

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

TRABAJO DE FIN DE MÁSTER

DISEÑO DE MESA INTERACTIVA PARA ANÁLISIS DE MOVILIDAD Y MODELADO DE ESCENARIOS URBANOS

Autor: Ignacio de Rodrigo Tobías

Directores:
Alberto Carnicero López
Jaime Boal Martín-Larrauri

MADRID

27 de agosto de 2020

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título Diseño de mesa interactiva para análisis de movilidad y modelado de escenarios urbanos

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2019/2020 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Ignacio de Rodrigo Tobías Fecha: 26 / 8 / 2020



Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO



Firmado digitalmente por CARNICERO LOPEZ ALBERTO - 50442634T
Nombre de reconocimiento (DN): c=ES, serialNumber=IDCES-50442634T, givenName=ALBERTO, sn=CARNICERO LOPEZ, cn=CARNICERO LOPEZ ALBERTO - 50442634T
Fecha: 2020.08.27 09:39:57 +02'00'

Fdo.: Alberto Carnicero López Fecha: ...../ ...../ .....



Firmado digitalmente por BOAL MARTIN LARRAURI JAIME - 05304600H
Fecha: 2020.08.27 09:45:04 +02'00'

Fdo.: Jaime Boal Martín-Larrauri Fecha: ...../ ...../ .....



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

TRABAJO DE FIN DE MÁSTER

DISEÑO DE MESA INTERACTIVA PARA ANÁLISIS DE MOVILIDAD Y MODELADO DE ESCENARIOS URBANOS

Autor: Ignacio de Rodrigo Tobías

Directores:
Alberto Carnicero López
Jaime Boal Martín-Larrauri

MADRID

27 de agosto de 2020

DISEÑO DE MESA INTERACTIVA PARA ANÁLISIS DE MOVILIDAD Y MODELADO DE ESCENARIOS URBANOS

Autor: Rodrigo Tobías, Ignacio de.

Directores: Carnicero López, Alberto y Boal Martín-Larrauri, Jaime.

Entidad colaboradora: ICAI – Universidad Pontificia Comillas.

Resumen del proyecto

El desarrollo de las zonas urbanas ha sufrido una gran aceleración durante las últimas décadas y se espera que esta tendencia no sólo continúe, sino que se acentúe. Asociado a este crecimiento exponencial de las ciudades, aparece un problema de movilidad en sus calles. Las distancias recorridas, los tiempos empleados o la peligrosidad que entrañan los desplazamientos de los ciudadanos son parámetros que deben tenerse en cuenta a la hora de diseñar un entramado urbano. En este proyecto se estudia cómo aportar un producto que sirva como lienzo en blanco sobre el que realizar diseños de ciudades de una forma colaborativa e interactiva. Así, a partir del uso de técnicas de visión artificial, los diseñadores pueden recrear un modelo virtual a partir de la manipulación de herramientas sencillas, como piezas de Lego con las que modelar edificios o rotuladores con los que trazar calles. A partir de la generación del modelo virtual, se pueden realizar simulaciones de movilidad con los que contrastar las diferentes propuestas. De esta forma, se consigue que el proceso de diseño de los espacios urbanos del futuro sea más transparente y completo.

Introducción

El proyecto describe el diseño de una herramienta —mesa interactiva inspirada en desarrollos previos del MIT Media Lab [1]— con la que se pretende favorecer el modelado urbano desde una perspectiva multidisciplinar y en la que se evalúa la calidad de la movilidad de cada modelo propuesto. Así, en la toma de decisiones pueden incluirse variedad de perfiles, de tal forma que cualquier persona puede aportar su visión gracias al uso de herramientas intuitivas.

Sobre la mesa se disponen piezas de Lego que modelan diferentes enclaves de la ciudad —colegios, hospitales, zonas verdes, etc.—. Además, los usuarios pueden trazar sobre la mesa los recorridos de las calles. Mediante visión artificial,



Figura 1: Bloques del proyecto junto con la secuencia de funcionamiento de los mismos.

se reconocen cada una de las piezas o trazos y se les asocia información. Gracias a estos datos de entrada, se ejecuta una simulación en el *software* Aimsun [2] con el que evalúa la calidad del diseño atendiendo a los flujos de movilidad que se establecen. Finalmente, los resultados de la simulación se muestran a los usuarios en un entorno virtual recreado en Minecraft.

Objetivos

El proyecto se enfoca en concebir un producto que ayude a diseñar enclaves urbanos evaluando su demanda de movilidad. Además, se pretende conseguir democratizar el proceso de diseño y facilitar la divulgación de los resultados alcanzados. Por tanto, los objetivos que se establecen son:

- Dotar al usuario de un método intuitivo y ágil para poder localizar edificios y trazar vías de comunicación.
- Integrar un modelo de movilidad que represente de manera fiel los desplazamientos que se generan a partir del modelo físico —en su defecto se debería desarrollar un modelo de menor complejidad—.
- Mostrar al usuario de manera sencilla y visual los resultados alcanzados mediante la simulación.
- Integrar todos los elementos de manera compacta y que favorezca la interacción entre los usuarios y al mismo tiempo entre usuario y máquina.

Solución

El modelo que se implementa se estructura en tres fases: entrada de datos, procesamiento y muestra de resultados. En la Figura 1 se muestra un esquema que describe la secuencia de los tres bloques mencionados.

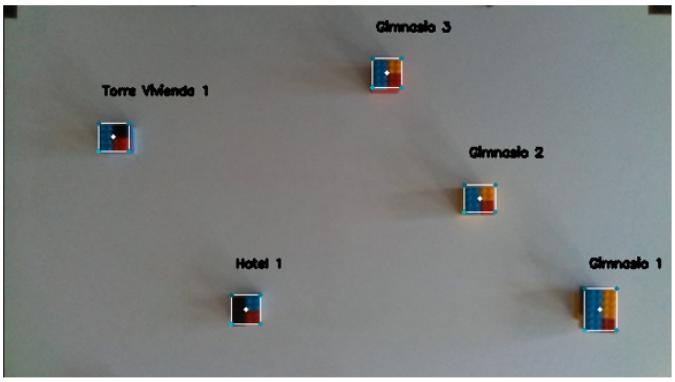


Figura 2: Ejemplo de entrada de datos. Detección de diferentes tipos de patrones mediante el uso del sistema de detección YOLO v3.

