



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería en Informática



**TFG del Grado en Ingeniería Informática
Algoritmos de búsqueda en visor 3D**



Presentado por Víctor Pérez Esteban
en Universidad de Burgos — 30 de junio de 2017

Tutores: Dr. José Francisco Díez Pastor
Dr. César Ignacio García Osorio



**UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería en Informática**



D. Dr. José Francisco Díez Pastor y D. Dr. César Ignacio García Osorio, profesores del departamento de Ingeniería Civil, área de Lenguajes y Sistemas Informáticos.

Exponen:

Que el alumno D. Víctor Pérez Esteban, con DNI 71279579A, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado «Algoritmos de búsqueda en visor 3D».

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección de los que suscriben, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 30 de junio de 2017

Vº. Bº. del Tutor:

D. Dr. José Francisco Díez Pastor

Vº. Bº. del Tutor:

D. Dr. César Ignacio García Osorio

Resumen

En los últimos años hemos visto que la evolución de los sistemas inteligentes se hace cada vez más presente en la sociedad. Uno de los sistemas inteligentes en que más se han observado esfuerzo en el desarrollo, inversiones por las grandes compañías y relevancia social han sido la aparición de los primeros prototipos de los vehículos autónomos, coches que se conducen solos.

Para la realización de estos sistemas inteligentes se tienen que combinar muchos procedimientos que doten del conocimiento necesario al sistema inteligente sobre su entorno, y que permitan, una vez adquirido, su utilización para desplazarse a su destino. Dos de estos procedimientos principales son el cómo conocer la ruta a seguir, y una vez conocida la ruta cómo seguirla, hasta alcanzar su final.

En este trabajo estudiaremos estos procesos en un entorno tridimensional, para poder simular algunos de los problemas que se encuentran los vehículos autónomos en el mundo real y tener una visión de como solventar los problemas que plantea. Se calcularán rutas que se puedan seguir por parte del vehículo y se implementarán métodos para seguir esas rutas de forma autónoma, viendo en cada paso los problemas que se presentan.

Descriptores

Búsqueda de rutas, vehículos autónomos, seguimiento de rutas

Abstract

In recent years, we have seen the evolution of intelligent systems that are beginning to be present in society. One of these intelligent systems that has experimented the bigger development effort, investments by large companies and social relevance has been the first prototypes of autonomous vehicles, cars that are driven by themselves.

For the realization of these intelligent systems many procedures have to be combined to provide the intelligent system with the necessary knowledge about its environment and to allow it to be used once it has reached its destination. Two of these main procedures are how to know the route to follow, and once the route is known, how to follow it until its end is reached.

In this work, we will investigate these processes in a three-dimensional environment, in order to simulate some of the problems that autonomous vehicles find in the real world and to have a vision of how to solve the problems that it poses. Routes will be calculated that can be followed by the vehicle and will be implemented methods to follow these routes autonomously, seeing at each step the problems that arise.

Keywords

Routes searching, autonomous vehicles, tracking routes.

Índice general

Índice general	III
Índice de figuras	v
Índice de tablas	vi
Introducción	1
Objetivos del proyecto	3
Conceptos teóricos	4
3.1. Algoritmo A*	4
3.2. <i>Path Smoothing</i> o Suavizado	7
3.3. Theta*	13
3.4. A* con vértices	14
3.5. <i>Hybrid A*</i>	15
3.6. <i>PID Controller</i>	23
Técnicas y herramientas	26
4.1. Programas usados	26
Aspectos relevantes del desarrollo del proyecto	28
5.1. <i>Navigation Mesh</i>	28
5.2. Cola de prioridad	29
5.3. Física del vehículo	30
5.4. Modelos utilizados	31
5.5. Comparativa de los algoritmos	31
Trabajos relacionados	33
Conclusiones y Líneas de trabajo futuras	34

ÍNDICE GENERAL

IV

7.1. Conclusiones	34
7.2. Líneas de trabajo futuras	34
Bibliografía	36

Índice de figuras

3.1.	Ejemplo de suavizado usando descenso gradiente	7
3.2.	Comparación entre la ruta del A* con y sin zigzag	8
3.3.	Efecto de α y β en el descenso gradiente	10
3.4.	Esquema de una curva Bézier cuadrática de Herman Tulleken	11
3.5.	Esquema de una curva Bézier cuadrática, Wikipedia	12
3.6.	Comparativa de rutas con A* y con curvas Bézier	12
3.7.	Comparación de los caminos elegidos por el A* y el Theta*	13
3.8.	Comparación entre los sucesores del A* y del A* usando los vértices	15
3.9.	Esquema del modelo de la bicicleta	16
3.10.	Representación del mapa de obstáculos	20
3.11.	Representación del mapa de distancias	21
3.12.	Esquema de un <i>PID controller</i>	24
5.13.	Imagen de un <i>Nav Mesh</i> en <i>Unity</i>	29
5.14.	Representación de los <i>Wheel Colliders</i> en <i>Unity</i>	30

Índice de tablas

5.1. Comparación de los algoritmos según su representación	32
5.2. Comparación de los algoritmos según su rendimiento	32

Introducción

El vehículo autónomo es uno de los avances tecnológicos que más impacto va a tener en la sociedad en los próximos años. Tal es así, que las mayores compañías ya están trabajando en el cambio que se avecina.

Brian Krzanich, director de Intel declaró [1] que el cambio será tan explosivo que las compañías que no se preparen, se enfrentarán al fracaso y a la extinción. Se esperá que las cantidades de dinero que mueva este nuevo sector sean gigantescas, 800 mil millones de dolares para el año 2035 y hasta 7 billones de dolares en los siguientes 15 años [1].

Este nuevo sector no solo involucra a la industria automovilística y sus empresas tradicionales. Empresas del sector de la informática, como Intel, Nvidia, Google, Apple están invirtiendo en el desarrollo de las tecnologías necesarias. Y muchas empresas más pequeñas que trabajan en tecnologías necesarias para el coche autónomo han sido compradas por gigantes tecnológicos, como Mobileye, comprada por Intel, especializada en visión computarizada, Primesense, comprada por Apple, especializada en los sensores de tres dimensiones, o Waze, comprada por Google, especializada en los mapas para la navegación GPS [2].

Pero no solo es una revolución en la economía. Supondrá también un cambio social. Los vehículos autónomos tienen menos accidentes que los humanos [3], son más ecológicos al reducir las emisiones de CO_2 [4], y cambiarán la manera de la sociedad de desplazarse. Los coches autónomos no solo permiten usar el tiempo que las personas pasan al volante en otras actividades, también permiten más independencia en sus desplazamientos a personas mayores o con discapacidad. Cambia la forma en que la gente puede compartir vehículos y desplazamientos, como lo ha hecho Uber. El espacio de las ciudades destinado a aparcamiento podría cambiar drásticamente con vehículos que se aparcan solos. También se verá una reducción del tráfico y la congestión en las carreteras debido a su conducción más eficiente.

Como hemos mencionado, el número de tecnologías necesarias para cons-

truir un coche autónomo es enorme. Desde el propio vehículo, todos los sistemas para la detección del entorno y la visión computarizada, hasta todo el software necesario que hay que desarrollar. Algoritmos que permitan la percepción, conocer el entorno y las variables imprevistas que surjan. Algoritmos que permiten conocer la localización exacta en tiempo real. Algoritmos que permitan planificar las acciones que debe llevar a cabo el vehículo. Y algoritmos que permitan la locomoción del vehículo y desplazarlo siguiendo la planificación.

De todas estas tecnologías, en este proyecto nos hemos fijado en las dos últimas. Al usar un entorno tridimensional para realizar la simulación, podemos conocer el entorno en el que se encuentra el vehículo, así como su localización. De esta forma, podemos centrarnos en el desarrollo de algoritmos que permitan planificar una ruta y que le indiquen a un vehículo como desplazarse de forma autónoma en un entorno continuo en un espacio tridimensional, como ocurre en la realidad. También nos permite ver las nuevas dificultades que plantea y adaptar los algoritmos que hemos ido viendo en la carrera en un entorno más realista, así como otros nuevos que permiten entender y solventar los problemas a los que se enfrentan los coches autónomos que circulan por las calles.

Objetivos del proyecto

Los objetivos del proyectos son:

1. Crear un espacio tridimensional dónde poder representar y probar el desarrollo de un vehículo autónomo.
2. Desarrollar algoritmos que permitan la búsqueda de rutas desde la posición del vehículo hasta una meta.
3. Dotar el vehículo de autonomía para poder seguir las rutas calculadas por sí mismo, mediante mecanismos que permitan calcular el giro del volante y la fuerza que el motor debe aplicar a las ruedas para ajustarse a la ruta planificada.
4. Utilizar un motor 3D, *Unity*, aprender su uso y los elementos disponibles.
5. Aprender y entender las dificultades que entraña el desarrollo de los vehículos autónomos.

En el proyecto se implementarán los algoritmos que hemos considerado más interesantes, y se plantea de tal forma que sea posible añadir en el futuro nuevos algoritmos para poder mejorar la autonomía del vehículo.

Conceptos teóricos

En esta sección vamos a explicar los algoritmos utilizados en el desarrollo del proyecto. Primero veremos los algoritmos relativos a la planificación de la ruta, y a continuación aquellos relativos al seguimiento de la misma de una forma autónoma por parte del vehículo.

3.1. Algoritmo A*

A* es un algoritmo de búsqueda informada del tipo primero el mejor, que usa una función de evaluación para elegir hacia qué nodo expandirse desde un nodo inicial hacia un nodo final.

Para representar el espacio de búsqueda del algoritmo, se usa una cuadrícula, es por lo tanto un algoritmo que opera en un espacio de estados discreto. Cada nodo de la cuadrícula puede representar un espacio donde es posible desplazarse o un obstáculo que es inalcanzable.

