MARGEN+LARGO) * columna + MARGEN, (MARGEN +
ALTO)  * fila + MARGEN, LARGO, ALTO])
    x=x+1

    reloj.tick(20) #con este marco la velocidad, a más grande más rápido
```

```
    # Avanzamos y actualizamos la pantalla con lo que hemos dibujado.
    pygame.display.flip()


pygame.draw.rect(pantalla,AZUL,[(MARGEN+LARGO) * ini[1] + MARGEN,(MARGEN+ALTO) * ini[0]+
MARGEN, LARGO, ALTO])
pygame.draw.rect(pantalla,ROJO,[(MARGEN+LARGO) * fin[1] + MARGEN,(MARGEN+ALTO) * fin[0]+
MARGEN, LARGO, ALTO])

pygame.display.flip()
iteraciones=Dijkstravariables.iteraciones
print(iteraciones)

pygame.quit()
```

C.2. Algoritmo A* en 2D

Código en Blender:

```
import bpy
from math import *
import random
import time
import os

def Astar(grid, inicio, fin):

    visita = [[0] *len(grid[0]) for i in grid]
    visita[inicio[0]][inicio[1]] = 1
    paso = [[-1] *len(grid[0]) for i in grid]

    accion = [[-1] *len(grid[0]) for i in grid]

    heuristic = [[0] * len(grid[0]) for row in grid]
    for x in range(len(grid)):
        for y in range(len(grid[0])):
            dist=abs(x-fin[0])+abs(y-fin[1])
```

```
            heuristic[x][y]=dist

    x = inicio[0]
    y = inicio[1]
    g = 0
    h = heuristic[x][y]
    f = g+h

    lista = [[f, g, h, x, y]]
    paso[x][y] = g
    guardapaso=[]

    file = open(os.path.splitext(bpy.data.filepath)[0] + "variables.py", 'w')

    completada = False
    fracasada = False
    contador = 0

    while not completada and not fracasada:
        if len(lista) == 0:
            fracasada = True
            print("Camino no encontrado")
        else:
            lista.sort()
            lista.reverse()

            siguiente = lista.pop()
            x = siguiente[3]
            y = siguiente[4]
            g = siguiente[1]
            paso[x][y]=contador

            guardapaso.append([x,y])
            contador += 1

            if x == fin[0] and y == fin[1]:
                completada = True
            else:
```

```
            for i in range(len(movimiento)):
                x2 = x + movimiento[i][0]
                y2 = y + movimiento[i][1]

                if x2 >= 0 and x2 < len(grid) and y2 >=0 and y2 <len(grid[0]):
                    if visita[x2][y2] == 0 and grid[x2][y2] == 0:
                        g2 = g + coste
                        h2 = heuristic[x2][y2]
                        f2 = g2 + h2
                        lista.append([f2, g2, h2, x2, y2])
                        visita[x2][y2] = 1
                        accion[x2][y2] = i

    esquema= [[''] *len(grid[0]) for i in grid]
    esquema[fin[0]][fin[1]]='*'

    ruta=[]
    ruta.append([fin[0], fin[1]])

    while x != inicio[0] or y != inicio[1]:
        x2 = x - movimiento[accion[x][y]][0]
        y2 = y - movimiento[accion[x][y]][1]
        esquema[x2][y2] = flecha[accion[x][y]]
        x=x2
        y=y2

        ruta.append([x2, y2])

    ruta.reverse()

    return ruta, guardapaso , fracasada


def escenario(grid,inicio,fin):

    bpy.ops.object.select_by_type(type='MESH')
    bpy.ops.object.delete(use_global=False)
```