- **Entrada de datos**. En el proyecto, se utiliza la cámara Intel Realsense D435 [3] para captar la información de los elementos que se disponen sobre la mesa —piezas de Lego y trazos de rotulador—. Con las piezas de Lego se puede formar patrones de colores. A partir de los patrones que se incluyen en el modelo, se pueden identificar un total de 19 enclaves urbanos diferentes como torres de oficinas, hospitales, restaurantes o estadios. Por otro lado, las líneas trazadas con rotulador permiten identificar las calles de la zona urbana que se desee simular. Una vez tomadas las imágenes, se da paso a la aplicación de algoritmos de obtención de datos. Para la detección de los patones de colores, se utiliza YOLO v3 [4], mientras que para la detección de carreteras, se utiliza la librería OpenCV [5]. Una vez extraídos los datos, éstos se recogen en archivos '.xml'. En la Figura 2 se muestra un ejemplo de entrada detección de diferentes tipos de patrones de colores.

- **Procesamiento**. A partir de los archivos '.xml' generados, se dispone de la información necesaria para recrear los escenarios en el *software* de simulación de movilidad, Aimsun. Una vez obtenidos los escenarios —generados mediante código— se lanza la simulación. Los resultados de la simulación muestran las coordenadas y tiempos de los diferentes trayectos realizados. Estos resultados se recogen de nuevo en un archivo '.xml'.

- **Muestra de resultados**. Finalmente, los resultados de la simulación se incluyen en un escenario virtual recreado en Minecraft.

Según lo expuesto en la enumeración anterior, en la Figura 3 se muestra un diagrama más detallado. Los bloques destinados a la detección de líneas y detección de piezas están asociados a la entrada de datos descritos en la Figura 1.

Resultados

La suma de los diferentes bloques descritos posibilita la captación y extracción de datos asociados a líneas curvas de geometrías complejas, la detección y clasificación

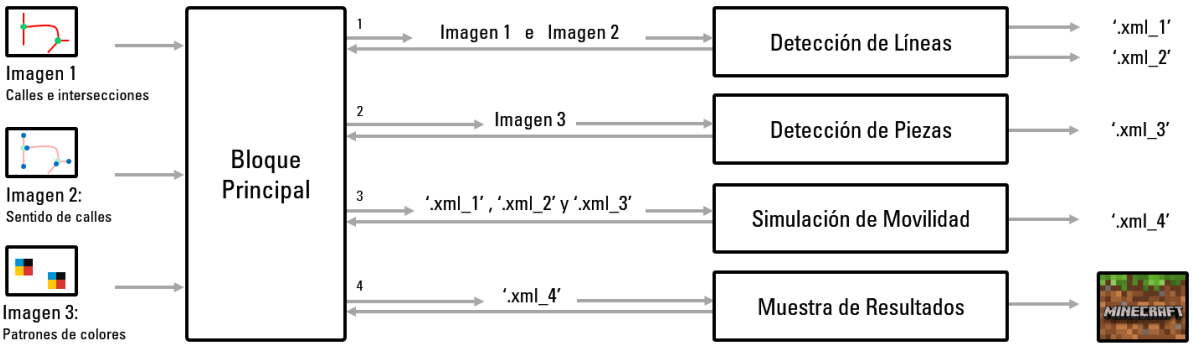


Figura 3: Diagrama de bloques del proyecto ampliado.

de patrones de colores, la recreación de manera autónoma de escenarios urbanos mediante el uso de código, y la recreación de escenarios virtuales.

La suma y coordinación de estos bloques ofrece como resultado una plataforma en la que simular escenarios urbanos sencillos. Por un lado se aporta la flexibilidad y facilidad de edición de modelos a través del uso de elementos sumamente intuitivos, mientras que por otro, se ofrece el respaldo y fiabilidad de un *software* contrastado. Es decir, el producto que se genera permite salvar las limitaciones asociadas al uso del software, simplificando el proceso de generación de modelos y abriendo la posibilidad de que este proceso sea mucho más participativo. En la Figura 4 se muestra un caso de estudio sencillo en el que muestran los elementos físicos que sirven como entrada de datos —trazos de rotulador y piezas de Lego—, y se muestran los resultados de la simulación y en el entorno virtual.

Conclusiones

Como se menciona en el apartado anterior, los resultados de este trabajo posibilitan utilizar el producto final para la generación y estudio de escenarios sencillos. De esta forma, se consigue aportar una estructura que posibilita la captación de datos de un modelo físico, la comunicación y ejecución de simulaciones en un *software* especializado, y la recreación de un escenario virtual.

El nivel de desarrollo de la estructura no alcanza todo su potencial en esta versión del proyecto, y se considera que existen posibilidades de mejorar ampliamente sus prestaciones en aspectos como la asociación de datos reales, la aportación de mayor nivel de detalle a los modelos, o la mejora de la experiencia del usuario final. Por ejemplo, a la hora de modelar los desplazamientos con mayor detalle y fiabilidad, es deseable introducir al modelo información de bases de datos —Google, ayuntamientos, etc.— o realizar estudios locales de movilidad con tecnología como la que ofrece Numina [6]. Por otro lado, desde el punto de vista de la mejora de la experiencia de los usuarios, se pueden mejorar los gráficos del entorno virtual o

