UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Planificación de trayectorias de vehículos aéreos a través de verificación de modelos

TESIS

Que para obtener el título de

Ingeniera en Computación

P R E S E N T A

Sandra Susana Pérez Gutiérrez

DIRECTOR DE TESIS

Dr. Ismael Everardo Bárcenas Patiño

Ciudad Universitaria, Cd. Mx., 2024

RESUMEN

Planificación de trayectorias en vehículos autónomos ha sido abordada en numerosas ocasiones y con diversos enfoques. Es una problemática que involucra un cambio de estado o traslado. Consiste en proporcionar una trayectoria que satisfaga una serie de requerimientos mediante un mapa (entrada del sistema) y un conjunto de restricciones (determinadas por el comportamiento deseado del sistema). Estas restricciones pueden ser: puntos, obstáculos y otros requerimientos. La planificación de trayectorias puede ser vista también como un problema de control donde se modifica la posición de un sistema haciendo uso de variables de entorno, donde los datos de entrada serán nuestras claves para contextualizar el sistema y en consecuencia generar una ruta que cumpla con las especificaciones. Planificación de trayectorias tiene varias aplicaciones en robótica, vehículos aéreos, desarrollo de aplicaciones, etc.

En este trabajo abordaremos una propuesta de solución a la planificación de trayectorias mediante verificación de modelos, la cual propone otra forma de toma de decisiones con base en la abstracción de un mapa y restricciones. Esta verificación consta de: dado un modelo abstracto y dado las propiedades requeridas, determina si las propiedades se cumplen en algún punto del modelo. El verificador de modelos comprobará si la negación de la expresión lógica se cumple o no en el modelo. Este contraejemplo representa una trayectoria dentro del mapa que satisface las restricciones.

En este proyecto se describe la implementación de un algoritmo de planificación de trayectorias para vehículos aéreos, basado en verificación de modelos. Se implementó un ambiente virtual para la visualización de la ejecución de la ruta generada por verificación de modelos; así como la prueba del algoritmo, ya que mediante el mapa dibujado en el ambiente virtual se pueden generar un sinfín de puntos a visitar por el vehículo aéreo, así como, permite visualizar su ejecución.

AGRADECIMIENTOS

Es un honor otorgar primacía de gratitud a mis padres, hermanos y familia cercana, quienes siempre me apoyaron a lo largo de mi desarrollo personal; a mis profesores y compañeros que aportaron de manera desmesurada en mi crecimiento profesional; y, finalmente, a mis amigos quienes me han acompañado a lo largo de mi vida de aprendizaje constante.

Índice general

RESUMEN I

AGRADECIMIENTOS II

CAPÍTULO 1 INTRODUCIÓN 1

1.1 Antecedentes . . . 1
1.1.1 Sistemas reactivos . . . 2
1.1.2 La ruta menos costosa . . . 4
1.1.3 Verificación de modelos . . . 5
1.2 Justificación . . . 8
1.3 Problema . . . 11
1.4 Objetivo . . . 12
1.4.1 Objetivo general . . . 12
1.4.2 Objetivos específicos . . . 12
1.5 Hipótesis . . . 12

CAPÍTULO 2 MARCO TEÓRICO 13

2.1 Planificación trayectorias . . . 13
2.2 Verificación de modelos . . . 13
2.3 Ambientes virtuales . . . 17
2.3.1 Simulador de drones en ROS . . . 18
2.3.2 Control de robots aéreos en entorno Matlab-ROS . . . 19
2.3.3 Tum_ardrone mediante ROS . . . 20
2.3.4 Niveles de fidelidad de simuladores de drones y ejemplos con Matlab . . . 21
2.3.5 Matlab y Simulink . . . 21
2.3.6 AirSim . . . 22
2.3.7 Opengl y C++ . . . 23
2.4 Implementación de Propuesta . . . 23
2.4.1 Búsqueda de modelos y diseño en Blender . . . 23
2.4.2 Implementación de puntos de interés de un drone . . . 24

2.4.3 Implementación de obstáculos en el mapa 24
2.5 Conclusiones del análisis de las diferentes herramientas a implementar para la resolución de esta problemática 27

CAPÍTULO 3 PROPUESTA DE IMPLEMENTACIÓN EN UN MAVIC PRO 29

3.1 Propuesta y validación 29
3.2 Desventajas de la propuesta 30
3.3 Ventajas de la propuesta 30

CAPÍTULO 4 DISEÑO DEL ENTORNO VIRTUAL 31

4.1 OpenGL y C++ 31
4.2 Declaración de objetos en el escenario y diseño de colliders 32
4.2.1 Bibliotecas utilizadas 32
4.2.2 Inicialización de Escenario 35
4.2.3 Animaciones 46

CAPÍTULO 5 PLANIFICACIÓN DE TRAYECTORIAS 50

5.1 Verificador de modelos 50
5.1.1 Ejemplo de uso de una estructura de Kripke en el problema de planificación de ruta 52
5.2 Implementación del código del proyecto 53
5.3 PyModelChecking y funcionamiento del código de planificación de trayectorias 54
5.4 Enlace con el entorno virtual 54
5.5 Construcción del modelo lógico en Python 55

CAPÍTULO 6 IMPLEMENTACIÓN DE PLANIFICADOR DE TRAYECTORIAS 66

6.1 Ejemplo de funcionamiento 67
6.1.1 Verificador de modelos 67
6.1.2 Ejecución de ruta 71

CONCLUSIONES Y TRABAJO A FUTURO 72

Índice de figuras

1.1 Diagrama de algoritmo de la ruta más corta de Dijkstra . Fragmento tomado de [1]. 5
1.2 Principios del verificador de modelos ejemplificado. Fragmento tomado de diapositivas de bibliografía [2]. 8
1.3 Diagrama de modelo predictivo. Fragmento tomado de [3]. 9
2.1 La implementación de esta propuesta nos provee una representación abstracta de un sistema en términos de lógica y donde la salida satisface los requerimientos planteados e implementados en términos de verificación de modelos. 15
2.2 Interfaz gáfica resultante Gazebo. 19
2.3 Tutorial e interfaz de desarrollo Gazebo. 20
2.4 Simulación de drones de baja fidelidad con el bloque Guidance Model para VANT de MATLAB. 22
2.5 Simulación de drones de alta fidelidad mediante el bloque Simulation 3D Scene Configuration. En este caso se requiere Unreal Engine para implementar los escenarios. 23
2.6 Primeras versiones del entorno virtual de diseño de ruta por model checking. 24
4.1 Resultado final del entorno gráfico. 32
4.2 Modelo de la iglesia 33
4.3 Textura de la iglesia 34
4.4 Escenario de día, tanto de shader de skybox como el de luces. 40
4.5 Escenario de noche, tanto de shader de skybox como el de luces. 41
5.1 Pseudocódigo de verificador de modelos [4] página 231. 51
5.2 Donde cada nodo representa un punto a visitar. 52
6.1 Diagrama de algoritmo implementado de verificación de modelos. 67

Índice de cuadros

5.1 Tabla de nodos etiquetados. 53

Índice de algoritmos

1 Algoritmo de Dijikstra para la ruta más corta [1]. 6

Lista de códigos

2.1 movdx,movdy y movdz son variables que toman la posición actual del vehículo aéreo. 25

2.2 En dicho código compara su posición actual con la posición de los obstáculos almacenados en el mapa y el rango de tolerancia entre ellos. Si la distancia es menor al rango de tolerancia, la función retorna un falso. 26

2.3 Aquí podemos ver que al oprimirse una tecla de movimiento manda a llamar a la función validadora y dependiendo de ella se mueve el dron en esa dirección, esto es para la manera "teleoperada". 28

4.4 Extracción de las características del monitor. 35

4.5 Modificar el tamaño y resolución de la ventana emergente del simulador. . . 36

4.6 Se inicializan los parámetros para despliegue de imagen y luz para el shader. 37

4.7 Despliegue de la ventana emergente y se inicializa el sonido. 38

4.8 Se asigna un vector para el shader perteneciente al skybox, el cual consta de una serie de imágenes cuyo número es el mismo que el número de caras que un cubo. 39

4.9 Se inicializa 3 shaders. 40

4.10 Skybox. 40

4.11 Ejemplo de dibujo de modelos. 42

4.12 Extracción de la información para dibujar skybox. 43

4.13 Bucle con el que se dibuja escenario. 43

4.14 Función animate. 44

4.15 Shader luces 45

4.16 Animaciones programadas en bucle. 47

4.17 Animaciones programadas por evento. 48

4.18 Programación de colliders. 49

5.19 Extracción de valores de discriminante. 54

5.20 Generación de matriz de adyacencia. 55

5.21 Guardar matriz de adyacencia en fichero. 56

5.22 Ejecución de código Python y extracción de ruta resultante. 57

5.23 Lectura de matriz de adyacencia. 58

5.24 Extracción de valores de matriz en variables 60

5.25 Cualificar puntos. 61
5.26 Estructura de Kripke. 61
5.27 Verificador de modelos. 62
5.28 Impresión de localización del vehículo aéreo en tiempo real 62
5.29 Iniciar vuelo. 63
5.30 Movimiento para llegar a los puntos requeridos en eje x y z 64
5.31 Movimiento para llegar a los puntos requeridos en eje y 65
5.32 Recorrido por los puntos a visitar 65

CAPÍTULO 1 INTRODUCIÓN

1.1 ANTECEDENTES

La planificación de trayectorias es una problemática común en cualquier sistema que busca un cambio de posición o estado con base en un comportamiento esperado. Los enfoques, el nivel de abstracción y el comportamiento del sistema per se son variables.

Existen variantes en las aplicaciones en las que se puede aterrizar esta problemática, pero podemos ver en ellas (como factor en común) el que se entregue como salida un conjunto de puntos resultantes que determinan el orden de nodos a visitar, es decir, se espera que entregue como resultado una ruta resultante, la cual cumpla con ciertas características planteadas desde la acotación del problema y requerimientos de solución.

Algunos ejemplos de uso de planificación de trayectorias son en aplicaciones de navegación tales como Google Maps [5] donde se usa, en específico, el algoritmo de Dijkstra. Otras aplicaciones son en robótica, como lo podría ser en cualquier problemática que involucre el diseño de una ruta mediante un análisis. Algunos ejemplos de trabajos que hacen uso de esto podrían ser: [6], donde se busca discutir y promover el uso de estos algoritmos para la resolución de diferentes problemáticas; o por ejemplo [7], se usa para la planificación de ruta de robots.

En este trabajo abordaremos la problemática de planificación de trayectorias enfocada en vehículos aéreos, donde el algoritmo arroje una propuesta de ruta personalizada según las necesidades planteadas y variables de entrada. Todo esto mediante un nivel de abstracción en 3D (haciendo uso de un entorno virtual) así como herramientas de programación que nos ayuden a abstraer y descomponer el problema entregando una solución visualizada en un simulador de vuelo de un vehículo aéreo.

Cuando desarrollamos soluciones a problemáticas del mundo real por medio de sistemas de cómputo, uno de los mayores retos es la interpretación de la naturaleza del problema a una solución que, aunque de manera acotada, cumpla (de manera total o

parcial) con los requerimientos necesarios para la resolución de la problemática; ya que un problema en el mundo real implica muchas variables de entrada con diferentes rangos en el sistema por lo que estandarizar su interpretación llega a ser complejo pero fundamental para solucionar una problemática mediante cómputo. En ese sentido, la reinterpretación de las entradas a una variable lógica (como veremos más adelante) nos da un estándar de entrada que pueda darnos la información necesaria a la vez que da pie a programar comportamientos más complejos de manera más sencilla debido al nivel de abstracción con el que se trabajan las entradas.

El aterrizar un problema del mundo real a una solución escalable (que nos permita que el sistema computacional [8] pueda manejar entradas variables sin comprometer la respuesta del mismo y adaptarse a un entorno cambiante) llega a ser muy útil y en algunos casos crucial dependiendo del tipo de sistema computacional que se esté desarrollando. Podemos observar entonces, que el análisis de la complejidad del control y las estructuras de datos que involucran la solución son fundamentales para que el sistema pueda cumplir su función y no solo dependen del diseño del sistema, sino del contexto variable en el que se encuentren las entradas y la abstracción de las mismas.

Para el análisis de este tipo de problemas, se propuso la verificación de modelos, el cual es un método de análisis de sistemas dinámicos finitos. Definiendo a los sistemas dinámicos como sistemas deterministas, los cuales, sus entradas (en un instante determinado) influyen en la respuesta del sistema. Este método está basado en principios de lógica que nos permite analizar de manera automática el problema, disminuyendo la cantidad de operaciones computacionales que se requieren para proporcionar una salida, ya que se usa variables lógicas para la discriminación de eventos, las cuales son operaciones computacionales muy sencillas de procesar. Los primeros en proponer este método fueron Edmund M. Clarke, E. Allen Emerson, y Joseph Sifakis. [4].

1.1.1. Sistemas reactivos

Podemos ver a un sistema reactivo como una caja negra que entrega salidas correspondientes al comportamiento deseado y no en función totalmente de las entradas

recibidas. Con esto podemos decir que las salidas dependen parcialmente de las variables de entrada y es más bien en función del comportamiento con el que se diseñó dicho sistema.

Entre las estrategias para desarrollar software ágil está el desarrollo de sistemas reactivos; los cuales nos permiten que un sistema pueda ser más flexible, con bajo acoplamiento, abierto a cambios, independiente y escalable. Una de sus ventajas más significativas es que son sistemas más tolerantes a fallos [9] y cuando llegan a fallar responden de manera refinada y no estrepitosa como los sistemas convencionales. Los sistemas reactivos son altamente responsivos, dando a los usuarios una retroalimentación efectiva e interactiva, de igual manera que dichos sistemas se retroalimentan constantemente de su entorno sin perder esa responsiva.

Jonas Bonér, publicó en el 2013 "Reactive Manifesto" [5]. En él propone un nuevo enfoque de desarrollo de software más ágil, es decir, propone el desarrollo de sistemas reactivos donde el sistema cumpla con ser:

- Responsivo: El sistema debe responder de manera oportuna, lo que implica que pudo detectar y visualizar la entrada, así como procesar las variables de entrada adecuadamente según los requerimientos del problema. Los sistemas responsivos se enfocan en proveer tiempos de respuesta rápidos y consistentes, estableciendo límites superiores confiables para así proporcionar una calidad de servicio consistente y deseable. Dicho comportamiento simplifica el tratamiento de errores, aporta seguridad al usuario final y fomenta una mayor interacción.

- Resiliente: El sistema permanece responsivo incluso en caso de errores. La resiliencia se alcanza mediante replicación, aislamiento, contención y delegación. Cada error es procesado y manejado mediante cada componente para aislarlo; asegurando así que cualquier parte del sistema que pueda fallar no altere en demasía todo el sistema y permita recuperarse sin comprometer el sistema como un todo. La recuperación de cada componente se delega en otro componente externo y la alta disponibilidad se asegura con replicación allí donde sea necesario.

- Elástico: Este rubro se refiere a la cantidad de peticiones que puede manejar, de

tal manera que pueda ser responsivo bajo variaciones en la carga de trabajo. Dicho de otro modo, pueden reaccionar a cambios en la frecuencia de peticiones, incrementando o reduciendo los recursos computacionales destinados a servir dichas peticiones. Esto implica diseños sin cuellos de botella centralizados o puntos de contención, resultando en la capacidad de dividir o replicar componentes y distribuir las peticiones entre ellos. Los sistemas reactivos soportan algoritmos de escalado predictivos, así como reactivos, al proporcionar relevantes medidas de rendimiento en tiempo real.

- Orientado a Mensajes: Los sistemas reactivos tienen un profundo intercambio de mensajes asíncronos, lo que asegura bajo acoplamiento, aislamiento y transparencia de ubicación. Este método nos proporciona los medios para delegar fallos como mensajes sin ser bloqueante ni caer de manera tajante en un error. La comunicación no-bloqueante permite a los destinatarios consumir recursos solo mientras estén activos, llevando a una menor sobrecarga del sistema.