```
    bpy.ops.mesh.primitive_plane_add(location=((len(grid)/2)-0.5,(len(grid[0])/2)-0.5,-0.5))
    bpy.ops.transform.resize(value=(len(grid)/2, len(grid[0])/2, 0))

    plano=bpy.context.scene.objects.active
    materialplano= bpy.data.materials.new(plano.name)
    materialplano.diffuse_color = (1,1,1)

    materialcubo= bpy.data.materials.new(plano.name)
    materialcubo.diffuse_color = (0,0.5,1)

    plano.data.materials.append(materialplano)

    for x in range(len(grid)):
        for y in range(len(grid[0])):
            if grid[x][y] == 1:
                bpy.ops.mesh.primitive_cube_add(location=(x,y,0), radius=0.5)
                bpy.context.scene.objects.active.data.materials.append(materialcubo)


def animacion(grid, ruta, inicio, fin):

    x = inicio[0]
    y = inicio[1]
    z = 0
    frame = 1
    i = 0

    bpy.ops.mesh.primitive_monkey_add(radius=0.3, enter_editmode=False, location=(x, y, 0))
    mono = bpy.context.object

    materialmono= bpy.data.materials.new(mono.name)
    materialmono.diffuse_color = (1,0,0)
    mono.data.materials.append(materialmono)

    while x!= fin[0] or y != fin[1]:

        mono.keyframe_insert(data_path="location", frame=frame)
```

```
        x=ruta[i][0]
        y=ruta[i][1]
        z=0

        mono.location[2] = z
        mono.location[1] = y
        mono.location[0] = x
        i+=1
        frame += 10
        mono.keyframe_insert(data_path='location', frame=frame)

    print("Creando el monito")
    bpy.data.scenes["Scene"].frame_end = frame+15


def dibujoruta(ruta):

    for i in range(len(ruta)):
        bpy.ops.mesh.primitive_uv_sphere_add(size=.1,location=(ruta[i][0],ruta[i][1],0))
        materialbolas = bpy.data.materials.new("Nombre del material")
        materialbolas.diffuse_color=(0,1,0)
        bpy.context.scene.objects.active.data.materials.append(materialbolas)


#*******-------------------------------------------------------
#Programa principal

dimx=20
dimy=20

grid = [[round(random.random()-.2) for y in range(dimy)] for x in range(dimx)]
grid1 = [[0 for y in range(dimy)] for x in range(dimx)]

inicio=[0,0]
fin = [15,18]

grid[inicio[0]][inicio[1]] = 0
grid[fin[0]][fin[1]]= 0
```

```
movimiento = [[-1, 0], # subir
          [0, -1], # izquierda
          [1, 0], # bajar
          [0, 1]] # derecha

flecha = ['^', '<', 'v', '>']
coste = 1



escenario(grid, inicio, fin)
camino, busqueda,fracaso  =  Astar(grid, inicio, fin)
if fracaso == True:
   print("Camino no encontrado")
else:
   print("Camino encontrado")
   mueve=animacion(grid, camino, inicio, fin)
   dibujoruta(camino)



file = open(os.path.splitext(bpy.data.filepath)[0] + "variables.py", 'w')
file.write('inicio =' + str(inicio).strip('') + '\n')
file.write('fin =' + str(fin).strip('') + '\n')
file.write('grid =' + str(grid).strip('') + '\n')
file.write('busqueda =' + str(busqueda).strip('') + '\n')
file.write('camino =' + str(camino).strip('') + '\n')
file.write('iteraciones=' + str(len(busqueda)).strip('') + '\n')
file.close()
```

Código en Python:

```
import pygame
import Astarvariables

# Definimos algunos colores
NEGRO = (0, 0, 0)
BLANCO = (255, 255, 255)
VERDE = ( 0, 255, 0)
```

```
ROJO = (255, 0, 0)
AMARILLO = (255, 255,0)
AZUL = (0, 255, 255)

ini = Astarvariables.inicio
fin = Astarvariables.fin
grid = Astarvariables.grid

fil=len(grid)
col=len(grid[0])

# Establecemos el margen entre las celdas y el tamaño del lado
MARGEN = 1
lado = 8

# Inicializamos pygame
pygame.init()

# Establecemos el LARGO y ALTO de la pantalla, DEJARLO FIJO
DIMENSION_VENTANA = [lado*col,lado*fil]
pantalla = pygame.display.set_mode(DIMENSION_VENTANA)

# Establecemos el LARGO y ALTO de cada celda de la retícula.
LARGO  = lado-MARGEN
ALTO = lado-MARGEN

# Establecemos el título de la pantalla.
pygame.display.set_caption("Mapa")

# Iteramos hasta que el usuario pulse el botón de salir.
hecho = False

# Dibujamos la retícula
pantalla.fill(NEGRO)
for fila in range(fil):
    for columna in range(col):
        color = BLANCO
        if grid[fila][columna]==1:
```