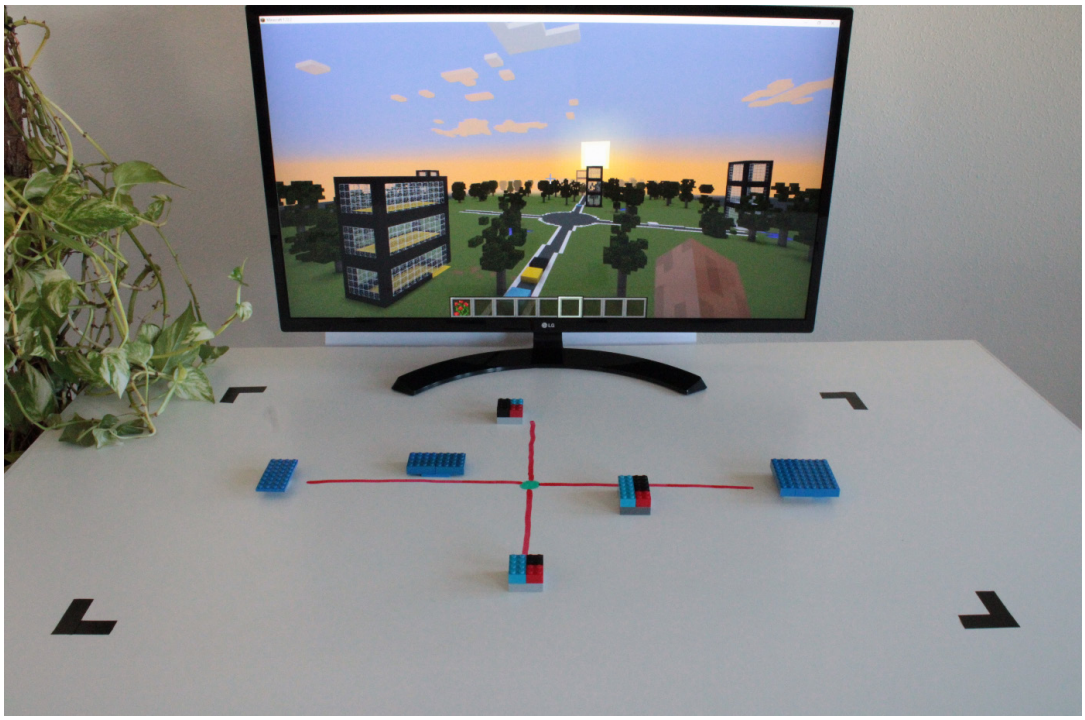


Figura 4: Muestra de un caso de estudio sencillo en el que intervienen los elementos y bloques enumerados. Sobre la mesa se ven varios patrones de colores situados en las inmediaciones de un cruce de cuatro calles. En la pantalla, se muestra el escenario virtual recreado en Minecraft con los edificios y carreteras detectados.

aportar resultados clave como gráficas que muestren el comportamiento del modelo de forma cuantitativa. Finalmente, pese a estos desarrollos potenciales, se considera que el nivel alcanzado por el producto es lo suficientemente adecuado como para ejecutar estos proyectos de mejora sobre una estructura robusta y funcional.

Bibliografía

[1] MIT Media Lab. URL: https://www.media.mit.edu/.

[2] Aimsun. *Aimsun Live*. 2020. URL: https://www.aimsun.com/aimsun-live/

[3] Intel. *Intel RealSense Product Family D400 Series*. 2020. URL: https://www.intelrealsense.com/wp-content/uploads/2020/05/Intel-RealSense-D400-Series-Datasheet-May-2020.pdf.

[4] Jonathan Hui. *Real-time Object Detection with YOLO, YOLOv2 andnow YOLOv3*. 2018.

[5] OpenCV. *OpenCV*. 2020. URL: https://opencv.org/.

[6] Numina. URL: https://numina.co/.

DESIGN OF INTERACTIVE TABLE FOR MOBILITY ANALYSIS AND MODELING OF URBAN SCENARIOS

Author: Rodrigo Tobías, Ignacio de.

Supervisors: Carnicero López, Alberto y Boal Martín-Larrauri, Jaime.

Collaborating entity: ICAI – Universidad Pontificia Comillas.

Summary of the project

The development of urban areas has accelerated greatly over the last few decades and this trend is expected not only to continue but to increase. Associated with this exponential growth of cities, a problem of mobility in their streets appears. The distances travelled, the time spent or the danger involved in the movement of citizens are parameters that must be taken into account when designing an urban network. This project studies how to provide a product that serves as a blank canvas on which to make city designs in a collaborative and interactive way. Thus, through the use of computer vision techniques, designers can recreate a virtual model by manipulating simple tools, such as Lego bricks to model buildings or markers to draw streets. By generating the virtual model, mobility simulations can be performed to contrast the different proposals. Therefore, the design process of urban spaces becomes more transparent and complete.

Introduction

The project describes the design of a tool —interactive table inspired by previous developments of MIT Media Lab [1]— which aims to favour urban modelling from a multidisciplinary perspective and in which the quality of mobility of each proposed model is evaluated. Thus, a variety of profiles can be included in the decision-making process, so that anyone can contribute their vision thanks to the use of intuitive tools.

On the table there are Lego bricks that model different parts of the city (schools, hospitals, green areas, etc.). In addition, users can draw on the table the routes of the streets. By using computer vision, each of the group of bricks or traces are recognized and information is associated with them. Thanks to this input, a simulation is executed in the Aimsun [2] software to evaluate the quality of the

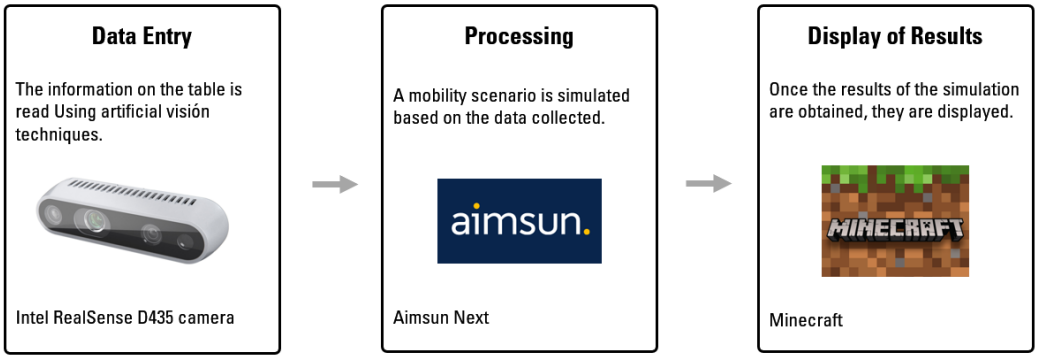


Figure 5: Project blocks together with their operating sequence.

design according to the established mobility flows. Finally, the simulation results are shown to the users in a virtual environment recreated in Minecraft.

Objectives

The project focuses on conceiving a product that helps to design urban areas by evaluating their demand for mobility. In addition, it aims to democratize the design process and facilitate the dissemination of the results achieved. Therefore, the objectives that are established are:

- To provide the user with an intuitive and agile method to locate buildings and trace communication routes.
- Integrate a mobility model that faithfully represents the displacements that are generated from the physical model —otherwise a model of less complexity should be developed—.
- To show the user in a simple and visual way the results achieved by the simulation.
- Integrate all elements in a compact way that favors interaction between users and at the same time between user and machine.