La función F() de evaluación calcula el coste de cada nodo, con lo que se eligen los nodos con menor coste para llegar al destino a través del camino óptimo. Esta función está a su vez formada por dos funciones: una función G() que calcula el coste del camino seguido desde el nodo inicial a ese nodo concreto, y una función H() o función heurística que hace una estimación del coste del camino desde ese nodo al nodo final o meta.

El algoritmo comienza desde el nodo inicial explorando los nodos adyacentes o sucesores. Un nodo ya explorado, es decir del que ya se ha buscado sus sucesores, se manda a la lista de nodos cerrados. Con los sucesores se forma una lista de nodos abiertos o por explorar, de la cual se elige el de menor coste para ser el siguiente en ser explorado, hasta que se alcance el nodo meta o no queden más nodos por ser explorados (si no quedasen más nodos significaría que no hay un camino entre el nodo inicial y la meta). Si al explorar un nodo

esta ya se encontraba en alguna de las listas de nodos abiertos o cerrados, se actualizarán los valores de los nodos al de menor coste encontrado.

Heurística

El algoritmo A* es completo, lo que significa que encontrará un camino hasta la meta siempre que este exista. Además, para que sea admisible, que significa que siempre encontrará un camino óptimo, su función H() también debe ser admisible.

Una función H() es admisible siempre y cuando no sobreestime el coste del camino desde un nodo hasta la meta. Por ejemplo, se puede considerar que el camino desde un nodo hasta la meta será la línea recta.

La admisibilidad del A* trae consigo un gran coste computacional debido al gran número de nodos explorados. Para mejorar la eficiencia podemos dar pesos a las funciones G() y H(), de tal forma que si damos más valor a G() la búsqueda se expandirá en anchura buscando el camino, mientras que si damos más valor a H() se expandirá más rápido acercándose a la meta.

Para el A* y algunas de sus variantes, el Theta* [3.3](#) y el A* con vértices [3.4](#), hemos usado la distancia euclídea, tanto para la función G(), como para la función H(). La función G() es la distancia recorrida desde el inicio hasta ese punto, y la función H() es la distancia en línea recta desde donde nos encontremos a la meta.

Pseudocódigo A*

```

abiertos = cerrados =  $\emptyset$ ;
actual = inicio;
abiertos.añadir(actual, actual.coste);

while abiertos no esté vacío do
    actual = abiertos.primerElemento();
    cerrados.añadir (actual);
    if actual es la meta then
        return Camino encontrado
    else
        sucesores = obtenerSucesores(actual);
        forall los sucesores de actual do
            if sucesor está en abiertos then
                if coste en abiertos es mayor que el del sucesor then
                    | abiertos.actualizar (actual, actual.coste);
                end
            end
            if sucesor está en cerrados then
                if coste en cerrados es mayor que el del sucesor then
                    | cerrados.borrar (actual);
                    | abiertos.añadir (actual);
                end
            end
            if sucesor no está ni en abiertos ni en cerrado then
                | abiertos.añadir (actual);
            end
        end
    end
end

2 return Error: Camino no encontrado

```

Algorithm 1: Pseudocódigo del A*

3.2. *Path Smoothing* o Suavizado

Un problema del A* tradicional es que las rutas que encuentra, aunque sean las más cortas, no tienen una apariencia realista. Esto es debido a que el A* discretiza el espacio de búsqueda para poder buscar la ruta. Dependiendo de la representación que elijamos, se acercará más a o menos a la realidad, pero en cualquier caso se producirá una diferencia que suele estar alejada de la representación ideal de esa misma ruta.

Por ejemplo, en el caso de una búsqueda en una cuadrícula, si seguimos el camino a través de cada casilla, cuando el camino siga una ruta en diagonal, el A* seguirá un camino en zigzag. Aunque este camino es perfectamente válido, en la realidad no se sigue un camino en zigzag si no que se sigue la línea recta que representa. En el caso de las curvas el problema es parecido. El camino encontrado estará formado por segmentos rectos en vez de seguir una ruta suavizada.

Por este motivo, una vez obtenida la ruta, es necesario un proceso de suavizado de tal forma que se acerque la ruta obtenida a través del A*, a la representación ideal de esa ruta.

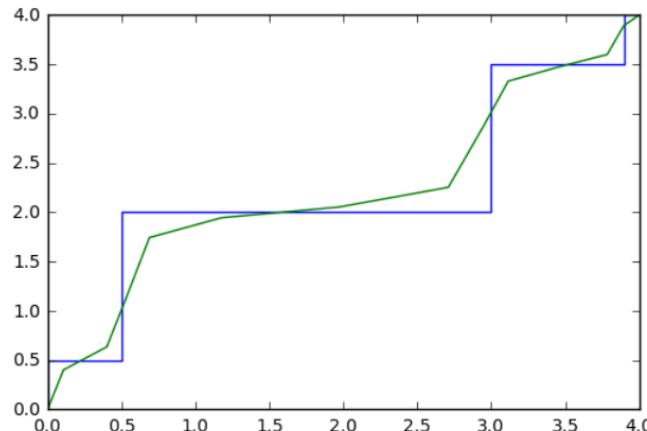


Figura 3.1: Ejemplo de suavizado usando descenso gradiente. La línea azul representa un camino encontrado por el A* y la línea verde el resultado de aplicarle el suavizado

Eliminar el zigzag

El primer método que hemos usado y que suele dar buenos resultados, es eliminar el zigzag. Este método es básicamente lo mismo que realiza el Theta* que explicaremos más adelante en 3.3.

El zigzag es el problema más habitual que nos hemos encontrado. Al representar un espacio continuo a través de casillas, se produce habitualmente porque aunque se use la representación octal permitiendo el movimiento diagonal, esto al pasarlo a un espacio no discreto quiere decir que sólo tenemos ocho posibles ángulos de movimiento: 0° , 45° , 90° , 135° , 180° , 225° , 270° y 315°

En realidad las posibilidades reales para movernos son cualquier ángulo de los 360° . Por tanto, si nuestro objetivo está en un ángulo diferente a esos ocho, se produce un zigzag combinándoles hasta que se consigue llegar.

La forma para eliminar el zigzag, es comprobar si es posible eliminar estos pasos intermedios. Es decir, si para llegar al objetivo, el A* ha usado varias casillas en el espacio discreto, es posible que en un espacio no discreto pudiéramos llegar directamente con un movimiento en un ángulo distinto a los ocho que permite el A*.

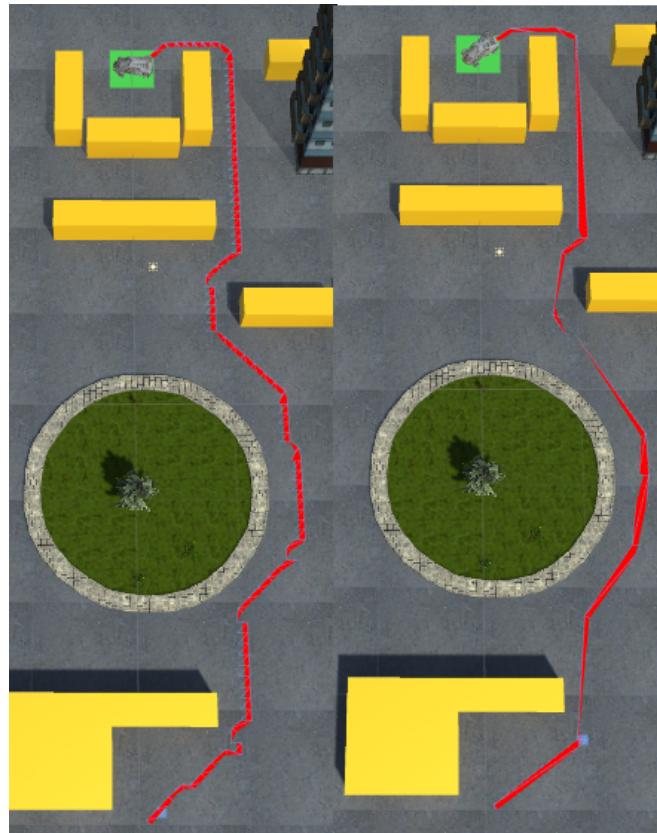


Figura 3.2: A la izquierda la ruta obtenida con el A*, y a la derecha la ruta después de eliminar el zigzag.

Para ello, usamos la línea de visión. Si existe una línea de visión entre

una casilla del A* y otra quiere decir que podemos desplazarnos siguiendo una línea recta formada entre esas dos casillas. Con las casillas devueltas por el A*, comprobamos si existe línea de visión entre ellas. Entre casillas consecutivas siempre habrá línea de visión, porque si no, no sería un camino válido, pero puede también haber línea de visión entre las siguientes de forma que haya casillas sobrantes debido al zigzag y a la forma que tiene el A* de buscar el camino.

Así que lo que comprobamos es que existe una línea de visión entre una casilla del A* y la siguiente más lejana posible, y eliminamos a las casillas intermedias. De esta forma obtenemos una línea recta entre esas dos casillas y eliminamos el zigzag producido por las casillas intermedias.

Descenso gradiente

El descenso gradiente es un algoritmo iterativo de optimización que a partir de N valores iniciales $(x_1, x_2, x_3, \dots, x_n)$, itera una función modificando esos valores gradualmente hacia un mínimo de una función $f(x_1, x_2, x_3, \dots, x_n)$. El algoritmo termina cuando el cambio que se producen en los valores es menor que la tolerancia indicada.

Para optimizar la ruta, queremos obtener los valores que minimizan la distancia entre los puntos sucesivos de la ruta, y a su vez que minimicen la distancia a la ruta original.

Para minimizar la distancia entre los puntos con respecto a la ruta original usaremos la función:

$$y_i = y_i + \alpha(x_i - y_i)$$

Donde α es el peso que le damos a cuanto de cerca queremos que la ruta suavizada esté de la ruta original, x es el punto de la ruta original e y es el nuevo punto suavizado.

Para minimizar la distancia entre los puntos sucesivos de la ruta, usamos la función:

$$y_i = y_i + \beta(y_{i+1} + y_{i-1} - 2 * y_i)$$

Donde β es el peso que le damos al suavizado de la ruta, y tenemos en cuenta la distancia tanto con el punto anterior como con el punto siguiente.