```
          color = NEGRO
        pygame.draw.rect(pantalla,color,[(MARGEN+LARGO) * columna + MARGEN,(MARGEN+ALTO) *
fila + MARGEN, LARGO, ALTO])


pygame.draw.rect(pantalla,ROJO,[(MARGEN+LARGO) * ini[1] + MARGEN,(MARGEN+ALTO) * ini[0]+
MARGEN, LARGO, ALTO])
pygame.draw.rect(pantalla,BLANCO,[(MARGEN+LARGO) * fin[1] + MARGEN,(MARGEN+ALTO) *
fin[0]+ MARGEN, LARGO, ALTO])
pygame.time.wait(500)
##########################################################################################


# Lo usamos para establecer cuán rápido de refresca la pantalla.
reloj = pygame.time.Clock()
x=0
tabla=Astarvariables.busqueda
contador=0


# -------- Bucle Principal del Programa-----------
while not hecho and x<len(tabla):
    for evento in pygame.event.get():
        if evento.type == pygame.QUIT:
            hecho = True


   # Establecemos el fondo de pantalla.
    contador=contador+1
    pygame.time.wait(2)
    fila = tabla[x][0]
    columna = tabla[x][1]
    pygame.draw.rect(pantalla,VERDE,[(MARGEN+LARGO) * columna + MARGEN,(MARGEN+ALTO) *
fila + MARGEN, LARGO, ALTO])
    x=x+1


    # Limitamos a 20 fotogramas por segundo.
    reloj.tick(200)


    # Avanzamos y actualizamos la pantalla con lo que hemos dibujado.
    pygame.display.flip()


pygame.draw.rect(pantalla,ROJO,[(MARGEN+LARGO) * ini[1] + MARGEN,(MARGEN+ALTO) * ini[0]+
```

```
MARGEN, LARGO, ALTO])

camino=Astarvariables.camino
x=0
contador=0
# -------- Bucle Para hacer el camino-----------
while not hecho and x<len(camino):
    for evento in pygame.event.get():
        if evento.type == pygame.QUIT:
            hecho = True

   # Establecemos el fondo de pantalla.
    contador= contador+1
    pygame.time.wait(20)
    fila = camino[x][0]
    columna = camino[x][1]
    pygame.draw.rect(pantalla,AMARILLO,[(MARGEN+LARGO) * columna + MARGEN, (MARGEN +
ALTO) * fila + MARGEN, LARGO, ALTO])
    x=x+1

    # Limitamos a 20 fotogramas por segundo.
    reloj.tick(200) #con este marco la velocidad, a más grande más rápido
    # Avanzamos y actualizamos la pantalla con lo que hemos dibujado.

pygame.display.flip()
iteraciones=Astarvariables.iteraciones
print(iteraciones)
pygame.quit()
```

C.3. Algoritmo RRT en 2D

Código en Blender:

```
import bpy
from math import *
import random
import time
import os
```

```
dimx=10
dimy=10

grid2 = [[0 for y in range(dimy)] for x in range(dimx)]
grid = [[round(random.random()-.4) for y in range(dimy)] for x in range(dimx)]

#Inicio y final
inicio=[5,5]
fin = [8,8]
fin2=[round(random.random()*(len(grid)-1)), round(random.random()*(len(grid[0])-1))]

grid[inicio[0]][inicio[1]] = 0
grid[fin[0]][fin[1]]= 0

visita = [[0] *len(grid[0]) for i in grid] #matriz como grid todo a 0
visita[inicio[0]][inicio[1]] = 1    #matriz con la posición inicial = 1
paso = [[-1] *len(grid[0]) for i in grid] #Tabla donde ire guardando en que pasos se llega aqui
accion = [[-1] *len(grid[0]) for i in grid]

busqueda=[]
busqueda.append([inicio[0], inicio[1]])

alcance=6

completada=False

def camino(p1,p2):

   x = p1[0]
   y = p1[1]
   movx = 0
   movy = 0
   pasosx = abs(p2[0]-p1[0])
   pasosy = abs(p2[1]-p1[1])