Solution

The model that is implemented is structured in three phases: data entry, processing and display of results. An outline describing the sequence of the three blocks is shown in the Figure 5.

- **Data entry**. In the project, the Intel Realsense D435 camera [3] is used to capture the information of the elements that are arranged on the table

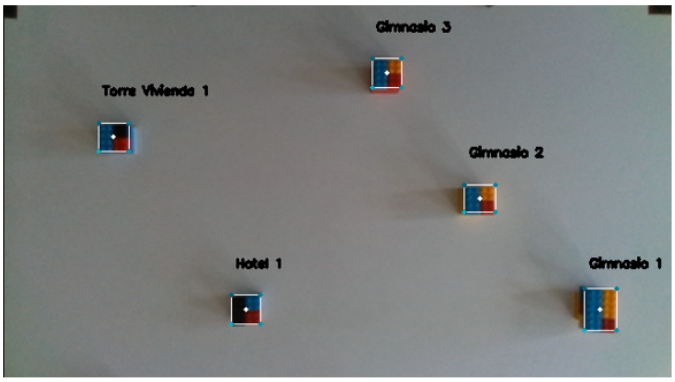


Figure 6: Example of data entry. Detecting different types of patterns using YOLO v3 detection system.

—Lego bricks and marker traces—. With Lego bricks, color patterns can be formed. Based on the patterns included in the model, a total of 19 different urban enclaves such as office towers, hospitals, restaurants or stadiums can be identified. On the other hand, the lines drawn with a marker allow to identify the streets of the urban area to be simulated. Once the images are taken, the data acquisition algorithms are applied. For the detection of coloured parts, YOLO v3 [4] is used, while for the detection of roads, the OpenCV [5] library is used. Once the data is extracted, it is collected in '.xml' files. In Figure 6 an input example for the detection of different types of color patterns is shown.

- **Processing**. From the '.xml' files generated, the information needed to recreate the scenarios is available in the mobility simulation software, Aimsun. Once the scenarios —generated by code— are obtained, the simulation is launched. The results of the simulation show the coordinates and times of the different routes taken. These results are collected again in a '.xml' file.
- **Display of results**. Finally, the results of the simulation are included in a virtual scenario recreated in Minecraft.

As explained in the previous list, a more detailed diagram is shown in Figure 7. The blocks for line detection and pattern detection are associated with the data input described in Figure 5.

Results

The sum of the different blocks described makes it possible to capture and extract data associated with curved lines of complex geometries, to detect and classify colour patterns, to recreate urban scenarios automatically by using code, and to recreate virtual scenarios.

The addition and coordination of these blocks offers as a result a platform to simulate simple urban scenarios. On the one hand, it provides the flexibility and

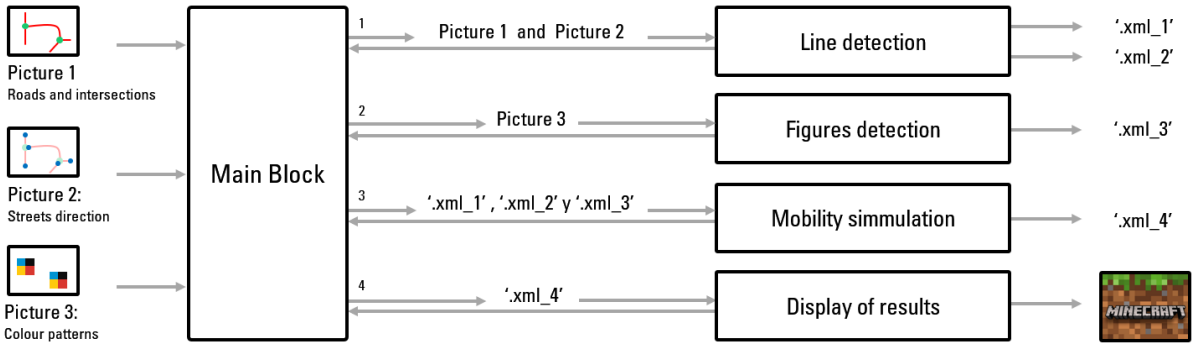


Figure 7: Expanded project blocsks diagram.

ease of editing models through the use of highly intuitive elements, while on the other, it offers the support and reliability of a proven software. That is to say, the product that is generated allows the limitations associated with the use of the software to be overcome, simplifying the process of generating models and opening up the possibility of this process being much more participative. The Figure 8 shows a simple case study in which the physical elements that serve as data input —marker traces and Lego bricks— are shown, and the results of the simulation and the virtual environment are displayed.

Conclusions

As mentioned in the previous section, the results of this work make it possible to use the final product for the generation and study of simple scenarios. For this purpose, the project provides a structure that enables the capture of data from a physical model, allows the communication and execution of simulations in a specialized software, and recreates a virtual scenario.

The level of development of the structure does not reach its full potential in this version of the project, and there are possibilities of greatly improving its performance in aspects such as the association of real data, the contribution of a higher level of detail to the models, or the improvement of the experience of the end user. For example, when modelling displacements with greater detail and reliability, it is desirable to introduce information from databases —Google, city councils, etc.— into the model or to carry out local mobility studies with technology such as that offered by Numina [6]. On the other hand, in terms of improving the user experience, it is possible to enhance the graphics of the virtual environment or provide key results such as graphics that show the behavior of the model in a quantitative manner. Finally, despite these potential developments, it is considered that the level reached by the product is adequate enough to execute these enhancement projects based on a robust and functional structure..