Podemos decir entonces que el diseño del sistema que vamos a plantear pertenece a un sistema de índole reactivo donde no solo depende de los puntos a visitar para arrojar una ruta correcta, sino de la posición actual del vehículo aéreo y si este encuentra (o no) obstáculos durante su trayectoria.

1.1.2. La ruta menos costosa

Edsger Dijikstra fue un científico que propuso un algoritmo que permite navegar en un grafo buscando la planificación de trayectorias al recorrer todos los puntos. Cabe destacar que el algoritmo está hecho para un grafo ponderado de valores positivos.

Dicho algoritmo es de tipo voraz, es decir, es un algoritmo que puede obtener una respuesta globalmente correcta y óptima, recorriendo localmente los vecinos del nodo inicial y ponderando los pesos entre los recorridos y las posibles rutas. Es un algoritmo ideal para resolución de problemas en tiempo real donde el tiempo de respuesta es clave. Dicho algoritmo no funciona con pesos de arista negativos; sin embargo, para nuestro tipo de problema, esto no es un inconveniente.

Figura 1.1: Diagrama de algoritmo de la ruta más corta de Dijkstra . Fragmento tomado de [1].

El algoritmo de Dijikstra determina cuál es la ruta más corta (menos costosa) entre dos puntos en un determinado mapa. El algoritmo que se describe a continuación [1.1] reinterpreta el algoritmo de Dijikstra bajo el enfoque de vehículo aéreo.

Pseudocódigo de algoritmo de Dijikstra [1]. En dicho pseudocódigo se aprecia la lógica detrás de un algoritmo de búsqueda de ruta por medio del recorrido por los nodos adyacentes y el análisis de una discriminante que permite el descarte de las posibles rutas.

En este proyecto se ponderará y diferenciarán los nodos a visitar, lo que nos apoyará en la navegación y la discriminación de puntos. Esto es fundamental para poder determinar y caracterizar cada ruta posible.

1.1.3. Verificación de modelos

La verificación de modelos es un método de verificación que nos ayuda a analizar un sistema formal, por medio de variables descriptibles bajo términos de lógica. Es decir, la forma de procesar las entradas con este método, es mediante la abstracción de las entradas a un lenguaje lógico, esto nos permite identificar el contexto del sistema y extraer características de las entradas en variables lógicas. Posteriormente, se busca formalizar un marco lógico que permita al verificador de modelos discriminar la información recibida e interactuar con su entorno mediante la información recibida, esto haciendo usa de especificaciones en términos de fórmulas en lógica temporal. El verificador de modelos toma esta información de las entradas expresadas en variables lógicas y el marco

Algoritmo 1: Algoritmo de Dijikstra para la ruta más corta [1].

```
1  Procedure Dijkstra(G,S):
2              ▷ Función que recibe un grafo y un vértice de origen d[s] ← 0;
3                                  ▷ Inicializa la distancia desde el nodo de origen s
4  for cada vertice ≠ s en G:
5  d[v] ← ∞;
6  end
7                                  ▷ Inicializa el vector resultante con distancias infinitas
8  Q ←conjunto de todos los vertices G
9                                  ▷ Arreglo con todos los vértices a visitar
10 while Q no esté vacío do
11     u ← vertice en Q con la menor distancia d[u]
12     Q remover U
13      ▷ Remueve el vértice seleccionado en función de los resultados obtenidos con una
        función que elija de un vector el elemento más chico
14     for cada vecino v de u:
15     alt =dist[u]+peso(u,v)
16      ▷ Obtiene los nuevos vectores con la distancia relativa al vertice seleccionado para
        visitar
17     end
18     if alt ≤ dist[v] then
19         dist[v]=alt
20             ▷ Si la distancia alternativa es menor a la distancia de v se guarda el vertice
           seleccionada y se actualiza la distancia de este
```

lógico para entregar una respuesta, tal y como se puede observar en la figura [1.2].

Podemos decir entonces que una entrada en verificación de modelos es representada con un grafo dirigido que relaciona todas las variables involucradas en términos de lógica. Siendo un sistema responsivo que discrimina las entradas en términos de su relación y de valores cualitativos que se le asignen a cada nodo. Verificación de modelos representa una herramienta más que ofrece un análisis alternativo de un sistema, donde se formalizan las entradas en términos y procesamiento de variables.

La lógica temporal es una extensión de la lógica modal y es usada para describir las reglas de cualquier sistema, así como declarar los simbolismos para representar y razonar las variables lógicas, sobre proposiciones en términos de tiempo. Dicha lógica temporal posee la característica de ser de tiempo no constante, sino que están condicionadas al sistema y su respuesta. Dicho de otra manera, este tipo de expresiones otorgan al sistema la habilidad de razonar y reaccionar sobre múltiples líneas de tiempo, siendo las entradas y el sistema los que dictan la respuesta y no un algoritmo per se. Este tipo de operadores lógicos, terminan siendo de gran ayuda para poder dictar el comportamiento de un sistema formal donde el tiempo se vuelve un factor importante del comportamiento esperado.

La naturaleza de los sistemas que pueden ser analizados con este método pueden variar, algunos ejemplos serían:

- Sistemas en tiempo real: este tipo de sistema tiene un margen restringido de tiempo para proporcionar una respuesta. Esto es así, ya que dichos sistemas se enfrentan a problemas de naturaleza concurrente de eventos, por lo que dicho sistema debe de responder en un determinado tiempo para proporcionar una respuesta del mismo tipo que la naturaleza del problema. Todos los recursos de dicho sistema deben de estar enfocados en dar una respuesta y predecir el mejor resultado con la menor cantidad de operaciones posibles, por lo que el uso de un marco lógico es idóneo en este tipo de sistemas.
- Sistemas híbridos: este tipo de sistemas están compuestos por una combinación de distintas tecnologías que se comunican entre sí y el método de verificación de

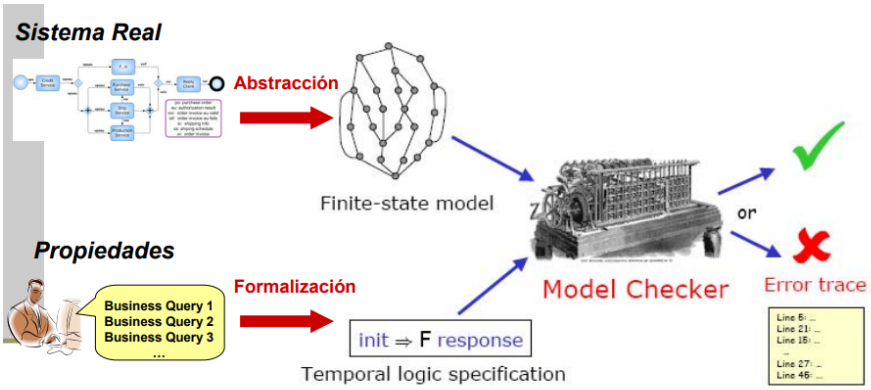

Figura 1.2: Principios del verificador de modelos ejemplificado. Fragmento tomado de diapositivas de bibliografía [2].

modelos nos permite estandarizar la respuesta entre sistemas de manera óptima, ya que de dichas entradas son posibles de abstraer su contexto y propiedades en variables lógicas. Fragmento tomado de [8] .

1.2 JUSTIFICACIÓN

Las aplicaciones de este método en cualquier tipo de problema son claramente visibles. Ya sea desde el punto computacional, reduciendo la complejidad de operaciones para entregar una respuesta, como la ventaja de la extracción de contexto de las entradas de un sistema por medio de variables lógicas.

Entre todo este mundo de aplicaciones, como lo son en sistemas industriales o de procesos de negocio, está también la aplicación en diseño de rutas; más específicamente en vehículos aéreos. Esta problemática involucra el análisis de un estado inicial, la relación entre los nodos a interaccionar (también llamado mapa) y un destino o estado final.

Actualmente, la resolución del problema de planificación de trayectorias, haciendo uso de vehículos aéreos, tiene varias vertientes. Algunos son llevados a cabo por medio de ser pilotados de forma remota (esto debido a la legislación en la que se desempeñe el vehículo aéreo), otras propuestas que otorgan autonomía al vehículo aéreo son por medio del desarrollo de un controlador que pueda manejar un sistema con entradas va-

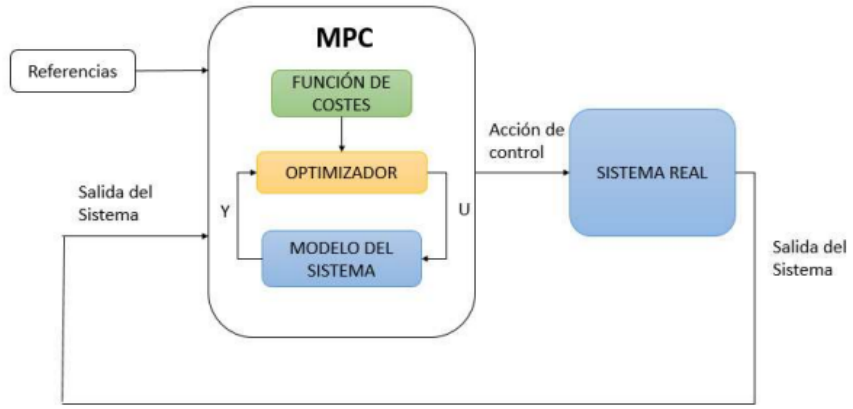

Figura 3: Esquema de control de un MPC

Figura 1.3: Diagrama de modelo predictivo. Fragmento tomado de [3].

riables y desempeñar las tareas necesarias para recorrer los puntos en cuestión.

La complejidad del desarrollo de un controlador para vehículos aéreos es motivo de investigación. La variabilidad en el entorno y la extracción de contexto nos han llevado a explorar varias propuestas de funcionalidades que solventen la necesidad planteada. Cada una implementada desde distintos puntos de abstracción; por ejemplo, desde el punto de vista de impulsos eléctricos (bajo nivel) hasta el punto de vista de abstracción de contexto basado en las entradas (alto nivel).

Cada una de estas propuestas es aceptable según la necesidad que se esté solventando, para problemas más complejos se necesitará mayor abstracción, pero se perderá exactitud y eficiencia, por otro lado, para problemas más simples, las soluciones impulsivas basadas en reglas reactivas nos dan mayor rapidez de respuesta y exactitud.

Dentro de los enfoques más abstractos de desempeño de un vehículo aéreo bajo el enfoque de planificación de trayectorias tenemos el modelo predictivo [1.3]. En él se visualiza la comunicación de diferentes módulos que, en conjunto, discriminan y entregan una salida correspondiente con el comportamiento esperado; donde cada módulo tiene una función en específico.

En dicho modelo podemos visualizar los siguientes elementos que lo componen:

- Modelo del sistema: Genera la salida del sistema a controlar ante la aplicación de determinada acción de control en un horizonte de predicción llamado N el cual nos engloba la recepción de entradas, su abstracción e interpretación.

- Función de costes: En el control predictivo es una parte fundamental, ya que de ella dependen la acción de control calculada y la estrategia buscada por el controlador. Se trata de una función que depende de las N acciones de control, las referencias en los N instantes siguientes, y las salidas predichas del sistema en el horizonte de predicción, provocadas por las N acciones de control.

- Optimizador: Es el algoritmo encargado de optimizar buscando la secuencia de N acciones de control que minimizan la función de costes.

Los modelos predictivos son modelos que usan herramientas de estadística o buscan patrones que provean de la información suficiente para poder predecir valores futuros en el sistema. Dichas herramientas se vuelven clave en el desempeño del sistema, ya que reducen en cierta medida el grado de error, siendo un sistema que contempla varios escenarios y posibles patrones de entrada que se reciban.

Para el desarrollo del presente proyecto, se presentará una propuesta de implementación de verificación de modelos en el problema de diseño de navegación de vehículos aéreos enfocado en una abstracción del problema de alto nivel; donde a partir de cualquier arreglo de puntos variable para una ruta de navegación se permita abstraer toda la información que nos concierne de dichos puntos por medio de geometría analítica, del mismo modo que se logra una abstracción de la información que recibe (en tiempo real) de los sensores. A su vez, dicho proyecto diseña y ejecuta una ruta por medio de un marco lógico que determine el orden de los puntos a recorrer según sea la especificación lógica propuesta, utilizando operaciones de baja complejidad y entregando una respuesta óptima.

Una de las ventajas de este proyecto es el nivel de abstracción que proporciona, ya que se puede extraer, de una serie de datos cuantitativos, características cualitativas de la ruta que nos permite generar un comportamiento de un vehículo aéreo de manera más natural. Sin involucrar un algoritmo por sí mismo, sino más bien involucrando una serie

de condicionales.

Un ejemplo sería que a través del conocimiento de 2 puntos se pueda sacar la distancia de manera cuantitativa y exacta como esto:

Punto 1: $x = 34, y = 5, z = 8$**; Punto 2:** $x = 38, y = 7, z = 8$

$$m = \sqrt{(34-38)^2 + (5-7)^2 + (8-8)^2} = 2\sqrt{5}$$

Y a raíz de estos datos en bruto, se pueden describir los puntos, de tal manera que comparando la distancia entre los puntos podemos describir si la distancia es mayor o no. Esto haciendo uso de un valor exacto como sería *m* con respecto a la comparación de dicho módulo del vector resultante con respecto a un valor discriminante que consideremos, donde un módulo mayor o igual a 5 es considerado "lejos" o "cerca".

Una vez obtenida esta información, podemos definir un comportamiento del vehículo aéreo para la ejecución de ruta:

$C \rightarrow V$ ▷ Si el punto está cerca (C) entonces visita (V)

De esta manera, podemos generar comportamientos más complejos, globales e intuitivos en un vehículo aéreo basándonos en datos cuantitativos de entrada, así como una función de transformación que nos dicte, con enunciados lógicos, un comportamiento de salida y no basado en un algoritmo per se, lo que nos dará un comportamiento más natural en el diseño de ruta y en consecuencia más adaptable con respecto a nuevas entradas o requerimientos emergentes.

1.3 PROBLEMA

Para este problema se presentará un simulador de diseño de ruta de un vehículo aéreo. Dicho proyecto estará implementado en un entorno virtual que nos permita visualizar la ruta trazada. Donde el simulador reciba, de un mapa, puntos variables y ejecute y genere una ruta con base en los requerimientos que se planteen.

A su vez, dicho proyecto nos proporcionará una alternativa de experimentación en un entorno virtual para vehículos aéreos.

Para el simulador se desarrolló en OpenGl y C++, en dicho lenguaje se obtenían las entradas y se enviaban a Python, donde se abstraían las entradas y se enviaban al verificador, este entrega la solución y C++ recibe dicha salida y ejecuta la ruta.

1.4 OBJETIVO

1.4.1. Objetivo general

Desarrollar un algoritmo de planificación de trayectorias para vehículos aéreos con verificación de modelos.

1.4.2. Objetivos específicos

- Implementar un algoritmo de verificación de modelos para solventar el problema de planificación de trayectoria.
- Implementar un entorno virtual para probar un algoritmo de planificación de trayectorias para vehículos aéreos.

1.5 HIPÓTESIS

A través de verificación de modelos se pueden generar trayectorias de vehículos aéreos por medio de un modelo lógico. De tal manera que el vehículo aéreo diseñe su propia ruta de manera eficiente y actué conforme a las entradas y los puntos a visitar que reciba siguiendo un marco lógico que le permita discernir, de manera más intuitiva, sobre los puntos en cuestión.

CAPÍTULO 2 MARCO TEÓRICO

A continuación se presentará el enfoque de desarrollo del proyecto empezando por analizar los conceptos de planificación de trayectorias, verificación de modelos y ambientes virtuales.

2.1 PLANIFICACIÓN TRAYECTORIAS

La planificación de trayectorias es un problema de control donde se modifica la cinemática y dinámica de un sistema con base en una serie de puntos objetivo que se desean visitar. Tiene por objetivo definir el movimiento de un vehículo, el cual implica establecer y controlar una trayectoria pre planificada. Dicha trayectoria pre planificada varía según los objetivos del proyecto y funciona dividiendo en objetivos fijos, cada tarea para la ejecución completa del objetivo global del sistema en cuestión.