   if (p2[0]-p1[0]) >=0 :
     siguientex = 1
```

```
   accionx='^'
else:
   siguientex = -1
   accionx='v'
if p2[1]-p1[1] >=0 :
    siguientey = 1
    acciony='<'
else:
     siguientey=-1
     acciony='>'

camino1= True
camino2= True
completada=False

while (movx < pasosx) and camino1 == True and completada==False:
    x=x+siguientex  #la coordenada x aumenta o disminuye
    movx=movx+1     #asi se cuantos pasos tengo que dar haia derecha o izquierda
    if grid[x][y] == 1:
        camino1= False
    else:
        if x == fin[0] and y == fin[1]:
            completada = True
            accion[x][y]=accionx
            print("lo encontreeeeeee")
        else:
            if accion[x][y]==-1:
                accion[x][y]=accionx
            visita[x][y] = 1
            busqueda.append([x,y])

while (movy < pasosy) and camino1 == True and completada==False:
    y=y+siguientey
    movy=movy+1
    if grid[x][y] == 1:
        camino1= False
    else:
        if x == fin[0] and y == fin[1]:
```

```
            completada = True
            accion[x][y]=acciony
            print("Lo encontreee")
        else:
            if accion[x][y]==-1:
                accion[x][y]=acciony
            visita[x][y] = 1
            busqueda.append([x,y])

if camino1 == False:
    x = p1[0]
    y = p1[1]
    movx=0
    movy=0
    print("Intento otra ruta")

    while (movy < pasosy) and camino2 == True:
        y=y+siguientey
        movy=movy+1
        if grid[x][y] == 1:
            camino2= False
            y=y-siguientey
        else:
            if x == fin[0] and y == fin[1]:
                completada = True
                accion[x][y]=acciony
                print("lo encontreeeeeee")
            else:
                if accion[x][y]==-1:
                    accion[x][y]=acciony
                visita[x][y] = 1
                busqueda.append([x,y])

    while (movx < pasosx) and camino2 == True:
        x=x+siguientex
        movx=movx+1
        if grid[x][y] == 1:
            camino2= False
```

```
            x=x-siguientex
        else:
            if x == fin[0] and y == fin[1]:
                completada = True
                accion[x][y]=accionx
                print("lo encontreeeeeee")
            else:
                if accion[x][y]==-1:
                    accion[x][y]=accionx
                visita[x][y] = 1
                busqueda.append([x,y])

    if camino2==False:
        print("no hay camino, mando la última coordenada")
        return [x,y], completada
    else:
        return[x,y], completada
  else:
    return [x,y], completada

def dist(p1,p2):
  return abs(p1[0]-p2[0])+abs(p1[1]-p2[1])



def step_from_to(p1,p2):
  if dist(p1,p2) <= alcance:
    return p2
  else:
    difx = p2[0]-p1[0]
    dify = p2[1]-p1[1]
    dif= abs(difx) + abs(dify)
    siguiente = p1[0] + round((difx/dif)*alcance), p1[1] + round((dify/dif)*alcance)
    return siguiente

def escenario(grid,inicio,fin):

  #Borramos el escenario que había antes
  bpy.ops.object.select_by_type(type='MESH')
```

```
bpy.ops.object.delete(use_global=False)

#Creamos tablero con posición y tamaño adecuado
bpy.ops.mesh.primitive_plane_add(location=((len(grid)/2)-0.5,(len(grid[0])/2)-0.5,-0.5))
bpy.ops.transform.resize(value=(len(grid)/2, len(grid[0])/2, 0))

#Seleccionamos el plano
plano=bpy.context.scene.objects.active

#Creamos un material para el plano
materialplano= bpy.data.materials.new(plano.name)
materialplano.diffuse_color = (1,0,1)



#Creamos un material para el laberinto
materialcubo= bpy.data.materials.new(plano.name)
materialcubo.diffuse_color = (0,1,1)



#Le añadimos el material al plano
plano.data.materials.append(materialplano)



for x in range(len(grid)):
    for y in range(len(grid[0])):
        if grid[x][y] == 1:
            bpy.ops.mesh.primitive_cube_add(location=(x,y,0), radius=0.5)
            bpy.context.scene.objects.active.data.materials.append(materialcubo)