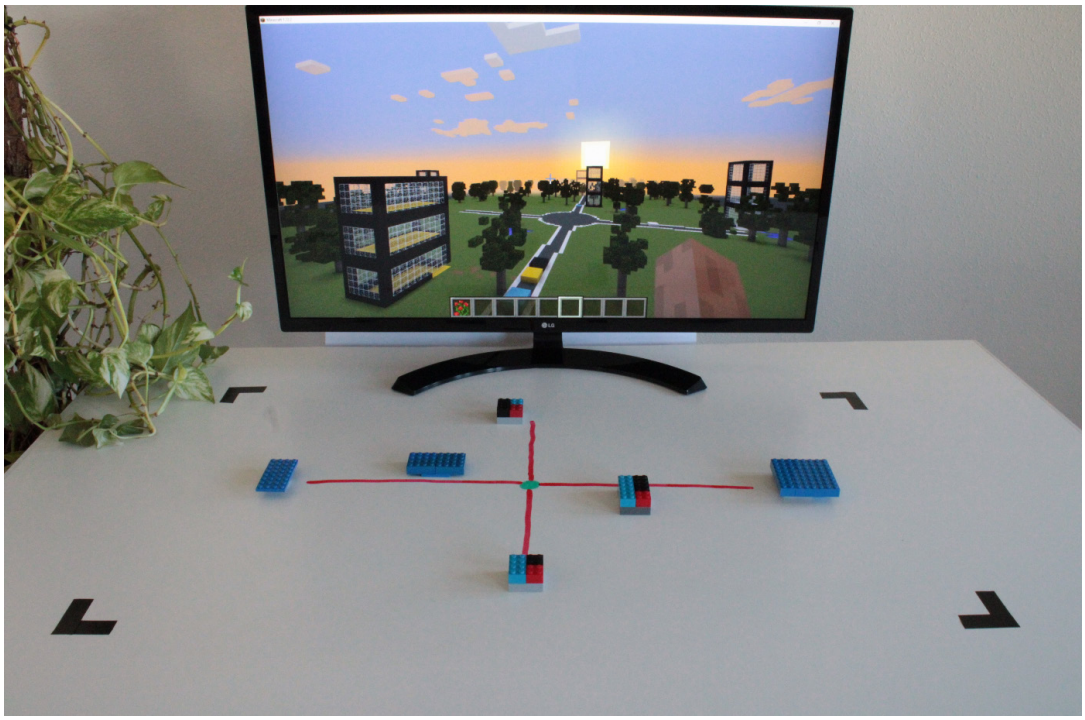


Figure 8: Sample of a simple case study involving the elements and blocks listed. On the table several colored patterns can be seen in the surroundings of a four street intersection. On the screen, the virtual scenario recreated in Minecraft is shown with the detected buildings and roads.

Bibliography

[1] MIT Media Lab. URL: https://www.media.mit.edu/.

[2] Aimsun. *Aimsun Live*. 2020. URL: https://www.aimsun.com/aimsun-live/

[3] Intel. *Intel RealSense Product Family D400 Series*. 2020. URL: https://www.intelrealsense.com/wp-content/uploads/2020/05/Intel-RealSense-D400-Series-Datasheet-May-2020.pdf.

[4] Jonathan Hui. *Real-time Object Detection with YOLO, YOLOv2 andnow YOLOv3*. 2018.

[5] OpenCV. *OpenCV*. 2020. URL: https://opencv.org/.

[6] Numina. URL: https://numina.co/.

Agradecimientos

Gracias a Jaime y a Alberto por su paciencia y sus indicaciones. También por su confianza, especialmente en el inicio del proyecto, cuando este trabajo presentaba una gran cantidad de incógnitas. Espero y deseo en el futuro sigáis apoyando las propuestas que os planteen los alumnos, tal y como lo habéis hecho conmigo.

Gracias al equipo de Aimsun, que siempre ha mostrado su disponibilidad para solucionar los problemas que han surgido durante el desarrollo del proyecto.

Gracias a todos mis amigos por estos seis años de estudios en ICAI.

Gracias a mis padres por priorizar siempre la educación de sus hijos.

Índice general

1. Introducción **29**

2. Estado de la cuestión **35**
2.1. Diseño interactivo 35
2.2. Movilidad urbana 36

3. Arquitectura del sistema **41**
3.1. Comunicaciones entre bloques 41
3.1.1. Detección de líneas 42
3.1.2. Detección de piezas 45
3.1.3. Simulación de movilidad 46
3.1.4. Muestra de resultados 46
3.2. Bloque principal 47

4. Detección de líneas **51**
4.1. Definición de sección e intersección 51
4.2. Especificaciones para una correcta comunicación con Aimsun 53
4.3. Algoritmo. Técnicas para la manipulación de imágenes, extracción de información y su comunicación 54
4.3.1. Secciones: Manipulación de la primera imagen 55
4.3.2. Secciones: Extracción de información de la primera imagen . . 59
4.3.3. Secciones: Manipulación y extracción de datos de la segunda imagen 69
4.3.4. Secciones: Preparación de datos de salida 71
4.3.5. Intersecciones 73

5. Detección de piezas **77**
5.1. Tipos de edificios y enclaves urbanos 77
5.2. Exploración de estrategias 78
5.3. Entrenamiento del modelo para la detección de piezas. YOLO 79
5.3.1. Preparación del conjunto de datos para el entrenamiento del modelo 80
5.3.2. Entrenamiento del modelo en Google Colab 83
5.3.3. Descarga e integración 85
5.4. Seguimiento de piezas 86
5.5. Especificaciones para una correcta comunicación con Aimsun 95
5.6. Observaciones del algoritmo para la detección de piezas 96

6. Simulación de movilidad: Aimsun **101**
6.1. Generación de una capa . . . 103
6.2. Generación de secciones e intersecciones . . . 103
6.3. Generación de centroides . . . 107
6.4. Generación de matriz origen-destino . . . 109
6.5. Generación de demanda . . . 111
6.6. Generación de escenario . . . 112
6.7. Ejecución de simulación . . . 112

7. Visualización de resultados: Minecraft **115**
7.1. Generación del área de trabajo . . . 116
7.2. Generación de carreteras . . . 116
7.3. Generación de edificios . . . 119
7.4. Generación de decorado . . . 120
7.5. Recreación de los desplazamientos de la simulación . . . 123

8. Resultados **125**

9. Conclusiones y futuros desarrollos **129**

A. Objetivos de desarrollo sostenible **133**

B. Diccionario de Términos **135**

Índice de figuras

1. Resumen. Bloques del proyecto 8
2. Resumen. Ejemplo de detección de patrones de colores 9
3. Resumen. Diagrama ampliado 10
4. Resumen. Caso de estudio 11
5. Summary. Project blocks 14
6. Summary. Example of color pattern detection 15
7. Summary. Expanded diagram 16
8. Summary. Case study 17