El control que se ejerce durante la ejecución de una planificación de trayectorias se divide en dos etapas [10]:

- Control cinemático o planificación de trayectorias: En esta etapa se describe el movimiento deseado, se proporciona una secuencia de puntos deseados, se extraen los datos relevantes con los objetivos planteados y finalmente se interpola el camino deseado mediante una clase de funciones polinomiales y genera una secuencia de puntos a lo largo del tiempo.

- Control dinámico o control de movimiento: Supervisa y ejecuta la ruta planteada, manejando excepciones y entradas recibidas por los sensores del sistema.

Dicho esto, el problema planeado requirió de una simulación para poder ejecutar y plantear una ruta. A continuación se hablará de la serie de validaciones y estrategias que se siguieron para el desarrollo de dicho simulador.

2.2 VERIFICACIÓN DE MODELOS

La verificación de modelos es un método que propone analizar modelos de manera automática, apoyándose de la lógica temporal para su desempeño. El objetivo de este

método es proporcionar una nueva forma de análisis que nos proporcione otra alternativa en la toma de decisiones de un sistema.

Este modelo analiza de manera abstracta un problema o situación, etiquetando las variables de entrada como variables lógicas. Dichas variables lógicas significan un determinado estado de nuestro sistema de manera cualitativa y se manejan como etiquetas de los nodos existentes en el mapa. Estas etiquetas lógicas o también llamadas proposiciones atómicas, nos describirán una parte de nuestro sistema. Posteriormente, se entregan digeridas en un grafo o gráfica dirigida en donde el conjunto de nodos posee las etiquetas lógicas necesarias para describir grosso modo su estado. El verificador toma este mapa y mediante un análisis combinatorio busca los nodos que no cumplan con la negación de las cualidades enunciadas en la proposición temporal requerida. Es decir, busca un contraejemplo que cuadre con la negación de la premisa de la no existencia de un nodo o nodos que cumplan con el comportamiento solicitado en el verificador.

Un sistema se conforma por los siguientes elementos :

- Entrada: Datos obtenidos por sensores, así como otros requerimientos de usuario, los cuales son comprendidos y construidos en forma de modelo abstracto con propiedades a verificar, los cuales también están escritos en el mismo nivel de abstracción.
- Función de transformación: Recibe los valores de entrada y conforme a una función de transformación entrega una salida con relación en las entradas y las propiedades verificadas. Entregando, como resultado, si satisface la propiedad y se obtiene un contraejemplo o ruta (en caso contrario).
- Salida: Salida que es reflejada como respuesta de actuadores o conjunto de nodos a visitar, la cual fue dictada y traducida, en términos del sistema, una vez que el verificador entregue un resultado.

De esta manera, y retomando lo visto en la sección 1.1.2, podemos ver que enfocado la verificación de modelos en el problema de planificación de trayectorias, tenemos como elementos del sistema [2.1]:

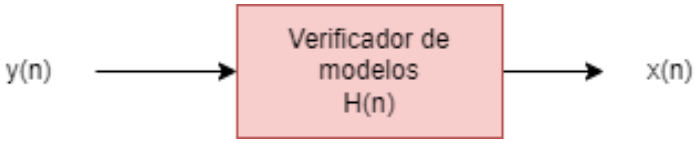

Figura 2.1: La implementación de esta propuesta nos provee una representación abstracta de un sistema en términos de lógica y donde la salida satisface los requerimientos planteados e implementados en términos de verificación de modelos.

- Entrada: vector de puntos a visitar, los cuales conservan los valores en distancia que nos permitirá tener un marco de referencia para determinar posteriormente la discriminante lógica que nos dicte el comportamiento del sistema en la función de transformación.
- Función de transformación: por medio de lógica proposicional transforma los valores de entrada a un vector resultante con los puntos a visitar según los requerimientos desglosados a base de enunciados de lógica proposicional.
- Salida: vector resultante con la serie de puntos a visitar o salida a los actuadores.

A manera de ejemplo, si se tiene una entrada que satisface el sistema, nos entregará una salida con base en el comportamiento esperado; en caso contrario (cuando las entradas no satisfacen), la salida será a un estado auxiliar con el cual se indica al sistema como reaccionar. Es decir, si deseamos pasar por los puntos cercanos y el grafo entregado, cumple aunque sea parcialmente con eso, entregará una ruta por la que pase por esos puntos deseados primero (ya que así fue determinado su comportamiento mediante un modelo lógico temporal); sin embargo, si todos los puntos estuvieran lejos, se mandaría al vehículo aéreo acercarse parcialmente a los mismos y volver a mapear su distancia posteriormente hasta que dichos puntos deseados cumplan aunque sea parcialmente con los requerimientos de entrada para entregar una salida oportuna.

Con el uso de modelos lógicos temporales se puede diseñar un algoritmo que controle y evalúe la mejor ruta, dividiendo la zona por áreas. En caso de que la visión del vehículo aéreo sea mala, el vehículo aéreo podría moverse a otra posición donde haya más probabilidad de tener mejores datos visuales. [4]

Por medio del método verificación de modelos se puede manejar diferentes objetivos y priorizar los objetivos a la hora de desenvolverse en un algoritmo de ruta inteligente. Esto gracias a que model checking (verificador de modelos) funciona con base en lógica temporal, este modelo fue hecho en un principio para el procesamiento del lenguaje natural. Esto le permite expresividad al vehículo aéreo a la hora de tomar decisiones, dicha lógica temporal proposicional en tiempo real tiene el poder de describir una secuencia de objetivos de manera condicional bajo un marco formal de referencia bien definida. Una ruta planeada en alto nivel con lógica temporal da una mayor ventaja de toma de decisiones.

Este marco formal de referencia bien definido nos da las herramientas suficientes para sintetizar y automatizar el control, así como generar código. Por medio de un modelo dinámico se evalúan las entradas como las aceleraciones deseadas. Dicho modelo discretiza en el dominio del tiempo los datos de entrada o condiciones iniciales (velocidades y aceleraciones) e inicializa el estado inicial y evalúa si las condiciones satisfacen una especificación de la fórmula lógica proposicional.

Para la planeación del recorrido de las áreas de interés, se evalúa un conjunto de áreas $P = area1, area2, ...$ Para esto, cada área contendrá información de interés como extensión, localización, casos exitosos de búsqueda, etc. Todo esto para facilitar la decisión de ruta. Posteriormente, se evalúa un conjunto potencia de P.

El algoritmo que define la ruta también debe poseer las restricciones a evaluar en la lógica proposicional, a manera de un informe informal de los operadores lógicos y temporales (eventualmente , siempre , hasta U, liberar R), la sintaxis y semántica a evaluar. Así mismo debe contener las fórmulas construidas sobre las proposiciones.

La lógica temporal proposicional nos permite validar cualquier evento y capturar cualquier comportamiento, como por ejemplo:

Evadir ciertas áreas: Se desean evadir area3,area5 y area23. Teniendo un conjunto de áreas P.

$$P = area1, area2, ...$$

La proposición lógica sería:$G[\neg(area1 \vee area5 \vee area23)]$

Para especificaciones más complejas se puede componer de una serie de reglas y especificaciones básicas relacionadas entre sí. Con esto se usaría un enfoque jerárquico para las sub proposiciones que contienen, en conjunto, una especificación compleja de un comportamiento conformado por una serie de reglas relacionadas entre sí.

Otro ejemplo con otro operador temporal sería si el vehículo tiene despejado el camino (A) entonces visita $punto1$ y $punto2$.

A lo que se usaría el operador "Eventualmente" (F) [11]. Por lo que proposición lógica sería:$F(A \rightarrow (punto1 \wedge punto2))$

Durante la planeación del recorrido se priorizarían los objetivos por medio de evaluar 3 aspectos:

1. Control de seguimiento [12]. Para ello primero se abstrae del modelo cinemático la posición 0 del dron, así como se limita la velocidad máxima del dron, este límite depende de las condiciones iniciales del modelo.

2. Satisfacción sólida de las fórmulas [12]. Teniendo una fórmula proposicional y calculando su nivel de precisión con respecto al comportamiento del vehículo aéreo y su respuesta, se deriva la fórmula y se busca una trayectoria que satisfaga la fórmula derivada, verificando que el tiempo y la precisión satisfagan la fórmula general.

3. Control híbrido para planificación de movimiento [12]. Diseñando un control híbrido que controle que la trayectoria esté en un sistema de loop que ejecute la ruta planteada.

2.3 AMBIENTES VIRTUALES

Se realizaron varias validaciones para el diseño del simulador, entre ellas se buscó un ambiente gráfico que cuente con algún lenguaje de programación que pueda implementar librerías de verificación de modelos.

2.3.1. Simulador de drones en ROS

Se buscaron referencias de códigos de simuladores en ROS [13] para tomar como referencia como una posible implementación del proyecto anteriormente planteado.

En ellos se pudieron visualizar algunas herramientas y códigos de simulación de partes de un drone como lo son:

- Las aspas girando y su velocidad dependiendo de las físicas.
- Físicas implementadas, así como la interacción con la ruta.

Se puede aprovechar un plug-in llamado Gazebo [2.2] para usar herramientas de simulación de robots. Tiene tanto entorno gráfico para visualizar mejor los valores del dron. También incluye físicas e interfaces programables para ROS.[1] Entre las ventajas que presenta esta implementación son:

- Tiene interfaz gráfica, así como el manejo de físicas.2.3
- Ya tiene desarrollado un proyecto con entorno gráfico de un vehículo aéreo, por lo que solo se concentraría en el algoritmo de calcular ruta.

A fin de ilustrar el alcance de este plug-in se hace referencia a imágenes de un proyecto en Gazebo.

Desventajas:

- Es una implementación que involucra una inversión económica, por lo que se plantearon otras opciones para el desarrollo del entorno gráfico.

Para saber más sobre esta opción se puede profundizar en ROS Development Studio – ROS DS, donde incluso se tiene cursos de ROS gratuitos y repositorios con proyectos[2].

[1] Para descargar el plug-in se puede desde la siguiente URL: http://gazebosim.org/tutorials?tut=install_ubuntu

[2] La página es: https://app.theconstructsim.com/. Funciona en cualquier sistema operativo.

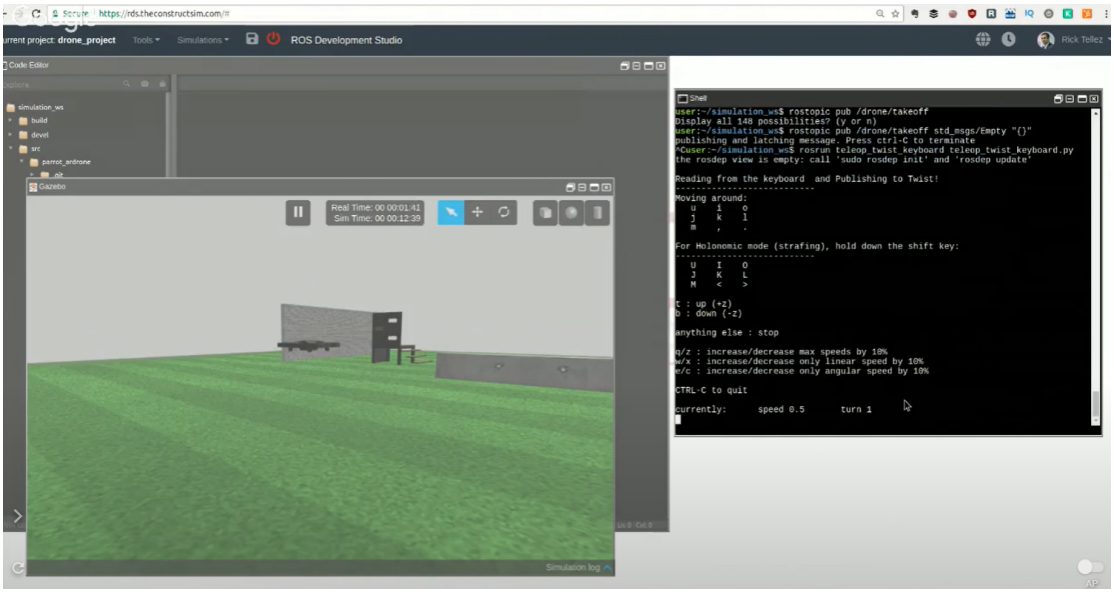

Figura 2.2: Interfaz gáfica resultante Gazebo.

En la opción de Gazebo se declaran los atributos del drone como:

- masa
- posición

Estos parámetros aportarán información para el desempeño de físicas ya implementadas en ROS para el simulador. Cabe destacar que estos valores son para ilustrar comportamientos más cercanos a físicas reales.

2.3.2. Control de robots aéreos en entorno Matlab-ROS

Matlab utiliza el paquete vrep_ros_bridge para trabajar la interfaz gráfica en gazebo. Este plug-in crea subscribers y publishers y mediante la comunicación de estos en una estructura de datos llamada "topic", se realiza la vinculación y el control entre todas las plataformas. Tomado de [14] .

- Su desventaja es que el plugin no dispone de documentación adecuada para implementar dicho comportamiento a un problema en específico, por lo que es complicado poder recompilar la información delos archivos que forman dicho plug-in para adaptarlo a las necesidades del proyecto.

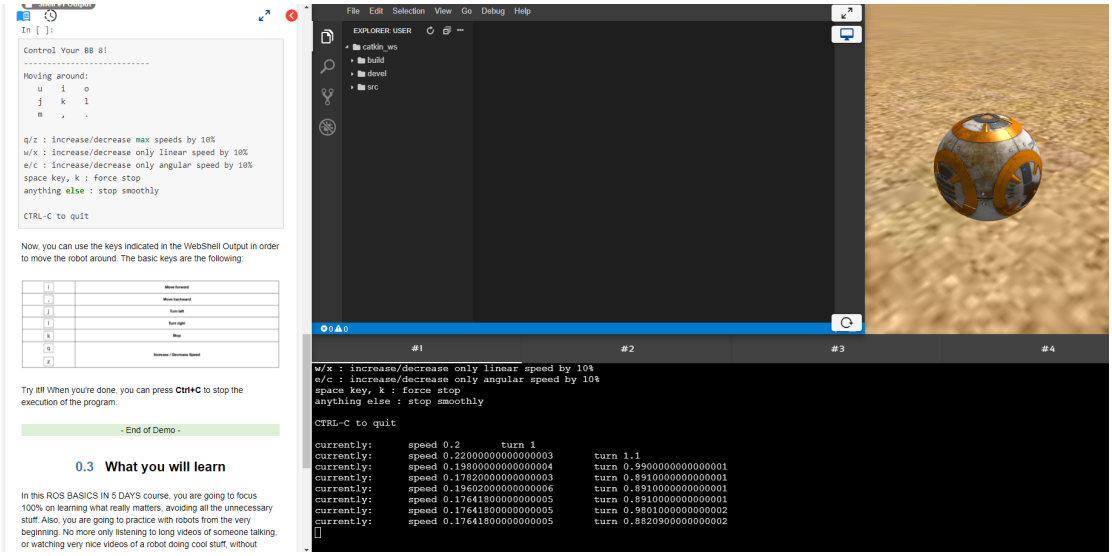

Figura 2.3: Tutorial e interfaz de desarrollo Gazebo.

- Su ventaja es que contiene la declaración de un topic llamado "effort", el cual manda 4 parámetros a V-REP a través de ROS para modificar la posición del vehículo aéreo en el simulador.

Matlab muestra cada estado del vehículo aéreo en un arreglo de datos con los campos: name, position, velocity y effort tipo float64. En el paquete vrep_ros_bridge muestra el parámetro effort con el primer y el segundo parámetro del arreglo, los cuales son el par o torque en el eje x y y respectivamente de las aspas del vehículo aéreo; el tercer parámetro es el yaw del dron (rotación alrededor del eje vertical) y el empuje o thrust.

Para el desarrollo de este proyecto primero tendría que desarrollarse cómo puede moverse dicho dron por la escena de manera real (siguiendo las leyes físicas) a partir de los cuatro parámetros de effort. Para finalmente proceder en el desarrollo de un simulador que ejecute y diseñe una ruta con base en lo solicitado por el operador.