Para calcular el error que se produce, o el cambio hacia el mínimo, usamos la función:

$$\text{error} = \text{error} + (z_i - y_i)$$

Donde z es el valor de y antes de calcular el nuevo mínimo.

En la figura 3.3 podemos ver los distintos resultados que obtenemos al usar distintos valores en α y β . El resultado que se observa más a la izquierda corresponde a $\alpha = 0,9$ y $\beta = 0,1$, con lo que obtenemos la ruta original. En el centro podemos ver el caso opuesto, con $\alpha = 0,0$ y $\beta = 0,3$, donde el camino más suavizado posible es la línea recta hasta la meta. Y por último, a la derecha, el resultado con $\alpha = 0,5$ y $\beta = 0,4$, que es bastante próximo al resultado de una ruta ideal.

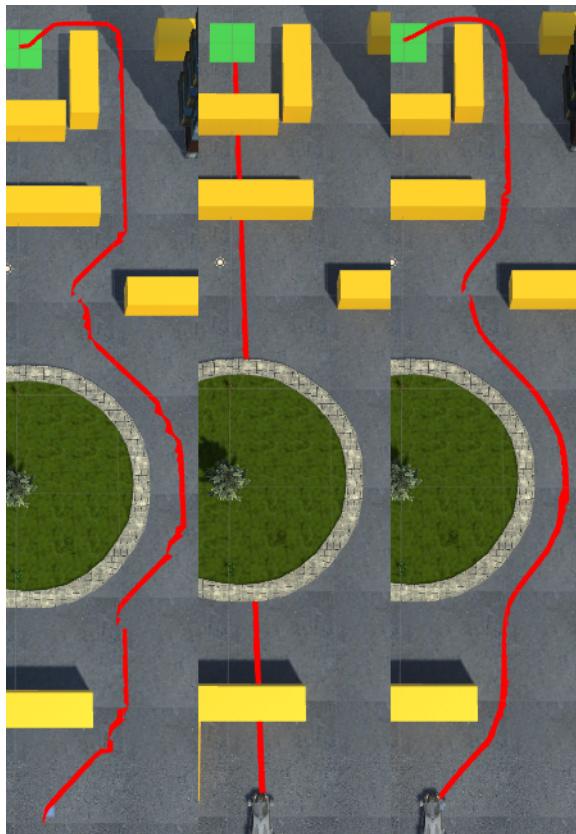


Figura 3.3: Comparación entre los resultados del descenso gradiente con distintos α y β .

Curvas Bézier

Las curvas Bézier [5, 6] son una forma de representar curvas a través de una función que recibe un parámetro. Hemos usado la función de las curvas de Bézier para suavizar la ruta y así obtener giros y curvas más cercanas a una representación real

Una función de una curva Bézier tiene varios puntos de control (al menos dos). De estos puntos, el primero y el último representan el inicio y el final de la curva. Esta función además recibe un parámetro que puede tomar los valores comprendidos entre cero y uno. Si el parámetro toma el valor cero, entonces la función devuelve el punto de inicio, mientras que si toma el valor uno devuelve el punto final. De esta forma, dando valores entre cero y uno, la función devuelve los valores que se encuentran entre los puntos de inicio y de fin.

Si la función toma dos puntos de referencia, lo que obtendremos sera una recta y sus puntos intermedios. Si la función toma tres puntos, entonces tendremos una curva donde el vértice será cercano al punto intermedio. Podemos usar más puntos para representar curvas más complejas o dar curvas con más variedad de formas.

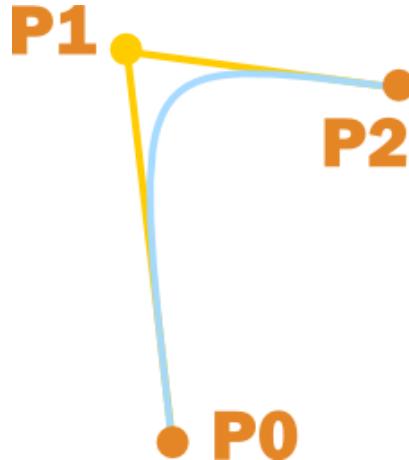


Figura 3.4: Esquema de una curva Bézier cuadrática de Herman Tulleken, Bézier Curves for your Games: A Tutorial [7].

Hemos usado la función Bézier con tres puntos después de eliminar el zig-zag, así que cada tres puntos del A* sin zigzag se usan como puntos de control que servirán para crear la curva. Si los puntos más o menos están alineados, el resultado será una recta suavizada, y si forman una curva, se eliminan los segmentos rectos y se reemplazan con puntos que forman una curva.

La función Bézier cuadrática para tres puntos de control y parámetro t es:

$$[x, y, z] = (1 - t)^2 P_0 + 2(1 - t)tP_1 + t^2 P_2$$

En formato expandido:

$$x = (1 - t)^2 x_0 + 2(1 - t)t x_1 + t^2 x_2$$

$$y = (1 - t)^2 y_0 + 2(1 - t)t y_1 + t^2 y_2$$

$$z = (1 - t)^2 z_0 + 2(1 - t)t z_1 + t^2 z_2$$

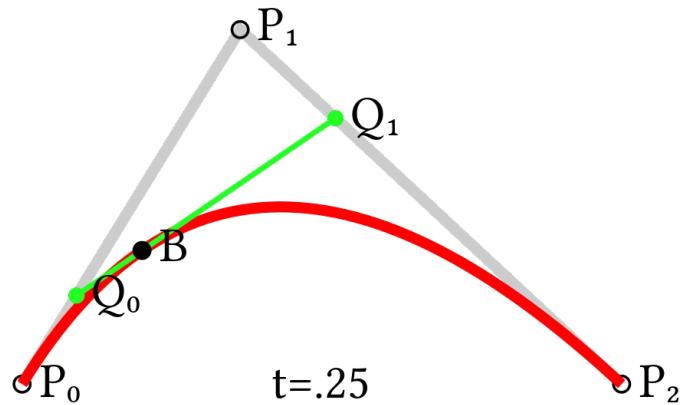


Figura 3.5: Esquema de una curva Bézier cuadrática de Bézier curve — Wikipedia [8].

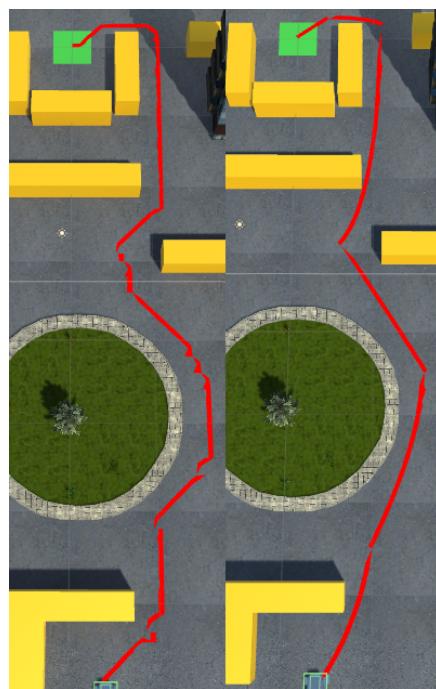


Figura 3.6: Comparación entre la ruta obtenida con el A* y con las curvas Bézier.

3.3. Theta*

Theta* es un algoritmo desarrollado por A. Nash, K. Daniel, S. Koenig y A. Felner [9, 10] en 2010 que modifica el A* original para obtener rutas más realistas.

Como explicamos anteriormente, el A* por la forma en la que representa el espacio de búsqueda, da resultados poco realistas, con lo que hay que realizar un postprocesado de la ruta obtenida. El Theta* incorpora este proceso al propio A* para obtener de forma directa rutas que no requieran del proceso de suavizado o que lo minimice.

Para ello, el cambio que hace el Theta* respecto al A* es que un sucesor a una casilla puede ser cualquier casilla, en vez de ser únicamente las que tiene alrededor. El algoritmo es el mismo que el A* original, pero cuando se calculan los sucesores, además de calcular si es un sucesor válido y el coste de ese sucesor, comprueba si hay línea de visión con la casilla del nodo padre del nodo que lo generó. Si no hay línea de visión, sigue el algoritmo del A* y le asigna al sucesor como padre la casilla que lo originó. Pero si hay línea de visión, le asigna como padre no el nodo que lo generó, sino el padre de este, calculando el coste del sucesor teniendo en cuenta esta otra casilla. De esta forma elimina las casillas intermedias que origina el A* que no son necesarias de forma similar a como se realizaría en el postprocesado para eliminar el zigzag.

Podemos observar el funcionamiento del algoritmo en la figura 3.7. La línea discontinua roja muestra el camino encontrado por el A*. En cambio, el Theta*, al haber una línea de visión entre el nodo S_{start} y el nodo S' , le asigna como padre al nodo S' el nodo S_{start} sin pasar por el nodo intermedio S como hace el A*.

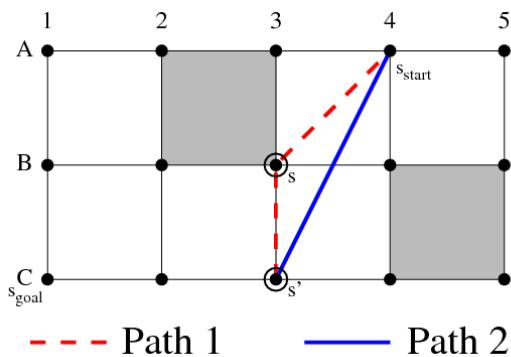


Figura 3.7: Comparación de los caminos elegidos por el A* (path1) y el Theta* (path2) de [11].

3.4. A* con vértices

El algoritmo A* como hemos visto, divide el espacio de búsqueda en casillas de un espacio discreto. Esto presenta los problemas de rutas poco realistas, y además al ser discreto, no podemos representar una línea recta entre dos puntos distantes, si no que debemos recorrer todas las casillas intermedias.