1.1. Bloques del proyecto 29
1.2. Herramientas de diseño para usuarios 30
1.3. Herramientas activas: piezas de Lego, rotuladores y pantalla 32

2.1. Mesa interactiva: proyecto *Transform* 36
2.2. Mesa interactiva 37
2.3. Urbanismo de París 38
2.4. Urbanismo de Brasilia 39

3.1. Arquitectura del sistema 42
3.2. Diagrama de entrada y salida de datos del bloque Detección de Líneas 43
3.3. Archivo '.xml' con datos de varias secciones 44
3.4. Archivo '.xml' con datos de dos intersecciones 44
3.5. Diagrama de entrada y salida de datos del bloque Detección de piezas 45
3.6. Archivo '.xml' con datos de patrones detectados 46
3.7. Diagrama de entrada y salida de datos del bloque Simulación 46
3.8. Archivo '.xml' generado en el bloque de simulación 47
3.9. Diagrama de entrada y salida de datos del bloque muestra de datos . 47
3.10. Escenario virtual generado en Minecraft 48
3.11. Selector de fases de diseño 48

4.1. Muestra de líneas detectables 52
4.2. Cómo solventar limitaciones 53
4.3. Diagrama. Elementos en en el bloque de detección de líneas 54
4.4. Caso de trabajo para explicación de algoritmo 55
4.5. Secciones e intersecciones captadas por cámara 57
4.6. Máscara de color aplicada sobre líneas 57
4.7. Contorno de una línea genérica 58
4.8. Contornos representados en Aimsun antes de filtrar datos 59
4.9. División de un contorno en dos trazos útiles 59
4.10. Ordenación de píxeles y ángulos 60

4.11. Contorno formado por píxeles 61
4.12. Secuencia de ángulos en contorno formado por píxeles 62
4.13. Interpretación y corrección de ángulo 63
4.14. Gráfica con registro de datos de un contorno 65
4.15. Relación entre espectro de color para la detección de líneas y valores atípicos de ángulos 66
4.16. Desfase y sensibilidad en la detección de extremos asociados al número de muestras en la media móvil de ángulos 67
4.17. Marcas para indicar el sentido 70
4.18. Resumen gráfico. De contorno de sección a tramo con puntos correctamente ordenados 71
4.19. Carretera representada en Minecraft 72
4.20. Técnica de segmentación para enviar coordenadas a Aimsun 72
4.21. Máscara de color aplicada sobre las intersecciones 74
4.22. Píxeles de intersección localizados y rodeados 74
4.23. Representación gráfica para el registro de secciones entrantes y salientes de una intersección 75

5.1. Piezas de Lego de color azul en diferentes tonalidades 79
5.2. Patrones de colores asociados a cada edificio o espacio urbano 81
5.3. Número de imágenes de entrenamiento por cada objeto que se desee localizar 81
5.4. Reducción del número de imágenes 82
5.5. Plantilla de posiciones de patrones para la obtención del conjunto de imágenes de entrenamiento 82
5.6. Imagen genérica para la formación del conjunto de datos de entrenamiento 83
5.7. Imagen genérica con los patrones etiquetados 84
5.8. Imágenes generadas con el modelo de detección de piezas 87
5.9. Seguimiento de patrones fotograma a fotograma 88
5.10. Casos para la actualización del estado de los patrones 89
5.11. Seguimiento de patrones: fotograma sin patrón y sin registro de IDs previo 90
5.12. Seguimiento de patrones: fotograma sin patrón pero con registro de IDs previo 90
5.13. Seguimiento de patrones: fotograma con patrón pero sin registro de IDs previo 91
5.14. Seguimiento de patrones: fotograma con mismos patrones que en el fotograma previo 91
5.15. Seguimiento de patrones: cálculo de distancias 92
5.16. Seguimiento de patrones: irrupción de un patrón entre otros ya detectados 93
5.17. Seguimiento de patrones: cálculo de distancias y discriminación del nuevo patrón 94
5.18. Coordenadas para definir una detección 95
5.19. Archivo '.xml' con la información de los patrones detectados 96
5.20. Patrones de diferentes tamaños 97
5.21. Distorsiones aceptadas de patrón aceptadas por el modelo 98

5.22. Error en el seguimiento debido a un número bajo de FPS 99

6.1. Archivos de entrada y salida de Aimsun 101
6.2. Centroide y secciones 102
6.3. Conexiones de secciones 105
6.4. Corrección de coordenadas verticales en Aimsun 106
6.5. Amplificación de coordenadas en Aimsun 106
6.6. Reducción de la longitud de las secciones 106
6.7. Generación de intersecciones en Aimsun 107
6.8. Gráficos de Aimsun: secciones y centroides 110

7.1. Puntos que definen un volumen para el comando *setBlocks* 117
7.2. Área de trabajo definida en Minecraft 117
7.3. Generación de carreteras en Aimsun 118
7.4. Entramado de calles en Minecraft 120
7.5. Edificios generados en Minecraft 121
7.6. Árboles generados en Minecraft 121
7.7. Técnica para generar estanques y jardines de flores 122
7.8. Elementos del escenario 122
7.9. Recreación de desplazamientos 123

8.1. Resultados. Elementos de trabajo 125
8.2. Resultados. Definición de calles 126
8.3. Resultados. Selección de fases de diseño 127
8.4. Resultados. Definición de enclaves urbanos 127
8.5. Resultados. Modelo completo 127
8.6. Resultados. Visualización de resultados 128