2.3.3. Tum_ardrone mediante ROS

Tum_ardrone es un paquete desarrollado por la universidad de Munich. Tomado de [15]. Contiene las siguientes implementaciones:

- Navegación consciente de la escala de un cuadricóptero de bajo costo con una cámara monocular (J. Engel, J. Sturm, D. Cremers).

- Navegación basada en cámara de un cuadricóptero de bajo costo (J. Engel, J. Sturm, D. Cremers).
- Figura precisa volando con un cuadricóptero utilizando sensores visuales e inerciales a bordo (J. Engel, J. Sturm, D. Cremers).

La implementación mediante Tum_ardrone permite la navegación de un vehículo aéreo haciendo uso de información recibida en una webcam y extrayendo de ella los puntos de interés según sea el objetivo del proyecto.

2.3.4. Niveles de fidelidad de simuladores de drones y ejemplos con Matlab

Cabe destacar que existen diferentes niveles de fidelidad en las simulaciones de drones. Con el propósito de ilustrar los diferentes niveles de abstracción con los que trabajan dichos simuladores se hace referencia a las imágenes [2.4][2.5] donde se aprecia que el primero es más para apreciar el vuelo del vehículo aéreo y en el segundo se aprecia la ruta sin perder tiempo en descripción de físicas en la ejecución del vuelo. En función de los algoritmos que se vayan a probar y de la etapa del proceso de desarrollo, independientemente de la herramienta que se use. Las simulaciones de drones de baja fidelidad se utilizan en las primeras etapas del proceso de desarrollo, consumen menos recursos computacionales y se ejecutan con rapidez. Se pueden utilizar, por ejemplo, para ajustar los modelos de control de vuelo o para probar algoritmos de planificación de trayectorias.

Las simulaciones de drones de alta fidelidad prueban las aplicaciones en un entorno virtual más próximo al mundo real. Pueden utilizar una alta carga computacional y tardar más tiempo en ejecutarse. Se pueden usar, por ejemplo, para probar algoritmos autónomos basados en LiDAR y cámaras o el comportamiento del dron según las condiciones meteorológicas.

2.3.5. Matlab y Simulink

Nos proporciona una opción para modelar diferentes comportamientos como:

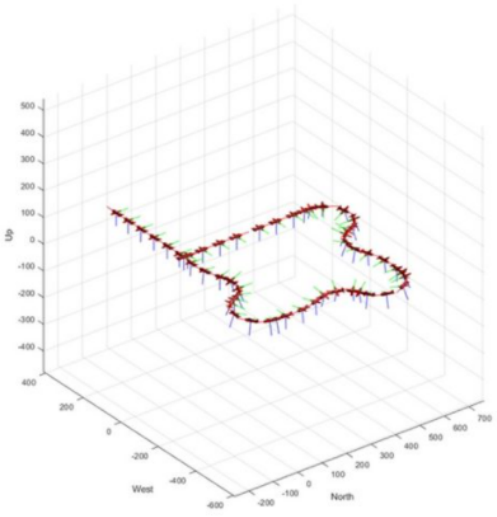

Figura 2.4: Simulación de drones de baja fidelidad con el bloque Guidance Model para VANT de MATLAB.

- Modelo dinámico del dron (modelo de planta) que consiste en las ecuaciones de movimiento del dron.
- Modelo de control de vuelo del vehículo aéreo que modela la lógica de control del vehículo aéreo.
- Modelos de sensores que simulan los sensores del dron, como GPS.
- Algoritmos autónomos que perciben el entorno e identifican los obstáculos.
- Entorno de simulación, como Cuboid World y Unreal Engine®, que son entornos virtuales creados para probar los algoritmos y visualizar el comportamiento de vuelo.

2.3.6. AirSim

Es un simulador para drones construido sobre Unreal Engine. Es de código abierto, multiplataforma y admite simulación de software en el bucle con controladores de vuelo populares como PX4 y ArduPilot y hardware en bucle con PX4 para simulaciones realistas física y visualmente. Está desarrollado como un complemento de Unreal que simplemente se puede colocar en cualquier entorno de Unreal. De igual manera, existe

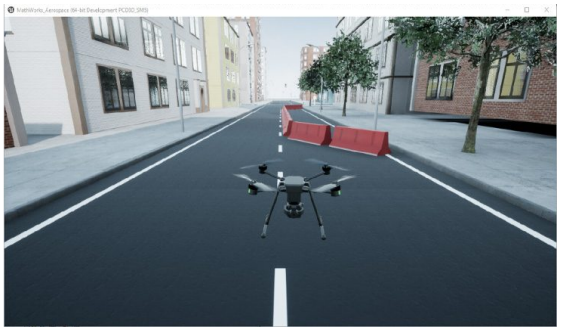

Figura 2.5: Simulación de drones de alta fidelidad mediante el bloque Simulation 3D Scene Configuration. En este caso se requiere Unreal Engine para implementar los escenarios.

una versión experimental para Unity.

2.3.7. Opengl y C++

El implementarlo en C++ con Opengl no exige grandes recursos computacionales, como lo puede ser el caso de una plataforma como es el caso de Unreal Engine.

También nos ofrece una forma de vincular varios lenguajes con el entono gráfico seleccionado, esto ayudará a escoger el mejor lenguaje para vincular con el verificador de modelos; así como poder personalizar y adecuar el código a las necesidades de lenguajes que se presenten. Esto nos permitió desarrollar, más adelante, el algoritmo de verificación de modelos que vincule los valores obtenidos del vehículo aéreo con los puntos de ruta solicitados.

2.4 IMPLEMENTACIÓN DE PROPUESTA

Al final se optó por implementar el simulador en OpenGL, ya que esta herramienta posee las funcionalidades necesarias para hacer el proyecto sin llegar a ser demasiado compleja.

2.4.1. Búsqueda de modelos y diseño en Blender

Se buscó un modelo gratuito de un dron y se fue modificando su aspecto visual en Blender. Posteriormente, se añadió el archivo en la carpeta del proyecto y se programó

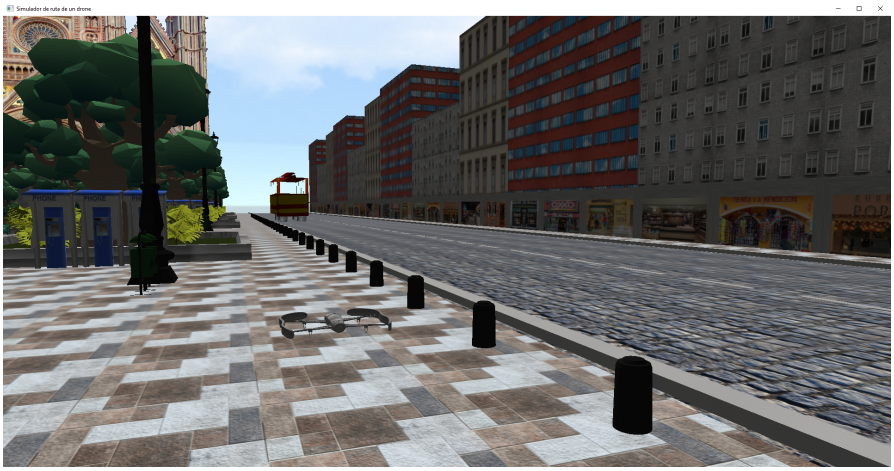

Figura 2.6: Primeras versiones del entorno virtual de diseño de ruta por model checking.

para que el modelo se dibuje en el escenario. Finalmente, se le añadieron las funcionalidades necesarias para que el modelo del vehículo aéreo se mueva por el escenario haciendo uso de las teclas correspondientes.

También se preparó el escenario para simular un ambiente real en el que podría estar un vehículo aéreo, esto con motivos estéticos.

2.4.2. Implementación de puntos de interés de un drone

Una vez teniendo un dron en movimiento (a manera "teleoperada", es decir, el usuario mueve el dron con las teclas establecidas), se procedió a marcar los puntos de interés los cuales, el dron, va a visitar. Para esto se implementaron las funcionalidades de:

- P- muestra la posición actual del vehículo aéreo.
- C- guarda todos los puntos de interés. Ya tiene almacenados 3 puntos del mapa; sin embargo, si el usuario lo desea, puede añadir más puntos con la tecla C. El máximo de puntos a guardar son 10.

2.4.3. Implementación de obstáculos en el mapa

Se determinaron 2 tipos iniciales de obstáculos en el mapa que se tuvieron en cuenta:

- Suelo.

```
if (glfwGetKey(window, GLFW_KEY_LEFT) == GLFW_PRESS) {
                std::cout << "MOVIENDO␣IZQUIERDA␣" << std::endl;
                if (validador() == true) {
                        movdx -= 0.2f;
                }
                else {
                std::cout << "Hay␣un␣obstaculo␣cerca␣" << std::endl;
                        movdx += 0.2f;
                }

        }
```

Código 2.1: movdx,movdy y movdz son variables que toman la posición actual del vehículo aéreo.

- Arboledas.

Para esto se pensó que antes de mover el dron se checaran los obstáculos que tiene registrado el mapa para calcular su distancia actual con respecto a los obstáculos, haciendo uso de un rango de acción que permita definir si "los sensores" pueden leerlo como un obstáculo cerca. Esto darán pauta para que el vehículo aéreo pueda moverse o no y a su vez nos sirve para simular un comportamiento tipo collider casero que nos permita darle un cierto tipo de comportamiento que no permita colisión entre los modelos. Es por ello que se hizo una función "válida" que checa estos dos tipos de obstáculos para arrojar un true o un falso en tiempo real respecto a la posición actual del vehículo aéreo y el mapa de obstáculos ya determinado desde el código y si esa zona puede moverse o no [2.1].

Esta función ayuda a simular si los sensores del vehículo aéreo pueden leer un obstáculo o no y con base en eso emplea una ruta de evasión de obstáculos, esto sirve tanto para el suelo como para las arboledas [2.2].

En el siguiente capítulo se va a explicar en profundidad sobre el código del proyecto, desde el aspecto de computación gráfica y su implementación.

```
bool validador(void) {
        int i;
        std::cout << "VALIDANDO MOVIMIENTO.. " << std::endl;
        //suelo
        if (movdy < -0.5f) {
                return false;
        }
        for (i = 0; i <= indice_obst1; i++) {
if ((movdx <= obstaculos1[i][0]+3.0f && movdx >= obstaculos1[i
    ][0]-5.0f)
&& (movdz <= obstaculos1[i][2]+5.0f && movdz >= obstaculos1[i][2]-4.0
    f)
&& (movdy <= obstaculos1[i][1]+4.8f && movdy >= obstaculos1[i][1]-2.0
    f))
{                       return false;
                }
        }
        //arboledas pequenas
        for (i = 0; i <= indice_obst2; i++) {
if ((movdx <= obstaculos2[i][0] +0.7f && movdx >= obstaculos2[i][0]
    -3.0f)
&& (movdz <= obstaculos2[i][2] +2.0f && movdz >= obstaculos2[i][2]
    -2.0f)
&& (movdy <= obstaculos2[i][1] +4.8f && movdy >= obstaculos2[i][1]
    -2.0f)) {
                                return false;
                }
        }
        //kiosco
if ((movdx <= -24.5f + 4.0f && movdx >= -24.5f - 4.5f)
&& (movdz <= -32.4f + 7.0f && movdz >= -32.4f - 6.0f)
&& (movdy <=  7.0f && movdy >= 0.0f)) {
std::cout << "EL DRON ESTA EN UNA COORDENADA CERCA " << std::endl;
                return false;
        }
```

Código 2.2: En dicho código compara su posición actual con la posición de los obstáculos almacenados en el mapa y el rango de tolerancia entre ellos. Si la distancia es menor al rango de tolerancia, la función retorna un falso.

2.5 CONCLUSIONES DEL ANÁLISIS DE LAS DIFERENTES HERRAMIENTAS A IMPLEMENTAR PARA LA RESOLUCIÓN DE ESTA PROBLEMÁTICA

En este capítulo se pudo apreciar varias abstracciones del problema implementadas mediante diferentes herramientas y lenguajes que atienden, de alguna manera, la problemática de ruta de un vehículo aéreo. Si bien todas tienen ciertas ventajas, es importante destacar que bajo nuestro enfoque de hacer uso del verificador de modelos como un método de diseño de ruta en un vehículo aéreo, se optó por vincular ambas herramientas haciendo uso de un ambiente virtual personalizado en C++ y Python. Siendo el vínculo mediante un fichero que comunique ambos puntos.

```
        if (glfwGetKey(window, GLFW_KEY_UP) == GLFW_PRESS) {
                if (validador() == true) {
                        movdz -= 0.2f;}
                else {
                        std::cout << "Hay un obstaculo" << std
                            ::endl;
                        movdz += 0.2f;
                }}
        if (glfwGetKey(window, GLFW_KEY_RIGHT) == GLFW_PRESS) {
                if (validador() == true) {
                        movdx += 0.2f;}
                else {
                        std::cout << "Hay un obstaculo" << std
                            ::endl;
                        movdx -= 0.2f;
                }}
        if (glfwGetKey(window, GLFW_KEY_SPACE) == GLFW_PRESS) {
                if (validador() == true) {
                        movdy += 0.2f;}
                else {
                        std::cout << "Hay un obstaculo" << std
                            ::endl;
                        movdy -= 0.2f;
                }}
        if (glfwGetKey(window, GLFW_KEY_L) == GLFW_PRESS) {
                if (validador() == true) {
                        movdy -= 0.2f;}
                else {
                        std::cout << "Hay un obstaculo" << std
                            ::endl;
                        movdy += 0.2f;
                }}
```

Código 2.3: Aquí podemos ver que al oprimirse una tecla de movimiento manda a llamar a la función validadora y dependiendo de ella se mueve el dron en esa dirección, esto es para la manera "teleoperada".

CAPÍTULO 3 PROPUESTA DE IMPLEMENTACIÓN EN UN MAVIC PRO

Se analizó también la posibilidad de implementar este algoritmo en un vehículo aéreo comercial como lo es el MAVIC PRO. El problema que se tiene con este tipo de vehículos aéreos es que no se puede acceder al valor de sus sensores en bruto, sino que solo se puede acceder, es controlar el vehículo aéreo desde una aplicación que no te indica distancias con respecto al punto inicial.

3.1 PROPUESTA Y VALIDACIÓN

Teniendo el inconveniente de no poder acceder a los datos de los sensores en bruto, y solo moverlo por medio de los botones de una interfaz de celular, nos encontramos con 2 inconvenientes principales:

- Vincular la aplicación móvil con la computadora para poder controlar el vehículo aéreo desde la computadora (esto debido a que los controles solo se encuentran desde una aplicación móvil).
- Vincular la ruta resultante con la ejecución en la aplicación móvil.

Para el primer problema, lo podemos solucionar usando de una aplicación de celular y de computadora que permite controlar dispositivos de manera remota como team Viewer, se hizo la prueba de vincular dicha aplicación con la computadora y mover el vehículo aéreo desde computadora y la prueba salió exitosa.

Hablando de la vinculación de la ruta resultante con el control del vehículo aéreo. La aplicación solo permite mover el vehículo aéreo por medio de clics de pantalla, por lo que se podría adaptar un bot que reciba la ruta y la traduzca a tiempo en clics por distancia. Para esto se propone usar Python y la librería de Selenium para programar un bot que simule los clics en el tiempo pertinente (Siguiendo el comportamiento de ruta que vimos en la sección 6.1.2).

También se puede usar de Automation Anywhere como una herramienta de automatización. Dicha herramienta permite generar un bot que emule clics, leer archivos y

ejecute rutinas con lógica condicional donde se evalúen las entradas para generar una salida. Esta es una herramienta que, en su versión gratuita, permite jugar con la automatización de manera más gráfica mediante una interfaz gráfica que incluso permite llamar a un bot a que ejecute la rutina a distancia por medio de un teléfono celular.

3.2 DESVENTAJAS DE LA PROPUESTA

Lastimosamente, no podemos acceder a los valores en bruto de los sensores de un vehículo aéreo comercial, teniendo siempre el inconveniente de que si se automatiza su navegación no podremos obtener de manera óptima la información de que existe algún objeto cerca, por lo que obstaculiza la automatización. Se puede disminuir un poco el riesgo implementando rutinas de emergencia que puedan ser ejecutadas por un operador para detener la ejecución de la ruta y evitar el obstáculo desde el bot; sin embargo, es importante que se encuentre un operador que vigile el vehículo aéreo para poder analizar la situación.