Podemos mejorar ambos problemas usando un *Nav Mesh*¹. Un *Nav Mesh*¹ usa polígonos para representar el espacio de búsqueda de tal forma que los lados de estos polígonos y sus vértices se encuentran en los bordes del espacio recorrible, que es el espacio del mapa sin obstáculos por donde puede moverse en nuestro caso el vehículo. De esta forma, podemos usar esos vértices como las casillas del espacio discreto, lo que produce tres mejoras importantes:

1. La representación del espacio recorrible se ajusta más a un espacio continuo al poder ir de un vértice a otro sin pasar por estados intermedios.
2. Permite que las rutas obtenidas sean más realistas que las obtenidas con la representación discreta.
3. Al ser un espacio continuo dentro de los polígonos del *Nav Mesh*, evitamos pasar por todos los estados intermedios de una representación discreta, reduciendo enormemente las casillas a explorar y reduciendo el tiempo de ejecución del algoritmo.

Este algoritmo es una variante del A*, donde la función de sucesores cambia de los espacios contiguos a la casilla que se va a explorar, a los vértices que son visibles desde la la casilla actual. Las casillas ahora no son las contiguas si no que se encuentran en la posición de los vértices.

En la figura 3.8 podemos ver una comparación entre los sucesores generados por el A*, a la izquierda, y el A* usando los vértices, a la derecha. Las casillas rojas representan un obstáculo, la amarilla la casilla que se está explorando, y las casillas azules representan a los sucesores. Mientras que en el A*, los sucesores son las casillas contiguas, usando los vértices del *Nav Mesh* evitamos las posiciones intermedias y obtenemos como sucesores las casillas visibles más alejadas desde la casilla que se está explorando.

Podemos ver una comparativa con los resultados y tiempos del A* original en la sección Comparativa de los algoritmos 5.5.

¹*Nav Mesh* es un conjunto de polígonos que se usan en los motores gráficos 3D para representar la zona recorrible de un agente

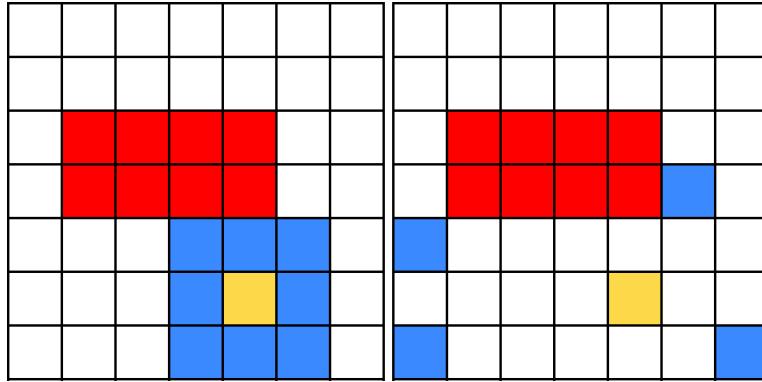


Figura 3.8: Comparación entre los sucesores del A* y del A* usando los vértices. Las casilla rojas representan obstáculos, las amarillas la casilla que se está explorando y las casillas azules los sucesores.

3.5. *Hybrid A**

Los algoritmos que hemos visto anteriormente, tanto los que usan una representación totalmente discreta, hasta los que buscan aproximarse a rutas más realistas en un espacio continuo, tienen el problema de que no tienen en cuenta las características del vehículo que va a realizar la ruta.

El *Hybrid A**^[12, 13] se basa en el A* tradicional, pero teniendo en cuenta estos dos factores: que el vehículo se mueve por un espacio continuo, y que lo hace siguiendo sus limitaciones físicas.

Las limitaciones físicas definen dos tipos de vehículos, los holonómicos y los no holonómicos. Un vehículo es holonómico si es capaz de desplazarse en el mismo número de grados de libertad que el espacio donde se mueve. Por ejemplo, un coche se desplaza en un espacio con tres grados de libertad, dos para la posición y uno para la orientación. Pero en cambio sólo puede desplazarse a través de dos grados de libertad, uno de posición y otro de orientación, lo que le impide poder realizar movimientos laterales. Es por tanto un vehículo no holonómico. Para que fuese holonómico y no tener esa limitación, tendría que poder desplazarse en ese otro grado de libertad usando por ejemplo ruedas omnidireccionales.

El *Hybrid A**, utiliza una aproximación continua al A*. En nuestro caso, la aproximación usada ha sido el modelo de la bicicleta. Este modelo es válido porque los ejes de un vehículo son simétricos, por lo que el resultado de un lado del vehículo será equivalente al del otro lado, pudiéndolo simplificar como si fuera un solo eje formado por la rueda delantera y la rueda trasera.

Los elementos del modelo de la bicicleta son:

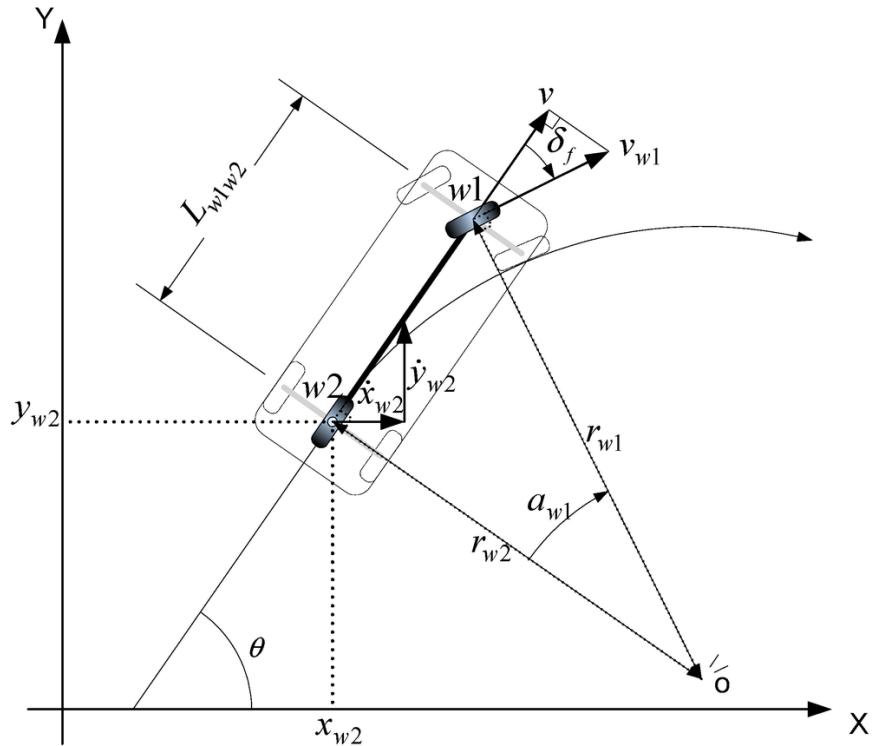


Figura 3.9: Esquema del modelo de la bicicleta de [14].

1. La rotación del vehículo θ .
2. El ángulo de giro de las ruedas δ
3. La distancia entre los ejes L
4. La distancia recorrida por el vehículo d

Con estos elementos podemos calcular en qué posición se encontrará el vehículo en un espacio de tiempo, teniendo en cuenta su capacidad de maniobra.

Las fórmulas para el cálculo de la nueva posición del vehículo son:

$$\begin{aligned}x' &= x + v\Delta t \sin \theta \\z' &= z + v\Delta t \cos \theta \\\theta' &= \theta + \Delta t \omega\end{aligned}$$

Para $v\Delta t$ hemos realizado una aproximación, tal que hemos considerado que la velocidad del vehículo v va a ser constante, y que Δt es el tiempo que

tarda en recorrer una unidad de nuestra representación. Esto es válido en nuestro caso y nos permite simplificar las ecuaciones porque la velocidad de nuestro vehículo a través del *PID Controller* (ver sección 3.6) va a ser aproximadamente constante. En un entorno real v sería la velocidad del vehículo entre el lapso de tiempo recorrido entre las mediciones representado por Δt

Para calcular ω , que es el radio de giro que realiza el vehículo en ese espacio de tiempo continuo, usamos la siguiente fórmula ²:

$$\omega = \frac{d}{L} * \tan \delta$$

Con estos elementos ya podemos calcular cual es el siguiente estado del vehículo en un espacio continuo. Los sucesores del estado actual serán esos estados continuos, variando el ángulo de giro, y podemos tener en cuenta el sentido permitiendo al vehículo ir marcha atrás añadiendo los sucesores correspondientes.

Esto nos permite además de obtener rutas más realistas, rutas que se adecuan a la maniobrabilidad del vehículo, rutas que permiten maniobrar y cambiar de sentido si fuera necesario.

Una de las desventajas del *Hybrid A** es que no es un algoritmo completo, es decir que aunque exista una solución no siempre dará con ella, al contrario que el A*. Cuando calculamos los sucesores de un estado se asigna el mejor a una casilla discreta correspondiente del espacio discreto. Además, de forma tradicional el *Hybrid A** si vuelve a una casilla ya explorada, descarta ese camino. De esta forma, aunque en la mayoría de los casos encontrará un camino, no siempre lo hará aunque exista.

Definición de los estados

En el A* cada estado tenía asociado solamente la posición y su coste. En cambio en el *Hybrid A** necesitamos más parámetros para poder definir cada estado.

Los parámetros necesarios para cada estado el *Hybrid A** son:

- **Posición continua:** es la posición a donde se puede mover el vehículo teniendo en cuenta sus características y limitaciones.
- **Posición discreta:** es la casilla del mapa representado de forma discreta que le corresponde a la posición continua.

²Norvig y Thrun «Intro to Artificial Intelligence» (<https://www.udacity.com/course/intro-to-artificial-intelligence--cs271>).