9.1. Datos de Google. Diagrama de concurrencia para un establecimiento . 131

B.1. Fragmento de archivo '.xml' 136

Fragmentos de código

1. Comandos para ejecutar un *script* de Aimsun. 49
2. Comandos para gestionar las ventana que se muestra por pantalla. . . 49
3. Conversión de imagen a espacio de color HSV. 56
4. Aplicación de máscara de color. 56
5. Extracción de las coordenadas de los contornos. 57
6. Comandos para la comunicación Python - Excel. Importación de librerías. 67
7. Comandos para la comunicación Python - Excel. Activación de hoja Excel y acceso a celda. 68
8. Comandos para la comunicación Python - Excel. Elaboración de un gráfico. 68
9. Comandos para la comunicación Python - Excel. Cómo guardar el documento. 69
10. Comandos para la creación de un archivo 'xml'. 73
11. Comando que permite rodear un contorno con una circunferencia de radio mínimo. 74
12. Comandos para cargar el modelo entrenado. 86
13. Instanciación del algoritmo de seguimiento de patrones 94
14. Comando para actualizar la información fotograma a fotograma . . . 94
15. Comandos para extraer las coordenadas de las cuatro esquinas de un rectángulo 95
16. Aimsun. Comandos para la generación de una capa en el modelo. . . 103
17. Aimsun. Comandos para la generación de una sección. 104
18. Aimsun. Comandos para la generación de un giro. 105
19. Aimsun. Comandos para la generación de un área de trabajo. . . . 108
20. Aimsun. Comandos para la generación de la red del problema. . . . 108
21. Aimsun. Comandos para la generación de una configuración de centroides. 109
22. Aimsun. Comandos para la generación de centroides. 109
23. Aimsun. Comandos para la generación de un matriz origen-destino. . 111
24. Aimsun. Comandos para la selección de un vehículo predefinido. . . 111
25. Aimsun. Comandos para la generación de un horario. 112
26. Aimsun. Comandos para la generación de un escenario, un experimento y una replicación. 113
27. Aimsun. Comandos para la ejecución de una simulación. 113
28. Minecraft. Comandos para la generación de una instancia que permite comunicar el código de Python con Minecraft. 115
29. Minecraft. Comandos para la generación de un área de trabajo. . . . 117

30. Minecraft. Comandos para la generación de un cuadrilátero o sección de carretera. 119
31. Minecraft. Comandos para la generación de una línea de bloques. . . 119
32. Minecraft. Comandos para la generación de una copa de árbol —esfera—. 121

Capítulo 1

Introducción

En este proyecto se persigue el diseño de una mesa interactiva que pretende, mediante la manipulación de elementos y herramientas sencillas e intuitivas, reducir los límites que pueden aparecer en la manipulación de *software* complejo. De este modo, la herramienta que se propone en este proyecto puede encontrar usos potenciales en contextos como la pedagogía, la democratización de tomas de decisiones, o la divulgación.

Dentro del amplio campo de aplicación de esta herramienta, la disciplina con la que se trabaja para su desarrollo y muestra de resultados es la movilidad urbana. Concretamente, con el proyecto se pretende aportar una plataforma colaborativa para el diseño y análisis de escenarios de movilidad dentro de núcleos urbanos, acercando a personas con diferentes perfiles: urbanistas, sociólogos o ciudadanos a la toma de decisiones y al diálogo sobre el diseño de las ciudades del futuro.

Para conseguir un producto funcional y que pueda aportar al usuario una capacidad y libertad de decisión suficiente, se recurre a la visión artificial para el reconocimiento de los elementos que se disponen y dibujan sobre la mesa, se incluye un modelo de movilidad para procesar la información recogida, y se comunican los resultados de cada propuesta mediante un entorno visual —Figura 1.1—.



Figura 1.1: Bloques del proyecto junto con la secuencia de funcionamiento de los mismos.



Figura 1.2: Herramientas con las que cuentan los usuarios para desarrollar sus diseños: mesa, rotuladores, piezas de Lego y pantalla (también aparece en imagen la cámara utilizada en el proyecto).

Por último, de forma paralela al diseño del *software* asociado al proyecto, se diseña el módulo físico —mesa— que integra todos los elementos necesarios —cámara, iluminación, superficies de trabajo, etc.—. Este diseño atiende no sólo a cuestiones técnicas impuestas por el *hardware* utilizado, sino que se pretende prestar especial atención a aquellos factores que permitan y favorezcan la interacción y el diálogo de los usuarios, aportando así coherencia al proyecto.

De este modo, se intenta simplificar al máximo tanto la complejidad de los elementos con los que pueden interactuar los usuarios como los procesos o acciones que deben ejecutar para comunicarse con el sistema. Esto da como resultado la elección de unas herramientas que se consideran mínimas para la elaboración de diseños y que se exponen en la Figura 1.2: una mesa de superficie impermeable y de color blanco —similar a una pizarra—, rotuladores de pizarra blanca, piezas de Lego, y una pantalla o proyector (además de la cámara, aunque este no sea un elemento con el que interactúen los diseñadores, sino un medio para ello).

Si se analizan estas herramientas, se puede adivinar que tanto los rotuladores como las piezas de Lego o la pantalla son elementos activos que ofrecen más libertad para el diseño que la mesa —puramente pasiva—. La misión que cumple cada una de estos elementos activos se describe a continuación.

- **Piezas de Lego**: permiten a los usuarios modelar un total de 19 tipos diferentes de edificios o espacios urbanos como torres de viviendas, espacios verdes, o paradas de autobuses. Para modelar los edificios, se definen códigos o patrones de colores. Estos patrones ya están definidos en el programa, por lo que el usuario tan solo debe reproducirlos agrupando las piezas de la manera correspondiente tantas veces como quiera. Para facilitar esta labor

a los usuarios, por la pantalla se muestran los códigos de colores que pueden implementarse y el edificio o espacio urbano al que equivalen. Además, las piezas de Lego pueden utilizarse en otras funciones, como la de incorporar información adicional al modelo —como por ejemplo la utilización de piezas azules 1x1 para indicar sentidos de calles—.

- **Rotuladores**: Los rotuladores suponen una herramienta fundamental, pues posibilitan la introducción al modelo de las calles, carreteras o cruces de la ciudad. La elección de los rotuladores como herramienta para incorporar información al modelo viene motivada por su fácil utilización por parte de los usuarios, que pueden trazar tantas líneas como deseen y editarlas de manera rápida y sencilla. Además, la posibilidad de incluir diferentes colores supone una gran ventaja, pues puede facilitar enormemente la introducción de datos al modelo. Un ejemplo de esta aplicación aparece a la hora de modelar las intersecciones o cruces. El usuario debe representarlos mediante una circunferencia sólida de color verde que contacta con las calles que, por otro lado, son representadas mediante líneas continuas de color rojo.