Internamente, el MAVIC PRO tiene rutinas de evasión de obstáculos, sin embargo, no son lo suficientemente complejas para evadir, sino más bien para detener la colisión.

3.3 VENTAJAS DE LA PROPUESTA

Al ser controlado por un bot, las rutas pueden ser mejor ejecutadas sobre tiempo, por lo que podría ser de gran utilidad el uso de un bot operador para la grabación de escenas o análisis de diseño de rutas aéreas.

CAPÍTULO 4 DISEÑO DEL ENTORNO VIRTUAL

A continuación se explicará el diseño e implementación del entorno virtual del simulador de ruta en la problemática de un vehículo aéreo.

4.1 OPENGL Y C++

A lo largo de la evolución del cómputo y usando de los recursos multinucleo que nos proveen los equipos de cómputo moderno, nos han permitido desarrollar soluciones gráficas que nos faciliten el aterrizaje y desarrollo de soluciones.

Open GL [16] es una extensión de C++ que permite crear una interfaz gráfica. Sus aplicaciones son variadas. En nuestro caso, se usará para el desarrollo de un entorno gráfico.

Entre las ventajas de usar open GL como la biblioteca gráfica de nuestro entorno, está C++ es compatible con C y Python, lo que nos ayuda a vincular el entorno con el software desarrollado para elegir la planificación de trayectorias.

Para la generación de este entorno se exportaron modelos gratuitos, los cuales se eligieron con la menor cantidad de vértices posibles, ya que estos consumen más recursos de procesamiento, y en consecuencia, pueden provocar un desempeño más lento de la ejecución de la planificación de trayectorias. A su vez, se personalizó cada modelo obtenido. Esto fue posible, ya que los modelos tenían las texturas aparte y se pudieron editar y añadir fotos para darle un toque de realismo.

Un ejemplo de la personalización de los modelos es con las figuras 4.1 donde podemos visualizar el entorno virtual al ejecutar el simulador. En él se personalizó cada modelo visible, desde la superficie del suelo, hasta el material del vehículo aéreo.

Tomando como ejemplo el modelo de la figura 4.2. Podemos ver cómo se personalizó el modelo de la iglesia con una foto de una catedral. A su vez, se puede visualizar que el muro de abajo se personalizó con la foto de un vitral.

Figura 4.1: Resultado final del entorno gráfico.

Esto fue posible, ya que cada figura carga su textura haciendo uso de la técnica de wrapper [17], en donde la imagen es heredada al modelo por medio de un componente de programación orientado a objetos que tiene la funcionalidad de cargar por fuerza bruta la textura y "envolver" la superficie del modelo creado. Para visualizar un archivo de textura podemos visualizar la imagen 4.3.

4.2 DECLARACIÓN DE OBJETOS EN EL ESCENARIO Y DISEÑO DE COLLIDERS

4.2.1. Bibliotecas utilizadas

Para la declaración de objetos se ocuparon las siguientes bibliotecas:

- #include <Windows.h> - esta biblioteca contiene instrucciones de Windows que nos ayudarán a desplegar la aplicación en el sistema operativo.
- #include <glad/glad.h> [18] - esta biblioteca ayuda a cargar punteros que nos permitirán dibujar los modelos solicitados.
- #include <glfw3.h> - carga los puntos de entrada principales para las versiones 3 y 4 de OpenGL.

Figura 4.2: Modelo de la iglesia

- #include <stdlib.h> - esta biblioteca contiene los prototipos de funciones de C para gestión de memoria dinámica, control de procesos y otras.
- #include <glm/glm.hpp> - permite crear matrices y vectores que nos permitirán trasladar objetos (tal es el caso del vehículo aéreo como los autobuses).
- #include <glm/gtc/matrix_transform.hpp> - permite transformar matrices, lo cual nos ayudará a dibujar modelos del entorno gráfico.
- #include <glm/gtc/type_ptr.hpp> - ayuda a que sean compatibles los tipos de matrices con OpenGL.
- #include <time.h> - ayuda a simular un comportamiento en el entorno, haciendo uso del reloj de la computadora y delays que nos permite cronometrar un comportamiento.
- #include <irrKlang.h> - es una biblioteca que posee funciones de sonido.
- #include <iostream> - es una de las bibliotecas más usadas para C++, ya que controla flujos de entrada y salida, declarando los objetos que controlan la lectura y escritura en los flujos estándar.

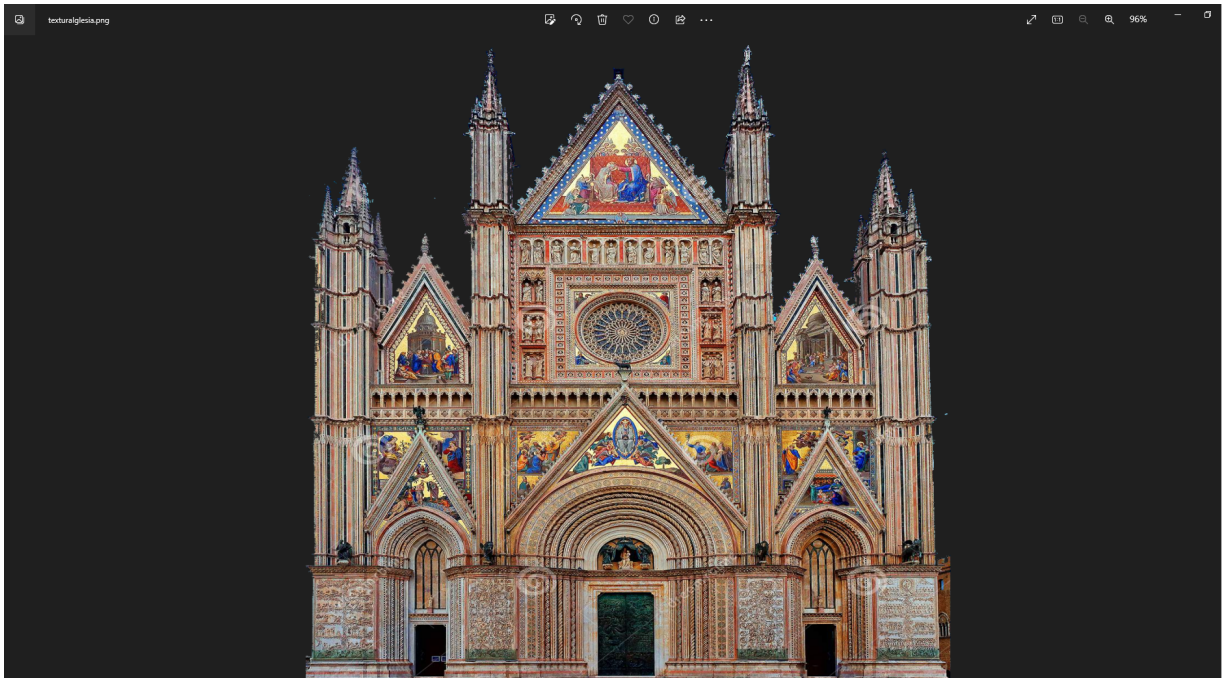

Figura 4.3: Textura de la iglesia

- #define STB_IMAGE_IMPLEMENTATION nos ayuda a definir una implementación que se usará para la biblioteca: <stb_image.h>.
- #include <stb_image.h> nos ayuda a cargar imágenes en un frame.
- #define SDL_MAIN_HANDLED define una variable que nos ayudará a implementar la biblioteca <SDL/SDL.h>.
- #include <SDL/SDL.h> es una biblioteca para el control de multimedia de una aplicación. Sirve para gestionar entradas de teclado y mouse.
- #include <shader_m.h> nos permite inicializar un shader y controlar los parámetros de este. Esto nos sirve para controlar luces y sombras en el escenario.
- #include <camera.h> permite manejar una cámara con la cual se pueden controlar los parámetros de entrada para visualizar el escenario.
- #include <modelAnim.h> nos ayuda a cargar modelos animados.
- #include <model.h> ayuda a cargar modelos con extensión .obj.
- #include <Skybox.h> ayuda a cargar un skybox que simula el cielo en un escenario.

```
void framebuffer_size_callback(GLFWwindow* window, int width, int
    height);
void mouse_callback(GLFWwindow* window, double xpos, double ypos);
void scroll_callback(GLFWwindow* window, double xoffset, double
    yoffset);

void my_input(GLFWwindow* window, int key, int scancode, int action,
int mods);
void animate(void);



unsigned int SCR_WIDTH = 1920;
unsigned int SCR_HEIGHT = 1080;
GLFWmonitor* monitors;

void getResolution(void);
```

Código 4.4: Extracción de las características del monitor.

4.2.2. Inicialización de Escenario

Se inicializaron varios valores referentes al entorno gráfico, entre ellos tenemos lo referente a la extracción de las características del monitor [4.4]. De esta forma podemos modificar el tamaño y resolución de la ventana emergente del simulador, a su vez que se extrae la posición del mouse y el scroll que permitirá navegar en el mapa. También se extraen los valores de entrada, los cuales permitirán interactuar con el simulador [4.5].

Se inicializan los parámetros para despliegue de imagen y luz para el shader. A su vez, se inicializan los parámetros de variables que ayudan a controlar la posición de los modelos desplegados en pantalla haciendo uso de un contador [4.6].

Tal y como se mencionó anteriormente, se extraen los valores del monitor para el despliegue de la ventana emergente y se inicializa el sonido por medio de $glfwInit()$; [4.7] .

Se inicializan la ventana emergente con los parámetros obtenidos e inicializados. A

```
// camera
Camera camera(glm::vec3(0.0f, 0.5f, 0.0f));
float MovementSpeed = 1.0f;
float lastX = SCR_WIDTH / 2.0f;
float lastY = SCR_HEIGHT / 2.0f;
bool firstMouse = true;

// timing
const int FPS = 60;
const int LOOP_TIME = 1000 / FPS;
double  deltaTime = 0.0f,
lastFrame = 0.0f;

//Lighting
glm::vec3 lightPosition(-15.0f, 10.0f, -20.0f);
glm::vec3 lightDirection(0.0f, -1.0f, 1.0f);
```

Código 4.5: Modificar el tamaño y resolución de la ventana emergente del simulador.

su vez se declaran las entradas de mouse, teclas y scroll, las cuales interactuarán con la ventana emergente [4.7].

Se inicializan 3 shaders, uno estático que sirve para luces, otro que nos permite inicializar el skybox y simular el cielo por medio de una "caja" la cual se declara por medio de un vector. También se declara un shader que nos permitirá cargar modelos animados[4.8].

Estos 3 shaders ayudarán a convertir vectores y transformar vectores según sea el tipo de shader. Permitiendo manipular una serie de datos mediante un pipeline por frame [4.9].

Hasta este momento hemos visto los primeros pasos para inicializar un ambiente virtual. Estas variables nos apoyarán para el control y la ambientación del ambiente virtual donde se desplegará el proyecto. Esto con el objetivo de darle un aspecto realista y visual a la ejecución del código.

```
void getResolution()
{
const GLFWvidmode* mode = glfwGetVideoMode(glfwGetPrimaryMonitor());

        SCR_WIDTH = mode->width;
        SCR_HEIGHT = (mode->height) - 80;
}



int main()
{

        glfwInit();
        ISoundEngine* engine = createIrrKlangDevice();
        if (!engine)
        {
                printf("No se pudo iniciar el motor de audio\n");
                return 0; //
        }
        engine->play2D("resources/audio/morning.wav", true);




#ifdef __APPLE__
        glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
#endif
```

Código 4.6: Se inicializan los parámetros para despliegue de imagen y luz para el shader.

```
        monitors = glfwGetPrimaryMonitor();
        getResolution();

        GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT,
"Simulador de ruta de un drone", NULL, NULL);
        if (window == NULL)
        {
                std::cout << "Failed to create GLFW window" << std::
                    endl;
                glfwTerminate();
                return -1;
        }
        glfwSetWindowPos(window, 0, 30);
        glfwMakeContextCurrent(window);
        glfwSetFramebufferSizeCallback(window,
            framebuffer_size_callback);
        glfwSetCursorPosCallback(window, mouse_callback);
        glfwSetScrollCallback(window, scroll_callback);
        glfwSetKeyCallback(window, my_input);


        glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);


        if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
        {
                std::cout << "Failed to initialize GLAD" << std::endl
                    ;
                return -1;
        }


        glEnable(GL_DEPTH_TEST);
```

Código 4.7: Despliegue de la ventana emergente y se inicializa el sonido.

```
Shader staticShader("Shaders/shader_Lights.vs", "Shaders/
    shader_Lights.fs");
        Shader skyboxShader("Shaders/skybox.vs", "Shaders/skybox.fs")
            ;
        Shader animShader("Shaders/anim.vs", "Shaders/anim.fs");

        vector<std::string> faces
        {
                "resources/skybox/right.jpg",
                "resources/skybox/left.jpg",
                "resources/skybox/top.jpg",
                "resources/skybox/bottom.jpg",
                "resources/skybox/front.jpg",
                "resources/skybox/back.jpg"
        };

        vector<std::string> facesNoche
        {
                "resources/skybox/rightNoche.jpg",
                "resources/skybox/leftNoche.jpg",
                "resources/skybox/topNoche.jpg",
                "resources/skybox/bottomNoche.jpg",
                "resources/skybox/frontNoche.jpg",
                "resources/skybox/backNoche.jpg"
        };
```

Código 4.8: Se asigna un vector para el shader perteneciente al skybox, el cual consta de una serie de imágenes cuyo número es el mismo que el número de caras que un cubo.

```
Skybox skybox = Skybox(faces);
Skybox skyboxNoche = Skybox(facesNoche);

// Shader configuration
// --------------------
skyboxShader.use();
skyboxShader.setInt("skybox", 0);

skyboxShader.use();
skyboxShader.setInt("skyboxNoche", 0);
```

Código 4.9: Se inicializa 3 shaders.

Figura 4.4: Escenario de día, tanto de shader de skybox como el de luces.

Se introducen los valores del skybox pertenecientes tanto de noche como de día [4.5][4.4][4.12].

Se declaran los modelos a dibujar en el escenario y se extrae la información de la carpeta correspondiente[4.10]. Si bien, estos modelos parecen ser más con fines estéti-

```
Model pisoDef("resources/objects/pisoDef/pisoDef.obj");
Model iglesia2("resources/objects/iglesia/iglesiav2.obj");
Model salida("resources/objects/salida1/salida1.obj");
Model drone("resources/objects/drone/drone.obj");
```

Código 4.10: Skybox.

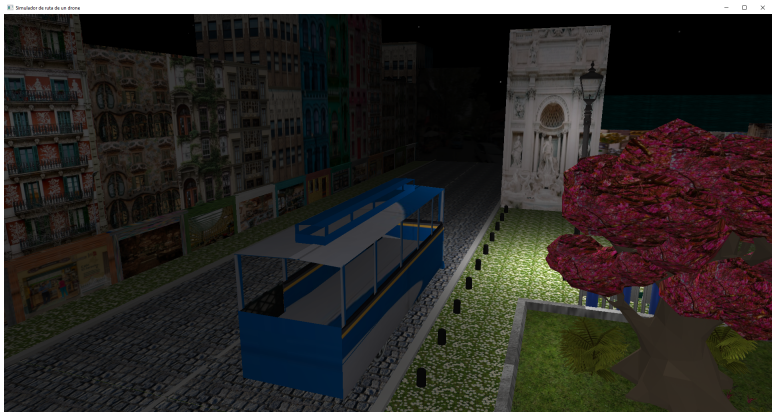

Figura 4.5: Escenario de noche, tanto de shader de skybox como el de luces.

cos; en el caso de las arboledas nos ayudan a simular obstáculos para el simulador.

Mientras la ventana exista ,se extrae la información del vector de caras del skybox correspondientes al día o noche y dependiendo de cuál se solicite, se cargan los sonidos correspondientes [4.11].