- **Orientación:** es el ángulo que forma el vehículo con el eje de referencia después de haber alcanzado la posición continua correspondiente.
- **Sentido:** indica si el vehículo se tienen que mover hacia adelante o hacia atrás para alcanzar la posición.
- **Coste:** es la suma del coste del camino hasta la posición continua más la estimación del camino hasta la meta.

Función de sucesores

Cuando con el A* calculamos los sucesores, cada uno de ellos corresponde a una posición discreta que a su vez corresponde a una casilla. Pero cuando calculamos los sucesores de forma continua para el *Hybrid A** esto no ocurre.

A través de las fórmulas 3.5 para calcular la posición continua, podemos calcular tantos sucesores como se crea conveniente (mediante un parámetro del método). Tendremos un sucesor en línea recta hacia delante, y un sucesor en línea recta hacia atrás. Pero para los giros tendremos tantos como deseemos desde 0° hasta el ángulo máximo de giro de las ruedas del vehículo. Aunque cuantos más sucesores se decida calcular el coste computacional de la exploración será mucho mayor. Por ello, para este proyecto, se calculan seis sucesores correspondientes a los dos mencionados anteriormente, que van en línea recta adelante y atrás, y a los lados con el ángulo máximo de giro de las ruedas.

A continuación hay que asignar esos sucesores a una casilla discreta. Para ello tomamos el valor entero de su vector de posición, pero al hacer esto es posible que varios sucesores coincidan en la misma casilla discreta. En ese caso se elige aquel sucesor cuyo coste sea menor, descartando los otros. Es posible guardar todos los sucesores, el problema es el mismo que cuando se decide cuantos sucesores generar, el coste computacional sería muy alto debido a que el número de sucesores que se generaría con las posiciones continuas sería mucho mayor que su equivalente en una representación discreta. Descartando los peores sucesores continuos de cada posición discreta, contenemos el coste computacional.

Cálculo del coste de la ruta

El cálculo del coste a través de las funciones G() y H(), es la forma que tenemos de controlar como se va a realizar la búsqueda de la ruta, así como también que rutas se van a seleccionar con respecto a otras posibles.

Para el *Hybrid A** hemos ampliado la función G() para que además de la distancia euclídea entre un estado actual y su estado sucesor tenga en cuenta que::

1. Si el vehículo gira es un coste adicional.

2. Si el vehículo retrocede es un gran coste adicional
3. Si pasa cerca de un obstáculo es un coste adicional

De esta manera se pretende que se seleccionen las rutas más realistas posibles. Por ejemplo, si no se penalizará el ir hacia atrás, podrían resultar casos donde el *Hybrid A** seleccionase una ruta donde fuese desde el inicio hasta la meta de forma continua marcha atrás, aunque pudiese girar e ir hacia adelante. Si ir marcha atrás es igual de válido que ir hacia adelante no habría motivo para dar la vuelta, pero ese resultado es poco realista debido a que en la realidad es peligroso y poco deseable ir marcha atrás más tiempo del necesario.

Al penalizar los giros ocurre un caso parecido puesto que en la realidad se prefiere ir en línea recta. Aunque, en este caso surge un conflicto con el tercer caso de mantener una distancia con los obstáculos. Es decir, a través de los pesos de la heurística el algoritmo debe decidir si es mejor girar y alejarse de un obstáculo o si es preferible mantenerse cerca pero ir en línea recta.

La fórmula utilizada para la función G() ha sido:

$$distanciaRecorrida = distanciaDesdeElPadre + distanciaPadre$$

$$pObstaculos = (100 - (distanciaObstaculo/10))/100$$

$$coste = (distanciaRecorrida * pgiro * patras) * pObstaculos$$

Siendo *pgiro* la penalización que se le aplica por realizar un giro, *patras* la penalización que le aplica por ir marcha atrás y *pObstaculo* la penalización por pasar cerca de un obstáculo.

Mapa de distancias

Cuando los algoritmos que hemos visto buscan la mejor ruta, a veces tienen el problema que el resultado que obtienen es una ruta que pasa demasiado próxima a los obstáculos. Puede ser debido a varios factores, como que la ruta más corta sea esa o que al realizar las heurísticas se haya preferido penalizar los giros para maximizar los tramos en que el vehículo sigue una línea recta evitando girar para alejarse de los obstáculos.

Pero normalmente no es conveniente que la ruta pase demasiado cerca de los obstáculos si no es necesario, para evitar que se produzcan choques, por ejemplo, al cambiar de dirección.

Para intentar evitar que se acerque demasiado, hemos utilizado dos métodos. Primero hemos dado un radio al *Nav Mesh* de forma que no haya posiciones válidas demasiado cerca de los obstáculos. Y segundo, hemos creado un

mapa de distancias que almacena la distancia de cada posición discreta al obstáculo más cercano. Al incorporar esta distancia a la función $G()$, el algoritmo prefiere posiciones que estén alejadas de los obstáculos.

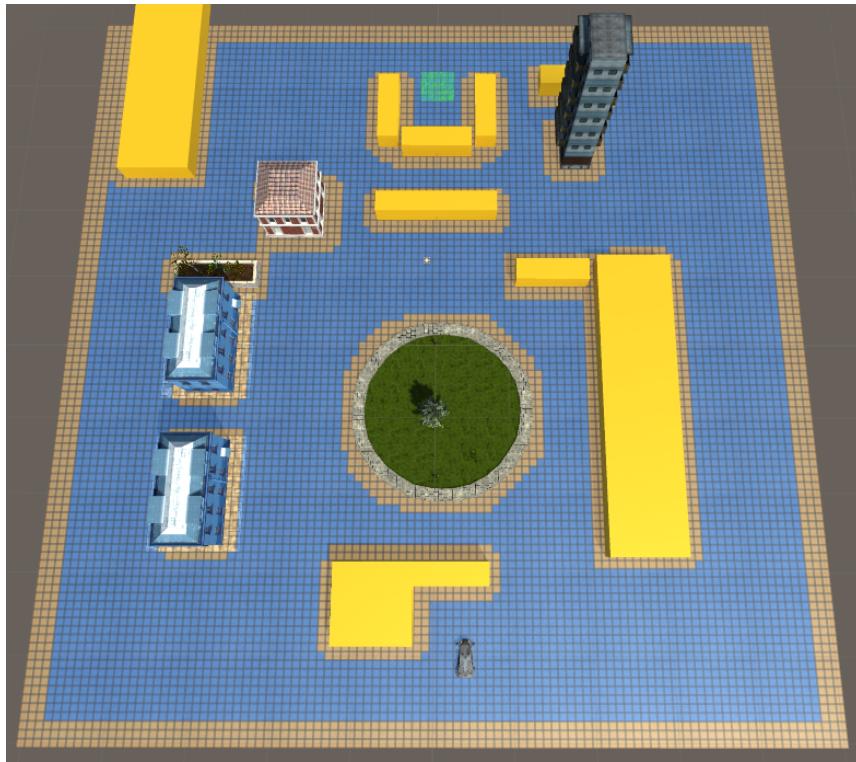


Figura 3.10: Mapa de obstáculos: las casillas azules representan espacios libres, y las amarillas a los obstáculos una vez que se añadido una pequeña distancia de seguridad.

Para crear el mapa de distancias, primero hemos creado un mapa de los obstáculos. Para ello hemos usado el *Nav Mesh* para generar una matriz con las casillas libre y las ocupadas. Iterando a través de todas las posiciones discretas hemos comprobado con el *Nav Mesh* si correspondían a una casilla recorrible o no.

A partir del mapa de obstáculos hemos creado el mapa de distancias. Hemos utilizado el algoritmo *BrushFire* o *GrassFire*. Este algoritmo parte desde una posición inicial a la que asigna un valor, generalmente cero, y asigna o actualiza a todas las posiciones adyacentes a su valor más uno. A continuación realiza el mismo proceso con esas posiciones adyacentes. De esta manera, partiendo desde las casillas de los obstáculos, podemos actualizar la distancia a la que se encuentran de ese obstáculo y crear el mapa de las distancias.

Primero se inicializa con un valor máximo a todas las casillas. Una vez

Inicializado el mapa de distancias, partiendo de los obstáculos a los que se les asigna el valor cero, se recorre las casillas adyacentes que se actualizarán su distancia al valor de la casilla previa más uno. Si una casilla ya ha sido actualizada previamente, se le asignará el valor menor que corresponde a la distancia al obstáculo más cercano.

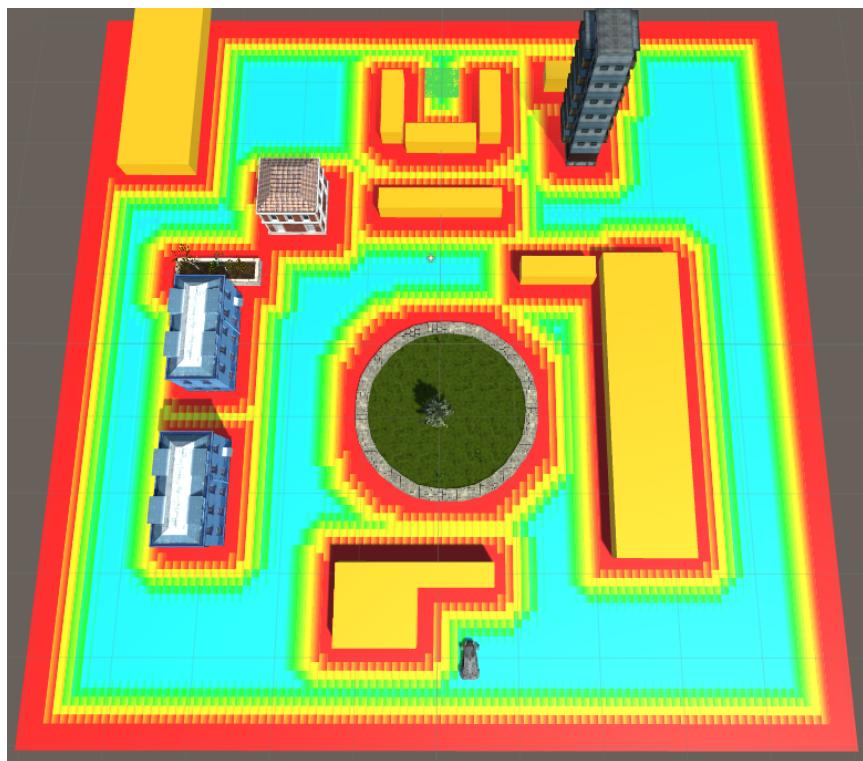


Figura 3.11: Mapa de distancias: los colores hacia el rojo representan cercanía a un obstáculo, y los colores cercanos al azul y verde casillas alejadas.