            #bpy.context.scene.objects.active = bpy.context.selected_objects[0]
            #if not cubo.material_slots.values():
                #cubo = bpy.context.scene.objects.active
                #name = cubo.name
                #material= bpy.data.materials.new(name)
                #cubo.data.materials.append(materialcubo)
            #else:
                #material=cubo.material_slots[0].material
```

```
        #material.diffuse_color = (0,0,1)
    print("Laberinto")



#Animación del monito
def animacion(grid, ruta, inicio, fin):

    #Primero las coordenadas
    x = inicio[0]
    y = inicio[1]
    z = 0
    frame = 1
    i = 0

    #Añadimos el mono
    bpy.ops.mesh.primitive_monkey_add(radius=0.3, location=(x, y, 0))
    mono = bpy.context.object
    print("Creando el monito")
    #Creamos un material para el mono
    materialmono= bpy.data.materials.new(mono.name)
    materialmono.diffuse_color = (1,1,0)
    mono.data.materials.append(materialmono)
    print("Creando el material")

    while x!= fin[0] or y != fin[1]:
        #frame += 1
        print("nueva ubicacio´")
        mono.keyframe_insert(data_path="location", frame=frame)

        #Posición
        x=ruta[i][0]
        y=ruta[i][1]
        z=0

        #print(i, len(path), y, goal[1], x, goal[0])
        mono.location[2] = z
        mono.location[1] = y
        mono.location[0] = x
```

```
        i+=1
        frame += 10
        mono.keyframe_insert(data_path='location', frame=frame)

    bpy.data.scenes["Scene"].frame_end = frame+15



#*******-----------------------------------------------------
#Programa principal

nodes = []
nodes.append(inicio)
numnodos=dimx*dimy #len(grid[0])*len(grid)
i=0
alcanzado =False

while i<numnodos and alcanzado==False:

    print("NODOS")
    i=i+1
    print(nodes)

    rand = round(random.random()*(len(grid)-1)), round(random.random()*(len(grid[0])-1))
    while grid[rand[0]][rand[1]]==1 and visita[rand[0]][rand[1]]==1:
        rand = round(random.random()*(len(grid)-1)), round(random.random()*(len(grid[0])-1))

    nn = nodes[0]
    for p in nodes:                 #en estas tres lineas selecccion el nodo ma´ cercano al
        if dist(p,rand) < dist(nn,rand):
                nn = p
    newnode = step_from_to(nn,rand)
    nuevo, alcanzado =camino(nn,newnode)
    nodes.append(nuevo)

print(nodes)
j=0
ruta=[]
x=nuevo[0]
```

```
y=nuevo[1]
ruta.append([x,y])

if alcanzado==True:
    while x!=inicio[0] or y!=inicio[1]: #and j<50
        j=j+1
        if accion[x][y] == '^':
            x=x-1
        else:
            if accion[x][y] == 'v':
                x=x+1
            else:
                if accion[x][y] == '<':
                    y=y-1
                else:
                    if accion[x][y] == '>':
                        y=y+1

        ruta.append([x,y])
    ruta.reverse()

busquedanew = []
for i in busqueda:
    if i not in busquedanew:
        busquedanew.append(i)

escenario(grid,inicio,fin)
animacion(grid, ruta, inicio, fin)



#Para el pygame
file = open(os.path.splitext(bpy.data.filepath)[0] + "variables.py", 'w')
file.write('inicio =' + str(inicio).strip('') + '\n')
file.write('fin =' + str(fin).strip('') + '\n')
file.write('busqueda =' + str(busqueda).strip('') + '\n')
file.write('grid =' + str(grid).strip('') + '\n')
file.write('nodes =' + str(nodes).strip('') + '\n')
file.write('ruta =' + str(ruta).strip('') + '\n')
```

```
file.write('iteraciones=' + str(len(busquedanew)).strip('') + '\n')
file.close()
```

Código en Pygame:

```
import pygame
import rrtvariables

# Definimos algunos colores
NEGRO = (0, 0, 0)
BLANCO = (255, 255, 255)
VERDE = ( 0, 255, 0)
ROJO = (255, 0, 0)
AMARILLO = (255, 255,0)
TURQUESA = ( 0, 255, 255)
AZUL = (0, 255, 255)
MM = (255, 0,255)

ini = rrtvariables.inicio
fin = rrtvariables.fin
grid = rrtvariables.grid

fil=len(grid)
col=len(grid[0])

# Establecemos el margen entre las celdas y el tamaño del lado
MARGEN = 0
lado = 7
# Inicializamos pygame
pygame.init()

# Establecemos el LARGO y ALTO de la pantalla, DEJARLO FIJO
DIMENSION_VENTANA = [lado*col,lado*fil]
pantalla = pygame.display.set_mode(DIMENSION_VENTANA)

# Establecemos el LARGO y ALTO de cada celda de la retícula.
LARGO  = lado-MARGEN
ALTO = lado-MARGEN
```

```
# Establecemos el título de la pantalla.
pygame.display.set_caption("Mapa")

# Iteramos hasta que el usuario pulse el botón de salir.
hecho = False

# Dibujamos la retícula
pantalla.fill(NEGRO)
for fila in range(fil):
    for columna in range(col):
        color = BLANCO
        if grid[fila][columna]==1:
            color = NEGRO
        pygame.draw.rect(pantalla,color,[(MARGEN+LARGO) * columna + MARGEN,(MARGEN+ALTO) *
fila + MARGEN, LARGO, ALTO])

pygame.draw.rect(pantalla,ROJO,[(MARGEN+LARGO) * ini[1] + MARGEN,(MARGEN+ALTO) * ini[0]+
MARGEN, LARGO, ALTO])
pygame.draw.rect(pantalla,ROJO,[(MARGEN+LARGO) * fin[1] + MARGEN,(MARGEN+ALTO) * fin[0]+
MARGEN, LARGO, ALTO])
pygame.time.wait(500)

reloj = pygame.time.Clock()
contador=0
nodos=rrtvariables.nodes

x=0
tabla=rrtvariables.busqueda

contador=0
# -------- Bucle Principal del Programa-----------
while not hecho and x<len(tabla):
    for evento in pygame.event.get():
        if evento.type == pygame.QUIT:
            hecho = True

   # Establecemos el fondo de pantalla.
    contador=contador+1
    pygame.time.wait(1) #esta variable controla la velocidad
```

```
    fila = tabla[x][0]
    columna = tabla[x][1]
    pygame.draw.rect(pantalla,VERDE,[(MARGEN+LARGO) * columna + MARGEN,(MARGEN+ALTO) * fila + MARGEN, LARGO, ALTO])
    x=x+1

    reloj.tick(500)

    pygame.display.flip()

ruta=rrtvariables.ruta
x=0
contador=0
# -------- NODOOOS-----------
while not hecho and x<len(nodos):
    for evento in pygame.event.get():
        if evento.type == pygame.QUIT:
            hecho = True

    contador=contador+1
    print("DIBUJO LOS NODOS")
    pygame.time.wait(1)
    fila = nodos[x][0]
    columna = nodos[x][1]
    #print(fila,columna)
    pygame.draw.rect(pantalla,AZUL,[(MARGEN+LARGO) * columna + MARGEN,(MARGEN+ALTO) * fila + MARGEN, LARGO, ALTO])
    x=x+1

    reloj.tick(1000)

    pygame.display.flip()

x=0
contador=0

# -------- RUTA----------
while not hecho and x<len(ruta):
```

```
    for evento in pygame.event.get():
        if evento.type == pygame.QUIT:
            hecho = True

    # Establecemos el fondo de pantalla.
    contador=contador+1
    pygame.time.wait(5) #esta variable controla la velocidad
    fila = ruta[x][0]
    columna = ruta[x][1]
    pygame.draw.rect(pantalla,AMARILLO,[(MARGEN+LARGO) * columna + MARGEN,(MARGEN+ALTO) * fila + MARGEN, LARGO, ALTO])
    x=x+1

    reloj.tick(50)
    pygame.display.flip()

iteraciones=rrtvariables.iteraciones
print(iteraciones)
```

Pórtate bien con el IDLE.