Una vez teniendo declarado el shader se dibuja según la bandera de estado que se tiene. A su vez, ya una vez declarado todo lo que se requiere dibujar en la escena, luego se ejecuta en un bucle, dando pie a que se pueda modificar el estado de las variables pivote que reciben entradas o contadores para modificar la posición a dibujar del modelo en cuestión[4.13]. Donde la función "animate" nos ayuda a dibujar el escenario [4.14].

Podemos visualizar cómo se dibuja el escenario haciendo uso de la función animate para pasar a limpiar el buffer por frame [4.14].

La declaración de luces correspondientes a si se desea simular el día y la noche se declara en [4.15]. A su vez, cabe mencionar, que se declararon una serie de luces correspondientes a las farolas.

En este fragmento podemos apreciar cómo se fueron declarando los modelos del shader estático y su despliegue en el escenario.

```
// LOOP DE RENDERIZADO
while (!glfwWindowShouldClose(window))
{
        skyboxShader.setInt("skybox", 0);
        skyboxShader.setInt("skyboxNoche", 0);
        lastFrame = SDL_GetTicks();

        if (ciclo == true) {
                if (dia == true) {
                        engine->stopAllSounds();
        engine->play2D("resources/audio/morning.wav", true);
                }
                else {
                        engine->stopAllSounds();
        engine->play2D("resources/audio/noche.wav", true);
                }
                ciclo = false;

        }
```

Código 4.11: Ejemplo de dibujo de modelos.

```
        //DIBUJO SKYBOX
                skyboxShader.use();
                if (dia == true) {
                skybox.Draw(skyboxShader, view, projection, camera);

                }
                else {
                skyboxNoche.Draw(skyboxShader, view, projection,
                    camera);
                }


                // Limitar el framerate a 60
                deltaTime = SDL_GetTicks() - lastFrame;

                if (deltaTime < LOOP_TIME)
                {
                        SDL_Delay((int)(LOOP_TIME - deltaTime));
                }
                glfwSwapBuffers(window);
                glfwPollEvents();
        }

        skyboxNoche.Terminate();
        skybox.Terminate();

        glfwTerminate();
        return 0;
```

Código 4.12: Extracción de la información para dibujar skybox.

```
        animate();
        glClearColor(0.3f, 0.3f, 0.3f, 1.0f);
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Código 4.13: Bucle con el que se dibuja escenario.

```
        staticShader.use();
        staticShader.setVec3("viewPos", camera.Position);
        staticShader.setVec3("dirLight.direction", lightDirection);
                if (dia == true) {
staticShader.setVec3("dirLight.ambient", glm::vec3(0.33f, 0.33f, 0.33
    f));
staticShader.setVec3("dirLight.diffuse", glm::vec3(1.0f, 1.0f, 1.0f))
    ;
staticShader.setVec3("dirLight.specular", glm::vec3(0,0,0));
                }
                else {
staticShader.setVec3("dirLight.ambient", glm::vec3(0.001f, 0.001f,
    0.001f));
staticShader.setVec3("dirLight.diffuse", glm::vec3(0.1f, 0.1f, 0.1f))
    ;
staticShader.setVec3("dirLight.specular", glm::vec3(0.1f, 0.1f, 0.1f)
    );
                }
```

Código 4.14: Función animate.

```
                staticShader.setFloat("material_shininess", 32.0f);
                glm::mat4 model = glm::mat4(1.0f);glm::mat4 tmp = glm
                    ::mat4(1.0f);glm::mat4 turi1 =glm::mat4(1.0f);glm
                    ::mat4 turi2 = glm::mat4(1.0f);
glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom),
(float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 10000.0f);
                glm::mat4 view = camera.GetViewMatrix();
                staticShader.setMat4("projection", projection);
                staticShader.setMat4("view", view);
                //// Light
                glm::vec3 lightColor = glm::vec3(0.6f);
                glm::vec3 diffuseColor = lightColor * glm::vec3(0.5f)
                    ;
                glm::vec3 ambientColor = diffuseColor * glm::vec3
                    (0.75f);                    model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(1.0f+movdx,0.0f+movdy,0.0f+
    movdz));
                staticShader.setMat4("model", model);
                drone.Draw(staticShader);
                for (int j = 0; j < 2; j++) {
                        for (int i = 0; i < 35; i++) {
                                model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(2.5f+(-61*j),-1.0f,-0.6f-(i
    *2.4f)));                                staticShader.setMat4("model",
     model);
                                tope.Draw(staticShader);
                                        }       }
                                model = glm::mat4(1.0f);
        model = glm::translate(model, glm::vec3(-20.0f, -1.0f, -30.0f
            ));
                model = glm::scale(model, glm::vec3(1));
                staticShader.setMat4("model", model);
                pisoDef.Draw(staticShader);
                for (int k = 0; k < 4; k++) {
                                model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(-3.5f , -0.8f, -20.6f - (24 *
     k)));      staticShader.setMat4("model", model);areasVerdes.Draw(
    staticShader);
```

Código 4.15: Shader luces

4.2.3. Animaciones

Tenemos tres tipos de animaciones en el simulador (estáticas, por medio de entrada de teclado y por medio de verificación de modelos), todas se basan en el mismo principio de controlar las posiciones de los modelos por medio de variables de estado (booleanas) o variables de tipo entero o flotante que ayuda a medir distancias durante el movimiento [4.16].

Refiriéndonos a la animación por medio de entrada de teclado se procedió mediante la extracción de los valores de entrada de la siguiente forma [4.17].

En este fragmento podemos ver cómo utiliza una función que válida si el vehículo aéreo puede moverse. Esta actúa como un collider, donde compara con un arreglo ya establecido de obstáculos y con base en su distancia con respecto al mismo recibe una respuesta de repulsión o avance. Dicha función tiene la siguiente forma [4.18].

Donde cada tipo de objeto tiene sus vectores de coordenadas correspondientes.

La animación referida al movimiento del vehículo aéreo por verificación de modelos se abordará en el capítulo siguiente; sin embargo, se recuerda que en este capítulo se ve la parte de computación gráfica que se utilizó para hacer un simulador que aporte la parte visual de la ejecución de la ruta.

```
void animate(void)
{               if(play){

                if ((movzAuto1 < 100) && (adelanteAuto1 == true)) {
                        if (rotRueda1 < 360) {
                                rotRueda1 += 2.0f;
                        }
                        else {
                                rotRueda1 = 0.0;
                        }
                        movxAuto1 = 0;
                        rotAuto1 = 90;
                        movzAuto1 += 0.3f;

                }
                else {
                        adelanteAuto1 = false;
                }

                if ((movzAuto1 > -19) && (adelanteAuto1 == false)) {
                        if (rotRueda1 < 360) {
                                rotRueda1 += 2.0f;
                        }
                        else {
                                rotRueda1 = 0.0;
                        }
                        movxAuto1 = -5;
                        rotAuto1 = 270;
                        movzAuto1 -= 0.3f;
                }
                else {
                        adelanteAuto1 = true;
                }
```

Código 4.16: Animaciones programadas en bucle.

```
if (glfwGetKey(window, GLFW_KEY_LEFT) == GLFW_PRESS) {
                std::cout << "MOVIENDO␣IZQUIERDA␣" << std::endl;
                if (validador() == true) {
                        movdx -= 0.2f;
                }
                else {
                std::cout << "Hay␣un␣obstaculo␣cerca␣" << std::endl;
                        movdx += 0.2f;
                }

        }
```

Código 4.17: Animaciones programadas por evento.

```
bool validador(void) {
        int i;
        std::cout << "VALIDANDO␣MOVIMIENTO..␣" << std::endl;
        //suelo
        if (movdy < -0.5f) {
                return false;
        }
        //arboledas grandes
        for (i = 0; i <= indice_obst1; i++) {
if ((movdx <= obstaculos1[i][0]+3.0f && movdx >= obstaculos1[i
    ][0]-5.0f)
&& (movdz <= obstaculos1[i][2]+5.0f && movdz >= obstaculos1[i][2]-4.0
    f)
&& (movdy <= obstaculos1[i][1]+4.8f && movdy >= obstaculos1[i][1]-2.0
    f))
{
                        return false;
                }
        }
        //arboledas pequenas
        for (i = 0; i <= indice_obst2; i++) {
if ((movdx <= obstaculos2[i][0] +0.7f && movdx >= obstaculos2[i][0]
    -3.0f)
&& (movdz <= obstaculos2[i][2] +2.0f && movdz >= obstaculos2[i][2]
    -2.0f)
&& (movdy <= obstaculos2[i][1] +4.8f && movdy >= obstaculos2[i][1]
    -2.0f)) {
                                return false;
                }
        }
        //kiosco
if ((movdx <= -24.5f + 4.0f && movdx >= -24.5f - 4.5f)
&& (movdz <= -32.4f + 7.0f && movdz >= -32.4f - 6.0f)
&& (movdy <=  7.0f && movdy >= 0.0f)) {
std::cout << "EL␣DRON␣ESTA␣EN␣UNA␣COORDENADA␣CERCA␣" << std::endl;
                return false;
        }
```

Código 4.18: Programación de colliders.

CAPÍTULO 5 PLANIFICACIÓN DE TRAYECTORIAS

5.1 VERIFICADOR DE MODELOS

Como ya se explicó anteriormente el concepto de verificación de modelos, en la sección 5.1 se debe explicar la herramienta utilizada, así como la aplicación de la verificación de modelos. Esto para posteriormente enfocar la herramienta de verificador de modelos en el problema en específico de diseño de ruta de vehículos aéreos, ya que verificación de modelos tiene aplicación en cualquier sistema (siempre y cuando este sea abstraído a variables lógicas) por lo que se debe de retomar con las variaciones que implica a nuestro problema planteado.

Para empezar, debemos de recordar que la verificación de modelos ocupa un análisis claro del sistema, así como una abstracción del mismo en un modelo que permita analizar el sistema en los mismos términos lógicos con los que trabaja. Formalizar las entradas de un sistema en términos matemáticos cuantificables dará pie a establecer variables lógicas que puedan abstraer el entorno en el que se envuelve el sistema para posteriormente ser analizadas mediante premisas lógicas necesarias para lograr salidas deseadas en términos de lo que se espera que sea el comportamiento del sistema.

Teniendo ya el modelo deseado, se selecciona el tipo de verificación que se desea para el análisis y entrega de salidas. Lo que vendría a ser parecida a la función de transferencia de un sistema de control, donde transforma las entradas en salidas en función con respecto a una función de transformación que entregue como salida un grafo resultante. En este caso llamaremos a dicha función "verificador".

En este caso (y buscando hacer la verificación lo más intuitiva posible) se decidió verificar el modelo mediante estructura de Kripke que nos permite analizar, mediante lógica, un sistema arrojando una salida según se declaren las premisas a filtrar y analizar.

Este análisis lógico lo podemos ver en la figura [5.1], donde tenemos una función que recibe un conjunto de estados, las variables lógicas vinculadas a cada estado y el conjunto de nodos que entrega como salida. Se inicializan las variables de conjunto de

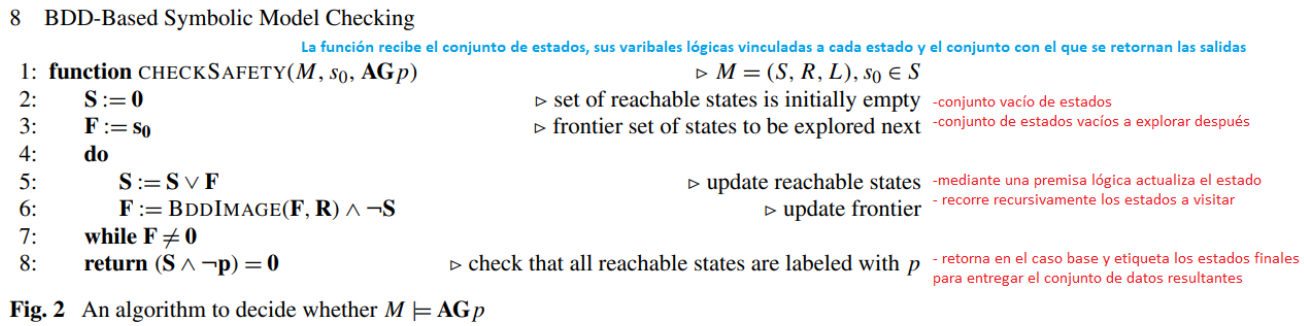

Figura 5.1: Pseudocódigo de verificador de modelos [4] página 231.

estados y se cicla que revise el estado en curso con respecto a una premisa lógica, este análisis por estado nos arroja si cumple el nodo o no con la verificación y si es así actualiza el hallazgo en una variable F y retorna cuando todo el conjunto de estados fueron verificados, entregando una serie de nodos que cumplen.

Como ya se ha comentado, verificación de modelos es un método de procesamiento por medio de un marco lógico. En nuestro caso, se deben filtrar las entradas (las cuales originalmente son una tupla de datos conocidos como la "posición a visitar" o simplemente la coordenada). De esta información en bruto se debe acotar o refinar a una variable lógica "cerca" o "lejos" que sirva para cualificar la información del punto. Esto lo podemos ver incluso en el comportamiento humano cuando discriminamos rutas de manera lógica bajo estas premisas de visitar "puntos cercanos antes de puntos lejanos" u otras.

Para acotar esta información es necesario establecer un criterio que permita definir cuando algo se encuentre lejos o cerca. Es decir, se define un marco de referencia en función del comportamiento que se desee. En nuestro caso, este criterio se estableció teniendo en cuenta el gasto de batería por distancia que se puede permitir por punto del mapa que se desee visitar.

A su vez, se opta por etiquetar también como información extra o heurística, si un punto a visitar está conectado a otros puntos (cercano a otros puntos) o está remoto, para esto también se añadió otra variable lógica que permita obtener esa información de

los puntos para poder discriminar de mejor forma la ruta final.

Finalmente, se ingresa esta información en una estructura de Kripke que nos permite filtrar y analizar cada una de las entradas de puntos vinculadas con sus variables lógicas para ir discriminando o priorizando la información pertinente según los criterios lógicos establecidos, que en este caso sería lo que intuitivamente pensaríamos como: ¿existe algún punto cerca y que a su vez esté conectado con otros?, ¿cuáles son los puntos cercanos?, ¿cuáles puntos lejanos conviene visitar primero al estar conectados?

Bajo estas premisas, el algoritmo prioriza puntos analizando recursivamente cada punto y sus etiquetas para entregar un resultado final.

5.1.1. Ejemplo de uso de una estructura de Kripke en el problema de planificación de ruta

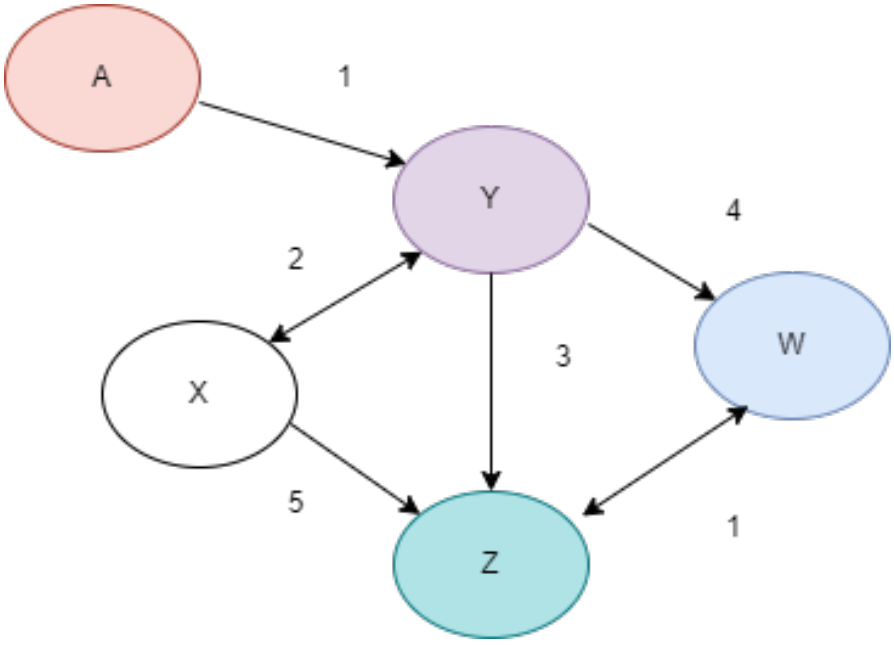

Figura 5.2: Donde cada nodo representa un punto a visitar.

Supongamos que nuestra matriz de adyacencia nos arrojó un mapa como el presentado en la figura 5.2. Donde tenemos *A* como la posición inicial del vehículo aéreo.

Nodo	Conexiones	Cercanía
Y	Conectado	1 (Cercano)
X	Conectado	3 (Cercano)
W	No conectado	5 (Lejos)
Z	No conectado	4 (Cercano)

Cuadro 5.1: Tabla de nodos etiquetados.

Para empezar se cualificará cada nodo con las etiquetas necesarias para poder discriminar y diseñar la ruta.

Refiriéndonos al nivel de conexión entre los nodos, generamos la discriminante de considerar un nodo como "Conectado" si posee 2 o más conexiones con otros nodos; en caso contrario, estamos hablando de un nodo "No conectado". Tales son los casos de Y y X. Mientras que se etiqueta como "No conectado" a los nodos W y Z.

Hablando de la distancia, clasificaremos como "Cercano" a un nodo si la suma desde la posición inicial del vehículo aéreo es 4 o menor; en caso contrario se considerará lejano. Es entonces que tenemos las siguientes etiquetas con respecto al mapa dado [5.1].

Posteriormente, establecemos una condición a cumplir en la ruta a generar mediante una proposición lógica. La cual será que visite los nodos que pasen por nodos conectados y cercanos. Teniendo entonces la siguiente expresión lógica a cumplir: $F(Conectado \wedge Cercano))$.

Siguiendo esta lógica, el algoritmo arrojaría la siguiente ruta resultante: $\{A, Y, X, Z, W\}$.

5.2 IMPLEMENTACIÓN DEL CÓDIGO DEL PROYECTO

A continuación se explicará el código implementado para la verificación de modelos del simulador del vehículo aéreo. Cabe destacar que el código que se verá en la sección 5.5 puede ser implementado para el desarrollo de un vehículo aéreo real, donde