Pseudocódigo *BrushFire* o *GrassFire*

```

abiertos = ∅;
forall las casillas del mapa de distancias do
| casilla.distancia = valorMáximo;
end

forall los obstáculos en el mapa de obstáculos do
| obstáculos.distancia = 0;
| mapaDistancias.actualizar(obstáculo, obstáculo.distancia);
| abiertos.añadir(obstáculo);

end

while abiertos no esté vacío do
| actual = abiertos.primero();
| sucesores = obtenerSucesores(actual);
forall los sucesores de actual do
| | if sucesor no es obstáculo then
| | | sucesor.distancia = actual.distancia + 1;
| | | anterior = mapaDistancias(sucesor);
| | | if anterior.distancia > sucesor.distancia then
| | | | mapaDistancias.actualizar(sucesor, sucesor.distancia);
| | | | abiertos.añadir(sucesor);
| | | end
| | end
| end

end

```

Algorithm 2: Pseudocódigo del *BrushFire* o *GrassFire***Heurística *Hybrid A****

Una estrategia común para obtener heurísticas es relajar las condiciones que determinan la distancia en el mundo real. Para la realización de la función heurística podemos considerar tres posibilidades principalmente:

1. La distancia euclídea en línea recta desde la posición actual hasta la meta.

2. La distancia hasta la meta teniendo en cuenta los obstáculos que se encuentren pero sin tener en cuenta las restricciones de movimiento.
3. La distancia hasta la meta sin tener en cuenta los obstáculos pero teniendo en cuenta las restricciones de movimiento.

Todos estos casos son una simplificación del caso real, debido a que la función heurística es una estimación del coste y como vimos en la heurística del A* [3.1](#), para que el algoritmo sea admisible no se debe sobreestimar el coste hasta la meta.

Para este proyecto hemos usado la heurística del segundo caso, y hemos precalculado el coste desde todos los posibles estados hasta la meta teniendo en cuenta los obstáculos que se encuentran en el mapa. Para ello, hemos usado el mapa de obstáculos [3.5](#) que generamos previamente, y usando el algoritmo *Brushfire o Grassfire* [3.5](#) desde la meta, hemos guardado un mapa con las distancias de cada estado hasta la meta dando a las casillas de obstáculos un valor máximo.

De esta forma conseguimos mejorar el coste computacional del *Hybrid A** puesto que la estimación del coste del camino hasta la meta, se acerca más al coste real aún no teniendo en cuenta las limitaciones del movimiento, consiguiendo reducir el número de estados a explorar.

3.6. *PID Controller*

Para mover el vehículo vamos a usar un *PID Controller* (Controlador Proporcional, Integral y Diferencial). Un controlador de este tipo itera y se retroalimenta para corregir el error existente entre el valor actual y el resultado que se quiere conseguir.

Este algoritmo se usa para conseguir que el vehículo siga la ruta planificada lo más fielmente posible y es posible usarlo con cualquiera de los algoritmos anteriores. En nuestro caso, comprueba en cada momento el error que existe entre la posición del vehículo y la ruta obtenida con los algoritmos para obtener el giro del volante necesario.

La parte **proporcional** considera la diferencia entre el valor actual y el valor objetivo, y aplica la corrección (la fuerza que debe hacer un motor, o el ángulo de giro de las ruedas) de forma proporcional a ese error. El problema con la parte **proporcional** es que puede excederse. Es decir, al acercarse al valor correcto utilizar una corrección excesiva que haga que sobrepase el valor correcto, y tenga que corregirse en sentido contrario a continuación. El efecto de solo considerar la parte **proporcional** sería un vehículo balanceándose de un lado al otro de la ruta que intenta seguir.

La parte **diferencial** sirve para corregir el exceso que se puede producir en la parte **proporcional**. No tiene en cuenta el error si no su variación en el tiempo e intenta que esta variación sea continua en el tiempo. De esta forma, cuando se acerca al valor deseado suaviza la corrección que produce la parte **proporcional**, y si nos excedemos evita que se realice una corrección brusca en el otro sentido. Así evita que el controlador balancee alrededor del valor objetivo y se acerque de una forma estable.

La parte **integral** se encarga de corregir el error que se produce por causas externas. Por ejemplo, si en un vehículo se produce siempre un sobregiro hacia un lado por una avería o alguna otra causa, la parte **integral** se encarga de añadir la corrección necesaria para compensarlo. Para ello no considera sólo el error, si no como se ha mantenido a lo largo del tiempo. Si un error persiste e impide acercarnos al objetivo, la parte **integral** intentará compensarlo. Por ejemplo, en el caso del vehículo con un problema de sobregiro, aplicará una corrección a la parte **proporcional** para compensar el sobregiro y que no gire en exceso las ruedas.

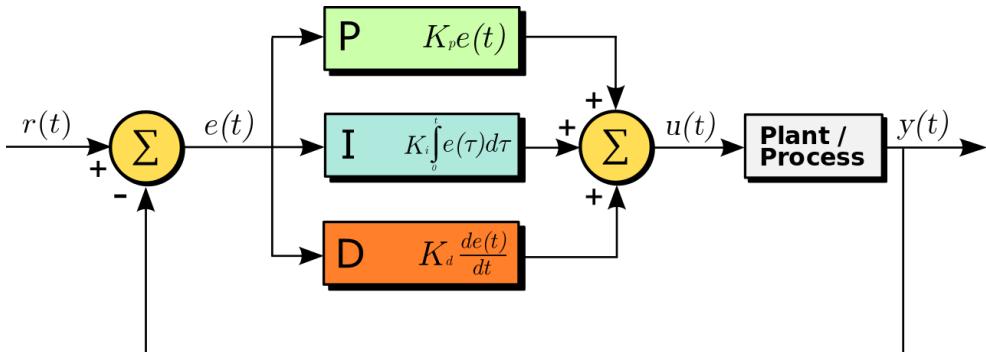


Figura 3.12: Esquema de un *PID controller* de [15].

Cada parte del controlador recibe un parámetro que controla cuánto afecta al resultado. Normalmente queremos que la mayor parte del resultado venga establecido por la parte **proporcional**, dando valores bajos tanto a la parte **diferencial** como a la **integral**. Si se diese un peso alto a la parte **diferencial** podría producir el efecto contrario al que pretende evitar y excederse. Si se diese un peso alto a la parte **integral**, podría producir un estado inestable debido al error acumulado a lo largo del tiempo.

Las fórmulas de un PID Controller son las siguientes:

$$P = paramP * (valorObjetivo(t) - valorActual(t))$$

$$I = paramI * \left(\int_0^t errorAcumulado(t) dt \right)$$

$$D = paramD * \left(\frac{errorAnterior(t) - errorActual(t)}{dt} \right)$$

Y el valor final que se usará para corregir el valor actual y acercarnos al valor objetivo lo obtenemos sumando los tres resultados:

$$error = P + I + D$$

Técnicas y herramientas

4.1. Programas usados

Unity

*Unity*³ un motor para juegos usado principalmente para desarrollar videojuegos y simulaciones. Que sea un motor para juegos quiere decir que está formado por varios motores a su vez, como el motor gráfico y el motor de físicas.

Unity proporciona un editor donde se puede crear escenas junto con un gran número de herramientas, ejemplos y modelos para crear escenarios tanto 2D y principalmente 3D.

Tiene una licencia de uso «por pasos» de ingresos obtenidos por el contenido creado con él. Existen distintas versiones de la licencia según el uso que se vaya hacer, así como de la cantidad de ingresos que se generen con su uso. En la versión Personal, se puede usar libremente siempre que los ingresos generados por el contenido usado no supere los 100 mil dólares anuales, y es la versión de la licencia usada. Las siguientes versiones de la licencia requieren un pago de una cuota anual por cada usuario y el límite de ingresos por el uso comercial del contenido creado va aumentando.

Para el proyecto probamos de forma descendiente las distintas versiones de *Unity* para Linux en un xUbuntu 16.04, debido a que las últimas versiones producían errores que cerraban el editor al iniciar o al poco tiempo después de abrirse. Creemos que esto se debía a un problema de compatibilidad entre los *drivers* de la tarjeta gráfica que no tenían soporte para algunas de las nuevas características de *Unity*. La primera versión estable que funcionaba en el equipo de desarrollo fue la versión 5.3.6f1, y ha sido la usada para su realización.

³Unity 3D <https://unity3d.com/>

Texmaker

Para la realización de la memoria se ha usado latex con el editor Texmaker⁴. Texmaker es un editor libre con licencia GPL.

Texmaker incluye varias herramientas como el corrector ortográfico o el visor de pdf que facilitan la creación de los documentos.

La versión usada ha sido la 4.4.1.

SmartGit

SmartGit⁵ es un gestor gráfico para git que soporta github y que facilita la realización de *commits*, *push* y *pull* de los repositorios del proyecto.

Tiene una licencia no comercial que permite el uso de forma gratuita a desarrolladores de código abierto, profesores, estudiantes, desarrollos no comerciales y también organizaciones sin ánimo de lucro.

La versión que hemos usado ha sido la 17.0.3

GitHub y ZenHub

La plataforma elegida para los repositorios ha sido GitHub, principalmente por su compatibilidad con Zenhub⁶ que es la herramienta de gestión de proyectos que hemos usado.