```
void movimiento(void) {
        bool validar;
        std::cout << "Indice " << indice << std::endl;
        std::cout << "INICIANDO RUTA " << std::endl;
        discriminante = bateria / (indice-1);
```

Código 5.19: Extracción de valores de discriminante.

lo importante es extraer los datos de entrada de localización de los puntos destino para emitir una ruta deseada.

5.3 PYMODELCHECKING Y FUNCIONAMIENTO DEL CÓDIGO DE PLANIFICACIÓN DE TRAYECTORIAS

Esta biblioteca de Python nos permite hacer uso de funciones que implementan la verificación de modelos con estructuras de Kripke, las cuales nos permitirán transicionar entre estados según sean las reglas de lógica planteadas. Para esto se declaró una función en C++ que es la que interacciona directamente con el simulador gráfico y genera las variables de entrada para posteriormente pasar la matriz resultante al código en Python por medio de un fichero donde nos ayudará a planear la planificación de trayectorias haciendo uso del model checking.

5.4 ENLACE CON EL ENTORNO VIRTUAL

La función "movimiento" nos permitirá tener un enlace entre el simulador y el código en Python que diseñará la ruta según lo planteado. A continuación se explicará línea por línea el código en C++.

Para empezar, los datos de entrada de los sensores del vehículo aéreo se encuentran en variables globales, ya que esta información debe de ser accesible a todas las funciones implicadas con el comportamiento del vehículo aéreo, ya sea para modificar su posición o para tomar decisiones [5.19].

Se declara una discriminante que nos permitirá decidir si existe batería suficiente para visitar todos los puntos [5.20].

```
std::cout << "Maximo promedio de pila por punto:  "<<discriminante << std::endl;
        for (int w = 0; w < (indice-1); w++) {
                for (int i = 0; i < (indice-1); i++) {
                if (sqrt(((pos_interes[i][0] - pos_interes[w][0]) *
                (pos_interes[i][0] - pos_interes[w][0])) +
                ((pos_interes[i][2] - pos_interes[w][2]) *
                (pos_interes[i][2] - pos_interes[w][2]))) < discriminante)
                {
        matriz_adyacencia[i][w] = sqrt(((pos_interes[i][0] - pos_interes[w][0]) *
 (pos_interes[i][0] - pos_interes[w][0]))+((pos_interes[i][2] - pos_interes[w][2])
 * (pos_interes[i][2] - pos_interes[w][2])));
                        }
                        else {
                                matriz_adyacencia[i][w] = 0.0000;
                        }
                }
        }
```

Código 5.20: Generación de matriz de adyacencia.

Con base a la cantidad de batería que se puede destinar por punto a visitar (teniendo en cuenta que queremos visitar todos) se crea la matriz de adyacencia, teniendo en cuenta que el módulo de la distancia entre los puntos de interés no debe de sobrepasar la distancia que se puede visitar por punto según la batería que se tenga.

Esta matriz se guarda en un fichero para destinarla al uso del código de Python con las funcionalidades de model checking [5.22][5.21]. Se usó el fichero, ya que permite comunicar Python (que posee funcionalidades ya implementadas de verificador de modelos) sin intervenir en la ejecución del despliegue de la máquina virtual, ya que C++ al ejecutar un análisis como la verificación de modelos pausaba el despliegue de pantalla en lo que se ejecutaba. Finalmente, se manda a llamar el código de Python en consola y se ve si existe un fichero con la ruta generada por el código mencionado anteriormente.

5.5 CONSTRUCCIÓN DEL MODELO LÓGICO EN PYTHON

A continuación se explicará el código que se generó para crear el model checking [5.23]. Primeramente, se inicializan las variables necesarias para leer e interpretar el fichero.

```
        std::cout << "Matriz de adyacencia:" << std::endl;

        for (int w = 0; w < 10; w++) {
                for (int i = 0; i < 10; i++) {
                        std::cout << matriz_adyacencia[i][w] << "
                            ";
                }
                std::cout << std::endl;
        }

        ofstream fich("matriz_adyacencia_resultante.txt");
        if (!fich)
        {
                cout << "Error al abrir matriz_adyacencia_resultante.
                    txt\n";
                exit(EXIT_FAILURE);
        }

        fich << bateria << endl;
        fich << indice-1 << endl;
        fich << endl;
        for (int w = 0; w < 10; w++) {
                for (int i = 0; i < 10; i++) {
                        fich << matriz_adyacencia[i][w] << "\t";
                }
                fich << endl;

        }
        fich.close();
```

Código 5.21: Guardar matriz de adyacencia en fichero.

```
        system("python3 modelchecking.py");
        int i = 0;
        ifstream fe("ruta.txt");
        while (!fe.eof()) {
                fe >> ruta;
                cout << ruta << endl;
std::cout << "Punto a visitar " <<ruta <<". Nodo: "<< i << std::endl;
                if (i < indice-1) {
                        nodos_visitar[i] = ruta;
                }
                i++;
        }
        fe.close();


        vuelo = true;

}
```

Código 5.22: Ejecución de código Python y extracción de ruta resultante.

```
from pyModelChecking import *
from pyModelChecking.CTL import *



print('------------------------------------------------------------')
print('Inicializando los valores en el model checker: ')
print('------------------------------------------------------------')
matriz_adyacencia=[]
relaciones=[]
totales=[]



with open("matriz_adyacencia_resultante.txt") as f:
    bateria = int(f.readline())
    visitas = int(f.readline())
    for i in range(0,11):
        if(i<=visitas):
                matriz_adyacencia.append(f.readline().strip("\n"))
```

Código 5.23: Lectura de matriz de adyacencia.

Se toma los valores resultantes del código de c++ guardados en el fichero y se introducen en una lista [5.24].

Se analizan los valores de esta matriz y se obtiene una lista con los nodos que tienen alguna relación entre sí, es decir, que están conectados. Esto nos servirá para una discriminante lógica en la que veremos si el nodo a visitar cumple con el requisito de estar conectado con otros. A su vez, también se guarda la distancia para que, con base a esa lista, se pueda analizar si es un nodo cercano o no [5.25].

Se analiza la lista obtenida y se obtiene una lista con las variables lógicas relacionadas con los nodos. Estas fueron obtenidas mediante la discriminación de la información presentada en [5.25]. Posteriormente, y recordando lo enunciado en [5.1], se provee 2 parámetros a la estructura de Kripke; el primero es el grafo, el segundo es las variables lógicas relacionadas entre los nodos adyacentes.

Una vez declarada esa estructura, que en nuestro caso cuenta con 10 listas con las adyacencias de los nodos[5.26], se procede a declarar la premisa a verificar en el verificador [5.27].

Finalmente, se usa la función verificadora de modelos [5.27] donde toma la estructura de Kripke y la premisa a verificar. Generando una ruta que se imprime posteriormente.

Se obtiene el resultado y se imprime en un fichero. Luego se activa la función vuelo que nos servirá para ejecutar la ruta obtenida[5.28].

Se imprimen los valores referentes a la posición del vehículo aéreo para evaluar su comportamiento [5.29]. Esto es con motivos de revisión y visualización del comportamiento del vehículo aéreo al ejecutar la ruta.

Primero se busca nivelar la altura con un punto máximo antes de navegar en x y z. Esto porque la menor probabilidad de toparse con obstáculos es en la altura, por lo que es ideal comenzar a moverse desde ahí [5.30].

```
num=''
tot=0
nodos_comun=0
nodo=0
for i in range(visitas+1):
        if(i<=visitas and i!=0):
                nodo=0
                for n in range(len(matriz_adyacencia[i])):
                        if (matriz_adyacencia[i][n]!='\t'):
                                num=num+matriz_adyacencia[i][n]
                        else:
                                nodo=nodo+1
                                if(float(num) !=0.0):
                                        nodos_comun=nodos_comun+1
                                tot=tot+float(num)
                                relacion=[float(num),int(i),int(nodo)
                                    ]
                                relaciones.append(relacion)
                                num=''
                res=[tot,nodos_comun,int(i)]
                totales.append(res)
                tot=0
                nodos_comun=0
print(relaciones)
print(bateria)
print(visitas)
print('Totales de distancia por punto: ')
print(totales)
nodos_relacionados=[]
cualificador_nodo=[]
for i in range(len(relaciones)):
        if(relaciones[i][2]<=int(visitas)):
                if(relaciones[i][0]!=0.0):
                        par=(relaciones[i][1],relaciones[i][2])
                        nodos_relacionados.append(par)
print(nodos_relacionados)


conectado=''
```

Código 5.24: Extracción de valores de matriz en variables

```
for i in range(len(totales)):
        if(totales[i][0]<=100):
                cerca='Cerca'
        else:
                cerca='Lejos'

        if(totales[i][1]>=(visitas/2)):
                conectado='Conectado'
        else:
                conectado='MalConectado'
        resul=(conectado,cerca)
        cualificador_nodo.append(resul)

while(len(cualificador_nodo) <10):
        resul=("NoExiste","Nulo")
        cualificador_nodo.append(resul)



print('Cualificador por punto')
print(cualificador_nodo)
```

Código 5.25: Cualificar puntos.

```
try:
        K=Kripke(R=nodos_relacionados,L={1:cualificador_nodo[0],2:cualificador_nodo[1],3:cualificador_nodo[2],4:
            cualificador_nodo[3]
    ,5:cualificador_nodo[4],6:cualificador_nodo[5],7:cualificador_nodo[6],8:cualificador_nodo[7],9:
        cualificador_nodo[8]
    ,10:cualificador_nodo[9]})
except:
        print('Hubo un problema al declarar la estructura de Kripke')
if(K):
        print('Declaraci n exitosa')
else:
        sys.exit()
```

Código 5.26: Estructura de Kripke.

```
print('------------------------------------------------------------------------
    ')
print('Decidiendo los puntos:')

phi=AU(True,Or('Cerca', 'Conectado'))
res=modelcheck(K,phi)
print('Buscando nodos pibote que est n bien conectados y a su vez
    est n cerca')

print('Ruta:')
print(res)
print('
   ------------------------------------------------------------------------')
with open('ruta.txt', 'w') as f:
        for i in res:
                f.write(str(i)+' ')
        f.close()
```

Código 5.27: Verificador de modelos.

```
void vuelo_drone(void) {
        std::cout << "Ruta decidida " << nodos_visitar[0] << " , " << nodos_visitar[1] << "
 , " << nodos_visitar[2] << " , " << nodos_visitar[3] << " , " << nodos_visitar[4] << "
 , " << nodos_visitar[5] << " , " << nodos_visitar[6] << " , " << nodos_visitar[7] << "
 , " << nodos_visitar[8] << " , " << nodos_visitar[9] << std::endl;
        std::cout << "Posicion: " << std::endl;
        std::cout << "COORDENADAS EN X, Y,  Z  del drone: " << movdx << "f," << movdy << "f," << movdz << "f" <<
             std::endl;
        std::cout << "Punto a visitar: " << nodos_visitar[indice_vuelo] - 1 <<std::endl;
        if (indice_vuelo < indice - 1) {
                std::cout << "Posicion del punto a visitar:  " << pos_interes[nodos_visitar[indice_vuelo] - 1][0]
                     << " EN X, "
  << pos_interes[nodos_visitar[indice_vuelo] - 1][1] << " EN Y, " << pos_interes[nodos_visitar[indice_vuelo] -
       1][2] << std::endl;
```

Código 5.28: Impresión de localización del vehículo aéreo en tiempo real

```
if (validador() == true) {
        if (movdy < 20.0f && !altura_inicio) {
                movdy = movdy + 0.1f;
        }
        else {
                altura_inicio = true;
        }
```

Código 5.29: Iniciar vuelo.

Nos movemos en distancia x y z. Esto es comparando la posición del punto a visitar que se obtuvo de la verificación de modelos con la posición actual del vehículo aéreo y aumentado o disminuyendo la distancia en sentido de llegar al punto requerido, si llega primero a la distancia x o z se activa una bandera para que deje de moverse en ese eje [5.31].

Finalmente, se mueve en la distancia y, ya una vez cumplido la posición en x y z.

Se imprime la posición actual del vehículo aéreo para verificar su comportamiento y se incrementa el índice con el que se está recorriendo el arreglo resultante de la verificación de modelos [5.32].

```
if (movdx < pos_interes[nodos_visitar[indice_vuelo] - 1][0]&& !distanciax &&
    altura_inicio) {
    movdx = movdx + 0.1f;
    if (igualar_coordenadas(movdx, pos_interes[nodos_visitar[indice_vuelo] - 1][0])) {
        distanciax = true;
    }
}

if (movdx > pos_interes[nodos_visitar[indice_vuelo] - 1][0]&& !distanciax &&
    altura_inicio) {
    movdx = movdx - 0.1f;
    if (igualar_coordenadas(movdx, pos_interes[nodos_visitar[indice_vuelo] - 1][0])) {
        distanciax = true;
    }
}

if (movdz < pos_interes[nodos_visitar[indice_vuelo] - 1][2]&& !distanciaz &&
    altura_inicio) {
    movdz = movdz + 0.1f;

    if (igualar_coordenadas(movdz, pos_interes[nodos_visitar[indice_vuelo] - 1][2])) {
        distanciaz = true;
        }
}

if (movdz > pos_interes[nodos_visitar[indice_vuelo] - 1][2]&& !distanciaz&&
    altura_inicio) {
    movdz = movdz - 0.1f;
    if (igualar_coordenadas(movdz, pos_interes[nodos_visitar[indice_vuelo] - 1][2])) {
        distanciaz = true;
        }
}
```

Código 5.30: Movimiento para llegar a los puntos requeridos en eje x y z

```
if (movdy < pos_interes[nodos_visitar[indice_vuelo] - 1][1] && !distanciay && distanciax && distanciaz &&
      altura_inicio) {
    movdy = movdy + 0.1f;

    if (igualar_coordenadas(movdy, pos_interes[nodos_visitar[indice_vuelo] - 1][1])) {
        distanciay = true;
        }
}

if (movdy > pos_interes[nodos_visitar[indice_vuelo] - 1][1] && !distanciay && distanciax && distanciaz &&
      altura_inicio) {
    movdy = movdy - 0.1f;
    if (igualar_coordenadas(movdy, pos_interes[nodos_visitar[indice_vuelo] - 1][1])) {
        distanciay = true;
        }
}
```

Código 5.31: Movimiento para llegar a los puntos requeridos en eje *y*

```
if (distanciax && distanciay && distanciaz) {
    std::cout << "El␣dron␣ya␣llego␣al␣punto␣␣" << nodos_visitar[indice_vuelo] << std::endl;
    std::cout << "COORDENADAS␣EN␣X,␣Y,␣␣Z␣␣del␣drone:␣" << movdx << "f," << movdy << "f," << movdz << "f" << std::
         endl;

                                altura_inicio = false;
                                distanciax = false;
                                distanciay = false;
                                distanciaz = false;

                                indice_vuelo++;
                        }

                }
                else {
                        movdy = movdy + 0.1f;
                }

        }
        else {
                vuelo = false;
        }
}
```

Código 5.32: Recorrido por los puntos a visitar

CAPÍTULO 6 IMPLEMENTACIÓN DE PLANIFICADOR DE TRAYECTORIAS

A continuación se profundizará sobre la lógica detrás de la implementación del código de Python con el que se implementó la verificación de modelos.

Como se vio anteriormente en la sección 5.5 se recibe como entrada una matriz de adyacencia, la cual se genera desde el código de C++ mediante la comparación con una discriminante que nos ayudará a analizar si el nodo está conectado o no. Esto debido a que estamos manejando un mapa abierto y se debe de acotar de alguna manera, en esta caso haciendo uso de la batería total para la generación del discriminante, como vemos en el código de la sección 5.4.

Nuestro sistema de verificación de modelos nos quedaría de la siguiente manera:

El método de resolución por medio de heurísticas es una alternativa de abstracción de problemas complejos por medio de análisis de variables primarias para obtención de variables más especializadas que nos permitan las decisiones asertivas por parte del algoritmo.

En este caso, se extrajo información a partir de la matriz de adyacencia del código referido en la sección **??**, donde pudimos obtener los nodos que poseen más conexiones, así como conservar la distancia entre cada par de puntos reflejados en la matriz.

Esto nos permitió analizar en un par de listas dicha información para clasificar por nodo en variables lógicas que tomará el verificador de modelos y diseñará la ruta tomando en cuenta las reglas lógicas que se establezcan con base en esto. El código donde se puede apreciar esta parte se encuentra en **??**.

Posteriormente, se declara el verificador de modelos y se le declara el enunciado lógico prioritario para el diseño de ruta donde se pide que priorice los nodos cercanos o conectados a otros nodos. Esto se visualiza en la sección 5.5.

Figura 6.1: Diagrama de algoritmo implementado de verificación de modelos.

Finalmente, el resultado es guardado en un archivo que se pueda leer en C++ para que el simulador pueda leerlo y ejecutarlo. Esto validando que no exista algún obstáculo con la función de validador, si existiera procede a subir lo suficiente para evadirlo como podemos ver en la sección 5.5. Finalmente, incrementa su posición, según sea el caso, para irse acercando al nodo solicitado en la ruta trazada, tal y como se visualiza en la sección 5.5. Esto se repite hasta que todos los nodos hayan sido visitados.

6.1 EJEMPLO DE FUNCIONAMIENTO

Ya hemos explicado el funcionamiento del código y la forma en la que se abstrajo el problema. Ahora procederemos a explicar el proceso con el que nuestro algoritmo genera respuesta desde un ejemplo.

6.1.1. Verificador de modelos

Ejemplo de diseño de ruta con todos los puntos sin criterios en común

Para iniciar necesitamos nuestras entradas en forma de coordenada de puntos, en este caso le daremos 3 puntos a visitar:

- Punto 1 (3,3,3).
- Punto 2 (6,6,6).
- Punto 3 (30,30,30).

Para poder tener una discriminante que nos permita visualizar si los puntos están cerca o conectado, se pondrá la información en una matriz de 3x3. Se determina un valor

máximo para determinar si un punto es adyacente a otro. En nuestro caso, primero analizaremos que nuestro dron tiene 30 puntos de batería, por lo que si repartimos equitativamente la batería entre los puntos a visitar tendríamos un máximo de 10 puntos de distancia por punto.

Se analiza la distancia del módulo del vector que relaciona la coordenada entre los puntos a visitar para determinar cuál punto puede ser considerado cercano o no. De esta forma la lógica sería:

$$\sqrt{(3-6)^2+(3-6)^2+(3-6)^2} = 3\sqrt{3} = 5.19$$

$$\sqrt{(6-30)^2+(6-30)^2+(6-30)^2} = 24\sqrt{3} = 41.5$$

$$\sqrt{(3-30)^2+(3-30)^2+(3-30)^2} = 27\sqrt{3} = 46.7$$

Como tenemos un módulo máximo de 10 de batería por punto, nuestra matriz de adyacencia nos quedaría de la siguiente forma contando la distancia que existe por punto cercano:

$$\begin{matrix} 0 & 5.19 & 0 \\ 5.19 & 0 & 0 \\ 0 & 0 & 0 \end{matrix} \tag{6.1}$$

También determinamos una discriminante de la cantidad de puntos a los que debe de ser adyacente un punto para estar conectado o remoto. En este caso, diremos que si tiene al menos 1 vecinos adyacentes a él, quiere decir que está conectado. Siguiendo este criterio podemos decir que el punto 1 y 2 están conectados o tienen más nodos relacionados con ellos intrínsecamente.

Se añade una discriminante que nos indique si hay puntos más cercanos que otros en la matriz de adyacencia, para este ejemplo vamos a determinar que es cercano si es menor a 6, por lo que los puntos que obtenemos tendrán las siguientes etiquetas:

- Punto 1: Cercano y conectado.
- Punto 2: Cercano y conectado.
- Punto 3: Lejos y no conectado.

Teniendo esto, se pasa esta información al verificador (el cual priorizará los nodos según las premisas lógicas que se determinen) donde se definirán las siguientes premisas:

- (Cercano $\wedge$ Conectado) $\vee$ Conectado

Por lo que el verificado priorizará el punto 1 y 2 a visitar entregando el siguiente resultado: 1,2,3.

Ejemplo de diseño de ruta con puntos con un criterio en común y operadores temporales

Para iniciar necesitamos nuestras entradas en forma de coordenada de puntos, en este caso le daremos 3 puntos a visitar:

- Punto 1 (3,3,3)
- Punto 2 (3,6,6)
- Punto 3 (3,4,3)

Para poder tener una discriminante que nos permita visualizar si los puntos están cerca o conectado, se pondrá la información en una matriz de 3×3. Se determina un valor máximo para determinar si un punto es adyacente a otro. En nuestro caso, primero analizaremos que nuestro dron tiene 30 puntos de batería, por lo que si repartimos equitativamente la batería entre los puntos a visitar tendríamos un máximo de 10 puntos de distancia por punto.

Se analiza la distancia del módulo del vector que relaciona la coordenada entre los puntos a visitar para determinar cuál punto puede ser considerado cercano o no. De esta forma la lógica sería:

$$\sqrt{(3-3)^2+(3-6)^2+(3-6)^2} = 3\sqrt{2} = 4.24$$

$$\sqrt{(3-3)^2+(3-4)^2+(3-3)^2} = 1$$

$$\sqrt{(3-3)^2+(6-4)^2+(6-3)^2} = \sqrt{13} = 3.6$$

Como tenemos un módulo máximo de 10 de batería por punto, nuestra matriz de adyacencia nos quedaría de la siguiente forma contando la distancia que existe por punto cercano:

$$\begin{matrix} 0 & 4.24 & 1 \\ 4.24 & 0 & 3.6 \\ 1 & 3.6 & 0 \end{matrix} \tag{6.2}$$

También determinamos una discriminante de la cantidad de puntos a los que debe de ser adyacente un punto para estar conectado o remoto. En este caso, diremos que si tiene al menos 1 vecinos adyacentes a él, quiere decir que está conectado. Siguiendo este criterio podemos decir que los puntos llegan a estar conectados entre sí y los 3 cumplen con ese criterio a discriminar, ya que tienen más nodos relacionados con ellos intrínsecamente.

Se añade una discriminante que nos indique si hay puntos más cercanos que otros en la matriz de adyacencia, para este ejemplo vamos a determinar que es cercano si es menor a 3, por lo que los puntos que obtenemos tendrán las siguientes etiquetas:

- Punto 1: Cercano y conectado.
- Punto 2: Lejos y conectado.
- Punto 3: Cercano y conectado.

Teniendo esto se pasa esta información al verificador, el cual priorizará según las premisas lógicas que se determinen, porque se definirán las siguientes premisas:

- (Cercano $\wedge$ Conectado) $\vee$ Conectado
- F($A \rightarrow R$) donde definimos que A es que el vehículo está libre de obstáculos y R significa seguir con la ruta trazada.

Por lo que el verificado priorizará el punto 1 y 3 a visitar siempre y cuando no existan obstáculos en el camino, es decir, siempre y cuando se cumpla la segunda premisa. Entregando el siguiente resultado: 1,3,2.

6.1.2. Ejecución de ruta

Una vez teniendo la ruta se pasa a ejecutarla, donde se programó un collider de los obstáculos que existen en el escenario. Esto nos sirve a manera de simular un sensor que nos indique si hay objetos cerca.

El vehículo aéreo pasa a subir una distancia en y esperando que en esa distancia pueda moverse con más facilidad por eje x y z (esto debido a que entre más altura menos obstáculos). Se mueve hasta llegar a esas coordenadas en x y z y finalmente ajusta su altura en función de las coordenadas en y.

En caso de que detecte algún obstáculo enfrente se procede a elevar la distancia esperando que pueda ya moverse con libertad en el eje x o z.

CONCLUSIONES Y TRABAJO A FUTURO

En conclusión, la verificación de modelos tiene múltiples usos mediante distintos enfoques de análisis en función de la naturaleza del problema, siendo clave la elección de la función de transformación pertinente para la obtención de las salidas deseadas de un sistema. En nuestro caso, la estructura de Kripke nos permitió analizar, de manera más intuitiva y completa, un problema aparentemente trivial como lo es seleccionar una ruta. A manera de abstraer, de una coordenada, información valiosa que nos permita discernir cuáles puntos se conviene visitar primero; todo esto mediante lógica, proposiciones y una serie de premisas clave en la toma de decisiones. Este tipo de marco lógico nos permite acercar más el análisis de problemas al lenguaje humano, donde haciendo uso de nuestro sentido común (serie de premisas lógicas) se toma decisiones.

De la misma manera, se puede observar que el desarrollo de un entorno gráfico (haciendo uso de OPENGL y C++) nos permite simular comportamientos y analizar posibles soluciones del mismo mediante el abordar el problema de manera más visual y controlada. De esta forma, se pudo generar rutas y programar rutinas de evasión de obstáculos, a modo de emergencia, en caso de que en la ruta generada llegara a interponerse un objeto en el camino trazado.

Por otra parte, bajo la lógica de solución planteada ante el problema de diseño de ruta en vehículos aéreos, se pudo aportar con una propuesta de implementación de planificación de ruta mediante estructuras de Kripke y verificación de modelos en vehículos aéreos comerciales, como lo es un mavic pro. Donde (a pesar de las limitantes que tiene a nivel control como vehículo aéreo comercial) se puede manipular y ejecutar trayectorias mediante la implementación de un bot que ejecute la ruta planeada, emulando los clics necesarios para interactuar con la interfaz.

Cabe destacar que el modelo del sistema, también es clave para obtener salidas más eficientes y refinadas. Siendo que, entre más información lógica se pueda obtener de una entrada, podemos discriminar mejor la información y en consecuencia tener una salida que sea más completa y acertada, por lo que nos podemos apoyar de heurísticas para refinar mejor las entradas.

Si bien existen varios inconvenientes en la navegación de un MAVIC PRO, se puede implementar este modelo en un vehículo aéreo no comercial. Donde, al igual que en el simulador, se puedan definir rutinas de evasión de obstáculos más idóneas en función de los valores que se vayan obteniendo de los sensores. A la vez se puede ejecutar la ruta haciendo uso del algoritmo de verificación de modelos planteado en esta tesis.

Para trabajos futuros, se puede analizar alguna problemática específica, como lo puede ser rutas para cinematografía, donde ocupan vehículos aéreos. Siendo el operador un bot, se puede contar con una precisión en milisegundos, por lo que añadiendo variables de operación como lo es rotación de la cámara (en el caso de MAVIC PRO) o lo es en velocidad. A su vez que se puede contar con una ruta de menor costo o, si es necesario, modificar las premisas para la toma de decisiones según lo requiera la complejidad de la ruta.

Como se mencionó, uno de los mayores problemas con los que se contó fue que los vehículos aéreos comerciales no permiten que uno pueda acceder de manera primitiva a los datos de los sensores en bruto, a su vez que tampoco se puede inyectar voltaje a los actuadores sin tener que interactuar más bien con los controles de la interfaz para que esta traduzca los movimientos. Esto no permite que haya un performance del vehículo que sea inexacto, hasta cierto punto, ya que no contamos con los valores de primera mano. Sin embargo, eso no impida que no se pueda implementar una solución que permita el uso de planificación de trayectorias en un vehículo comercial.

Bibliografía

[1] Adeel Javaid. Dijkstra's algorithm. *Available at SSRN 2340905*, 2013.

[2] María José Ibañez. Model checking. Instituto de Investigación en Ingeniería de Aragón (I3A) http://webdiis.unizar.es/~ezpeleta/lib/exe/fetch.php?media=misdatos:pc:introduccion_al_model_checking.pdf, Mayo 2010. Recuperado 12 de diciembre de 2023.

[3] Adrià Torres Jurado. Diseño de controladores predictivos multivariables para uavs. 2015.

[4] Edmund M Henzinger y otros. *Handbook of model checking*, volume 10. Springer, 2018.

[5] Daniel R Harrell y otros. Dijkstra's algorithm y google maps. In *Proceedings of the 2014 ACM Southeast Regional Conference*, pages 1–3, 2014.

[6] Enric Galceran y Marc Carreras. A survey on coverage path planning for robotics. *Robotics y Autonomous systems*, 61(12):1258–1276, 2013.

[7] Reynaldo Martell Avila. *Simulador de robots móviles*. PhD thesis, Facultad de ingeniería, Octubre 2016.

[8] Cruz Jiménez BJ. Modelación y análisis de un sistema híbrido: Un caso de estudio con un sistema de tanques.

[9] Luca Ingólfsdóttir y otros. *Reactive systems: modelling, specification y verification*. cambridge university press, 2007.

[10] Valentín Osimani. Planificación de trayectorias óptimas. 2022. Universidad de la República (Uruguay). Facultad de Ingeniería.

[11] Vasumathi Donzé y otros. Reactive synthesis from signal temporal logic specifications. In *Proceedings of the 18th international conference on hybrid systems: Computation y control*, pages 239–248, 2015.

[12] Georgios E Girard y otros. Temporal logic motion planning for dynamic robots. *Automatica*, 45(2):343–352, 2009.

[13] Rafael Bermúdez y otros. A simulation framework for developing autonomous drone navigation systems. *Electronics*, 10(1):7, 2020.

[14] Óscar Pérez Gil. Control de robots aéreos en entorno matlab-ros utilizando el simulador v-rep. 2018. Universidad de Alcalá. Escuela Politécnica Superior.

[15] Francesco d'Apolito y Christoph Sulzbachner. Obstacle avoidance system development for the ardrone 2.0 using the tum_ardrone package. In *2017 Workshop on Research, Education y Development of Unmanned Aerial Systems (RED-UAS)*, pages 49–54. IEEE, 2017.

[16] Jiaji Deng y otros. 3d terrain real-time rendering method based on cuda-opengl interoperability. *IETE Technical Review*, 32(6):471–478, 2015.

[17] Andrés Ramírez Escobar. Conjunto de herramientas para el diseño de aplicaciones en directx9. 2009. Universidad EIA.

[18] Donald Baker y otros. *Computer graphics with OpenGL*, volume 3. Pearson Prentice Hall Upper Saddle River, NJ:, 2004.