Zenhub es una herramienta que permite la planificación y control del proyecto. Es un complemento de Firefox que se integra con github y añade funciones como las *boards* para manejar el control de las *issues* y las *milestones* del proyecto, y que permite ver los gráficos del progreso y *burnouts* que se ha realizado.

La versión utilizada ha sido la 2.34.2

Remarkable

Remarkable⁷ es un editor del lenguaje *markdown* usado para crear archivos de texto plano que puedan ser convertidos a HTML, PDF o otros formatos.

Lo hemos usado para crear y editar el archivo readme.md del repositorio de GitHub.

El programa tiene una licencia libre de código abierto que permite usarlo sin limitaciones, incluido fines comerciales.

La versión utilizada ha sido la 1.87.

⁴Texmaker <http://www.xm1math.net/texmaker/>

⁵SmartGit <http://www.syntevo.com/smartgit/>

⁶Zenhub <https://www.zenhub.com/>

⁷Remarkable <https://remarkableapp.github.io/>

Aspectos relevantes del desarrollo del proyecto

5.1. *Navigation Mesh*

Una *Navigation Mesh* o *Nav Mesh* es un conjunto de polígonos, triángulos normalmente, que se usan en los motores gráficos 3D para representar la zona o camino recorrible de un agente. Se crean a partir de los objetos estáticos de una escena, que son los obstáculos, donde no se puede desplazar. El resto de la zona de la escena será zona recorrible por donde nos podemos desplazar.

En relación por ejemplo al A*, sería el equivalente a la matriz que representa el mapa donde se buscará los nodos a explorar.

Está formada por los vértices de los polígonos, y si un punto está dentro del área que forman, entonces ese punto es recorrible. Esto es más realista en un escenario en tres dimensiones, como los representados por los motores 3D, debido a que permite un movimiento menos discretizado, al contrario que las cuadrículas de casillas que se adecuan más a una representación en dos dimensiones. Y si consideramos los vértices los nodos sucesores, resulta más óptimo computacionalmente, ya que el número de nodos (vértices) a explorar es menor, debido a que no tenemos que tener en cuenta todos los puntos intermedios hasta llegar al vértice, si no únicamente si el vértice es visible o alcanzable desde el nodo actual.

Hemos usado una *Nav Mesh* para generar de forma automática el mapa para el A*. En *Unity* esto se puede hacer de dos formas. En versiones anteriores a la versión 5.6 solo es posible generarlo desde el editor antes del tiempo de ejecución. A partir de la versión 5.6, que permite acceder a las herramientas del editor en tiempo de ejecución, se puede construir a través de *NavMeshBuilder*, dando mucha más libertad a la hora de crear mapas de forma dinámica o interactiva. Lamentablemente, no se pudo usar la versión 5.6 y la versión más

actual que funcionaba en el equipo de desarrollo fue la 5.3.6.



Figura 5.13: Imagen de un *Nav Mesh* en *Unity*.

La clase *NavMesh* de *Unity* tiene métodos que podemos usar como *RayTrace*, que lanza un rayo desde un vector a otro y devuelve si son visibles dentro de la *NavMesh*, es decir, si existe una línea recta entre ellos que los una dentro del camino recorrible. Este método lo hemos usado para decidir si un nodo sucesor es válido o no. Al usarlo encontramos un *bug* que producía que a veces no se encontrará el camino o bucles infinitos al recorrer las listas. Esto es debido a que puede ocurrir que no devuelva lo mismo dependiendo del orden o sentido en el que se le indiquen los vectores entre los que lanzar el rayo. Por ejemplo *RayTrace(vector1, vector2)* puede ser visible pero *RayTrace(vector2, vector1)* no. Para solucionarlo, tuvimos que vigilar el orden con el que recorríamos las listas, y decidimos, que cuando fuera posible, comprobar que para que fuera un sucesor válido debe ser visible en los dos sentidos.

5.2. Cola de prioridad

Uno de los problemas que se encontraron al usar C# y en concreto la versión que utiliza *Unity* ha sido el no poder usar todos las estructuras de datos de

C# y que algunas estructuras de datos no estaban disponibles directamente en C#.

Una de estas estructuras de datos que no se encuentran en C# es la cola de prioridad. La cola de prioridad es utilizada por el A* para almacenar los estados que se van a explorar. Se ordenan según su coste y se extrae de la cola aquél que tenga menos coste para ser el siguiente en ser explorado.

Para tener acceso a una cola de prioridad hemos usado *High Speed Priority Queue for C#*⁸[16].

Los motivos para utilizarla han sido que tiene una licencia MIT que permite usarla libremente. Es sencilla y fácil de usar, además está creada con el objetivo de ser usada para *Path Finding* con lo que encaja en los objetivos del proyecto.

5.3. Física del vehículo

Unity dispone de un motor de físicas integrado basado en el *PhysX* de *Nvidia*⁹. Una de las características que se encuentran en el motor de físicas y que han resultado de gran utilidad para la realización del proyecto han sido los *Wheel Colliders*

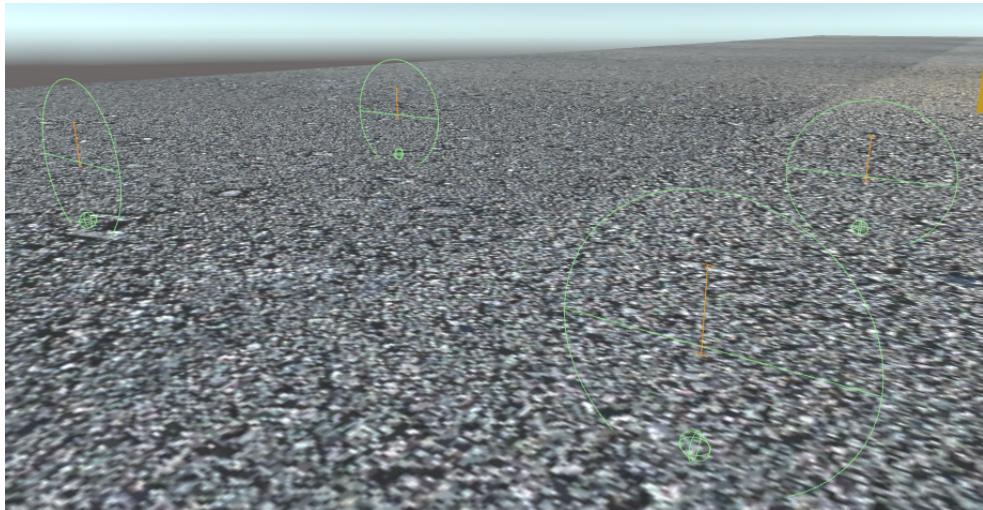


Figura 5.14: Representación de los cuatro *Wheel Colliders* en *Unity*.

Los *Wheel Colliders* son una simulación realista de las ruedas de un vehículo y permiten simular todas las físicas necesarias para el vehículo. Tienen tanto simulación del giro de las ruedas como de la fuerza que aplica el motor a las

⁸<https://github.com/BlueRaja/High-Speed-Priority-Queue-for-C-Sharp>

⁹<https://blogs.unity3d.com/es/2014/07/08/high-performance-physics-in-unity-5/>

mismas, y una simulación del freno, además de otras simulaciones como la amortiguación que no se han controlado de forma directa en este proyecto.

El uso de los *Wheel Colliders* nos ha permitido la simulación realista de un vehículo con tracción a las cuatro ruedas y un radio de giro de las mismas de 45 grados. Su uso es independiente del modelo del vehículo usado, pudiendo adaptarlo a cualquier modelo cambiando sus parámetros. Tampoco hay limitaciones al número de *Wheel Colliders* que podemos usar, de tal forma que es posible simular una motocicleta, un camión o cualquier otro vehículo que use ruedas.

5.4. Modelos utilizados

Los modelos usados han sido los que estaban disponibles en la *Assets Store*¹⁰ de *Unity*.

Principalmente hemos usado los *Standard Assets 1.1.2*¹¹ de *Unity* de donde hemos utilizado el modelo del coche, y cuyos ejemplos nos han servido para aprender como usar *Unity*.

El siguiente conjunto de *assets* que más hemos utilizado ha sido *Low Poly Street Pack 1.0*¹² de *Dynamic Art*, de donde hemos usado la mayor parte de los edificios y elementos del escenario.

En menor medida también se han utilizado elementos de *Abandoned Building 1.0*¹³ de Aleksey Kozhemyakim, *Block Building Pack 1.0*¹⁴ de CGY (Yemelyan K.), *Building Apartment 1.0*¹⁵ de LowlyPoly, *San Francisco House 1.0*¹⁶ de Bretwalda Games, *UK Terraced House Pack 1.0*¹⁷ de rik4000.

Los *assets* usados son gratuitos y siguen la licencia de la *Unity Store*¹⁸ en el que su uso es libre dentro de juegos y de medios interactivos.

5.5. Comparativa de los algoritmos

En esta sección vamos a comparar los algoritmos dependiendo de si utilizan una representación del espacio de búsqueda continua o discreta. A continuación también compararemos el rendimiento de los distintos algoritmos tomando como referencia al A*.

¹⁰<https://www.assetstore.unity3d.com/>

¹¹<https://www.assetstore.unity3d.com/en/#!/content/32351>

¹²<https://www.assetstore.unity3d.com/en/#!/content/67475>

¹³<https://www.assetstore.unity3d.com/en/#!/content/62875>

¹⁴<https://www.assetstore.unity3d.com/en/#!/content/13925>

¹⁵<https://www.assetstore.unity3d.com/en/#!/content/80004>

¹⁶<https://www.assetstore.unity3d.com/en/#!/content/16640>

¹⁷<https://www.assetstore.unity3d.com/en/#!/content/63481>

¹⁸https://unity3d.com/es/legal/as_terms

Algoritmo	Discreto	Continuo	Modelo Físico	Varios Sentidos
A*	X			
Theta*	X			
A* vértices		X		
Hybrid A*		X	X	X

Tabla 5.1: Comparación de los algoritmos según su representación

Algoritmo	Segundos	Porcentaje
A*	6.6	100 %
Theta*	10.2	155.5 %
A* vértices	2.05	31.1 %
Hybrid A*	4.01	60.8 %

Tabla 5.2: Comparación de los algoritmos según su rendimiento

En la tabla 5.1 comparamos los algoritmos teniendo en cuenta como representan el espacio de búsqueda y que limitaciones tienen en cuenta. Podemos observar que el *Hybrid A** es el único que tiene en cuenta el modelo físico de los vehículos, y con ello el único que busca rutas válidas para la mayoría de los casos con vehículos no holonómicos. Además, debido a esto, es el único que puede buscar rutas que indiquen al vehículo a moverse en varios sentidos según su dirección. El resto de algoritmos realizan la búsqueda para vehículos holonómicos no teniendo en cuenta estas restricciones.

En la tabla 5.2 comparamos el rendimiento de los algoritmos usados en las mismas condiciones y buscando una ruta con los mismo inicio y meta. Hemos tomado el A* como referencia ya que el resto se basan en él. Los resultados son los esperados. El Theta* es más lento debido a los cálculos extra que debe hacer para comprobar si puede asignar un nodo anterior al nodo sucesor. La versión del A* con vértices es mucho más rápida debido a que el número de estados a explorar es mucho menor, al usar los vértices del *Nav Mesh*. El *Hybrid A**, aunque es el que más cálculos debe llevar a cabo, es más rápido debido a la heurística mejorada 3.5 que le hemos incluido, donde tiene precalculado el coste desde todos los estados posibles hasta la meta teniendo en cuenta los obstáculos.

Trabajos relacionados

Los trabajos relacionados que se han estudiado han sido:

- Path Planning and path following for an autonomous car de Matthew Bradley [17].

Es un trabajo donde se implementa el *Hybrid A** en un espacio 3D creado con XNA y Farseer Physics. Implementa el *Hybrid A** con distintos algoritmos para mejorar su búsqueda y rendimiento pero no incluye otros algoritmos de planificación de rutas.

- Estudio sobre algoritmos de inteligencia artificial aplicables a drones de Alvin Vanni [18].

Este trabajo implementa algoritmos para el vuelo autónomo de quadcopters. Usa estos quadcopters para el análisis a través de algoritmos genéticos de las imágenes que capturan. Esta desarrollado en *Unity* y Matlab.

- Self-driving car de Erik Nordeus [19]

Este trabajo realiza el *Hybrid A** en *Unity*. Usa rutas Reed-Shepp y un *PID controller* para el movimiento del vehículo. Se puede ver la versión final implementada en un *plug-in WebGL* de *Unity* para navegadores web.

- AirSim de Microsoft [20]

Es un proyecto de código libre realizado por Microsoft que simula drones. Esta realizado en Unreal Engine e implementa el control del drone y las simulaciones de las vistas que tiene a través de sus sensores.

Conclusiones y Líneas de trabajo futuras

7.1. Conclusiones

En la realización de este proyecto hemos sido capaces de aprender como funciona y hemos usado un motor 3D, hemos implementado distintos tipo de algoritmos que permiten la planificación de rutas y hemos implementado métodos para el seguimiento de las mismas de forma autónoma. En general hemos sido capaces de alcanzar los objetivos que nos habíamos propuesto.

También hemos visto de primera mano las dificultades que entrañan los vehículos autónomos y nos hemos encontrado con la gran cantidad variables a considerar que no se aprecian en una primera fase cuando se están estudiando los algoritmos a usar.

Una de las mayores dificultades que hemos encontrado ha sido la planificación del proyecto, debido a que cuando empezamos no teníamos conocimiento de la mayoría de subtareas que eran necesarias para su realización, con lo que fue difícil calcular la cantidad de trabajo que llevaría cada una de ellas.

7.2. Líneas de trabajo futuras

Las posibles mejoras del proyecto son muchas, debido a que es un campo donde aún se está en desarrollo y que las disciplinas que abarca son múltiples.

Algunas de las posibilidades que planteamos son mejoras que se pueden introducir al trabajo ya realizado:

- Mejorar las rutas obtenidas a través de la inclusión de un mapa de Voronoi[21] que mejore la distancia a la que pasa el vehículo de los obstáculos. También se pueden mejorar a través de la inclusión de una

heurística que precalcule la distancia a la meta teniendo en cuenta los obstáculos que hay por el camino. De este manera se puede reducir el número de estados a explorar y mejorar el rendimiento.

- Mejorar el movimiento del vehículo usando rutas Reeds-Shepp[22]. Estas rutas son costosas computacionalmente pero usadas junto con el *Hybrid A** permiten mayor precisión a la hora de alcanzar la meta.
- Crear un sistema de sensores virtuales que sea capaz de detectar el mapa en tiempo real y los obstáculos según se mueve el vehículo.
- Añadir otros vehículos que se muevan por el mapa, creando sensores para el vehículo que sean capaces de detectar otros vehículos, su movimiento y evitar una colisión.
- Diseñar un sistema de reconocimiento de señales de tráfico y carreteras, de tal forma que el vehículo se mueva por el carril derecho y sea capaz de respetar las señales.

Bibliografía

- [1] A. Tovey, “Self-driving cars will add \$7 trillion a year to global economy, says intel,” 2017, [Online; accessed 8-June-2017]. [Online]. Available: <http://www.telegraph.co.uk/business/2017/06/02/self-driving-cars-will-add-7-trillion-year-global-economy-says/>
- [2] I. Lunden, “Intel buys mobileye in \$15.3b deal, moves its automotive unit to israel,” 2017, [Online; accessed 8-June-2017]. [Online]. Available: <https://techcrunch.com/2017/03/13/reports-intel-buying-mobileye-for-up-to-16b-to-expand-in-self-driving-tech/>
- [3] C. Farr, “The first study of self-driving car crash rates suggests they are safer,” 2017, [Online; accessed 8-June-2017]. [Online]. Available: <https://www.fastcompany.com/3055356/the-first-study-of-self-driving-car-crash-rates-suggests-they-are-safer/>
- [4] C. Thompson, “8 ways self-driving cars will drastically improve our lives,” 2016, [Online; accessed 8-June-2017]. [Online]. Available: <http://www.businessinsider.com/how-driverless-cars-will-change-lives-2016-12/>
- [5] H. Tulleken, “Bézier curves for your games: A tutorial,” [Online; accessed 6-June-2017]. [Online]. Available: <http://devmag.org.za/2011/04/05/bzier-curves-a-tutorial/>
- [6] Wikipedia, “Bézier curve — wikipedia, the free encyclopedia,” 2017, [Online; accessed 6-June-2017]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=B%C3%A9zier_curve&oldid=783711812
- [7] H. Tulleken, “Bézier curves for your games: A tutorial. quadratic curve,” [Online; accessed 6-June-2017]. [Online]. Available: http://devmag.org.za/blog/wp-content/uploads/2011/04/bezier_2.png

- [8] Wikipedia, “Construction of a quadratic bézier curve,” 2017, [Online; accessed 6-June-2017]. [Online]. Available: https://en.wikipedia.org/wiki/B%C3%A9zier_curve#/media/File:B%C3%A9zier_2_big.svg
- [9] K. Daniel, A. Nash, S. Koenig, and A. Felner, “Theta*: Any-angle path planning on grids,” pp. 533–579, 2010.
- [10] A. Nash, “Theta*: Any-angle path planning for smoother trajectories in continuous environments,” [Online; accessed 6-June-2017]. [Online]. Available: <http://aigamedev.com/open/tutorials/theta-star-any-angle-paths/>
- [11] ——, “Theta*: Any-angle path planning for smoother trajectories in continuous environments. figure 6 comparison theta* and a*,” [Online; accessed 6-June-2017]. [Online]. Available: <http://aigamedev.com/static/tutorials/aap-path12.png>
- [12] D. Dolgov, S. Thrun, M. Montemerlo, and J. Diebel, “Path planning for autonomous driving in unknown environments,” in *Proceedings of the Eleventh International Symposium on Experimental Robotics(I SER-08)*. Athens, Greece: Springer Tracts in Advanced Robotics (STAR), July 2008.
- [13] ——, “Practical search techniques in path planning for autonomous driving,” in *Proceedings of the First International Symposium on Search Techniques in Artificial Intelligence and Robotics (STAIR-08)*. Chicago, USA: AAAI, June 2008.
- [14] Y.-S. Byun, R.-G. Jeong, and S.-W. Kang, “Vehicle position estimation based on magnetic markers: Enhanced accuracy by compensation of time delays,” *Sensors*, vol. 15, no. 11, pp. 28 807–28 825, 2015. [Online]. Available: <http://www.mdpi.com/1424-8220/15/11/28807>
- [15] Wikipedia, “Pid controller — wikipedia, the free encyclopedia,” 2017, [Online; accessed 23-June-2017]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=PID_controller&oldid=786368579
- [16] BlueRaja, “High speed priority queue for c#,” 2016, [Online; accessed 23-June-2017]. [Online]. Available: <https://github.com/BlueRaja/High-Speed-Priority-Queue-for-C-Sharp>
- [17] M. Bradley, *Path planning and path following for an autonomous car*, 2012.
- [18] A. Vanni, *Estudio sobre algoritmos de inteligencia artificial aplicables a drones*, 2016.
- [19] E. Nordeus, “Self-driving car,” 2015, [Online; accessed 23-June-2017]. [Online]. Available: <http://www.habrador.com/p/self-driving-car/>

- [20] Microsoft, “Airsim,” 2017, [Online; accessed 23-June-2017]. [Online]. Available: <https://github.com/Microsoft/AirSim>
- [21] Wikipedia, “Voronoi diagram — wikipedia, the free encyclopedia,” 2017, [Online; accessed 26-June-2017]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Voronoi_diagram&oldid=786507557
- [22] J. Reeds and L. Shepp, “Optimal paths for a car that goes both forwards and backwards,” *Pacific journal of mathematics*, vol. 145, no. 2, pp. 367–393, 1990.