- **Pantalla**: este elemento permite al sistema comunicarse con los usuarios y aportarles información útil para el diseño. Por un lado, los usuarios pueden recibir información sobre cómo comunicarse con el sistema. Por ejemplo, durante el proceso de disposición de las piezas de Lego, la pantalla muestra todos los patrones de colores y el edificio al que equivalen, mientras que durante la fase de trazado de líneas, la pantalla muestra para qué se debe utilizar cada color de rotulador y un ejemplo en bucle que permite a los usuarios entender fácilmente la mecánica de diseño. Por otro lado, en la última fase, en la que ya se han leído todos los datos de entrada, los resultados de la simulación se muestran por pantalla utilizando los gráficos que proporciona Minecraft, en cuyo entorno se habrá dibujado automáticamente el escenario urbano que haya sido representado sobre la mesa —calles, intersecciones, edificios, etc.—.

Un ejemplo sencillo de cómo trabajan estos elementos en conjunto se muestra en la Figura 1.3, donde en la mesa aparecen cuatro calles que se intersecan en el centro. A su alrededor, se disponen los edificios modelados con diferentes patrones de colores. La pantalla muestra una vista aérea del entorno virtual creado en Minecraft, donde se refleja y amplía la información recogida.

Además de los elementos que se han mencionado hasta ahora —que constituyen los elementos físicos con los que interactúan los usuarios—, existen otros componentes que pueden no ser percibidos de manera igualmente explícita, pero que son fundamentales dentro del proyecto. Algunos de ellos, como la cámara o el videojuego Minecraft, ya han sido mencionados. A continuación se aporta una breve descripción de todos ellos.

- **Cámara**: se utiliza una cámara Intel RealSense D435 [Int20] por la experiencia previa adquirida en otros proyectos desarrollados también en la Universidad. Ofrece una calidad de imagen adecuada para realizar las operaciones necesarias y funciones que enriquecen la captación de datos, como la indicación de la profundidad de los elementos que entran en escena.

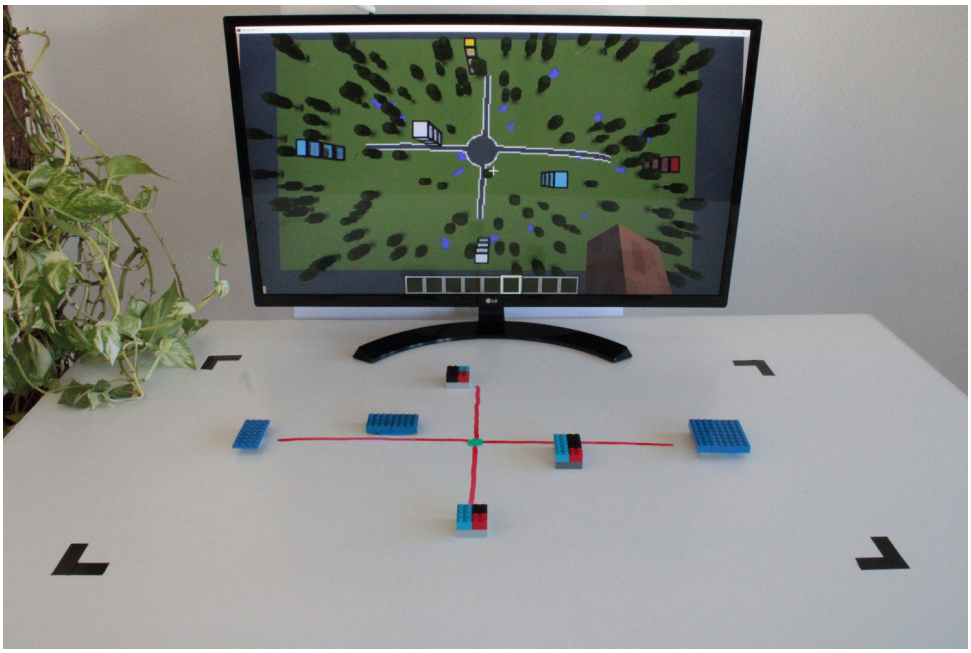


Figura 1.3: Ejemplo en el que intervienen las herramientas activas para modelar un escenario urbano sencillo

- **Python**: se elige Python como lenguaje de programación por su flexibilidad: permite utilizar librerías como OpenCV que resultan cruciales para el desarrollo del proyecto, está suficientemente extendido en la industria como para encontrar e incorporar de manera sencilla *software* externo (programa para simulación de movilidad e interfaz para exposición de resultados), y además, resulta sencillo acceder a documentación para realizar consultas y solucionar posibles problemas.

- **Aimsun Next**: se elige como *software* de simulación de movilidad por su experiencia y reconocimiento dentro de este sector. Además, la capacidad de integrar Python de manera directa, su cercanía al mundo académico y la posibilidad de ofrecer soporte al proyecto mediante un contacto fluido, representan una oportunidad de mejora notable.

- **Minecraft**: Minecraft ofrece un lienzo en blanco sobre el que poder dibujar cualquier escenario de manera divertida y visualmente agradable gracias a las posibilidades que ofrece en sus últimas versiones.

- **YOLO**: es un sistema de detección de objetos que permite trabajar con una velocidad de imágenes por segundo más alta de la que permiten otras técnicas de reconocimiento. La abreviatura YOLO hace referencia a la técnica que utiliza: *you only look once*. La rapidez con la que detecta los elementos dentro de las imágenes hace que este sistema sea compatible con proyectos que requieren trabajar en tiempo real [Hui18]. Esta técnica ha sido utilizada para el reconocimiento de los patrones que forman las piezas de Lego —que representan los edificios— y su aplicación puede extenderse para introducir información adicional de manera sencilla al modelo.

Cada uno de los elementos nombrados en los párrafos anteriores son cruciales para desarrollar los diferentes bloques del proyecto. En el Capítulo 3, se desgrana

cada uno de ellos, explicándose con mayor detalle cuáles son sus funciones, cómo se integran con el resto de bloques y cuáles son las estrategias que se han seguido para traducir los trazos y bloques sobre la mesa en datos que pueden representar la movilidad de una ciudad.

Finalmente, en relación a los objetivos, el proyecto se enfoca en concebir un producto que ayude a diseñar enclaves urbanos evaluando su demanda de movilidad. Además, se pretende conseguir democratizar el proceso de diseño y facilitar la divulgación de los resultados alcanzados. Por tanto, los objetivos que se establecen son:

- Dotar al usuario de un método intuitivo y ágil para poder localizar edificios y trazar vías de comunicación.
- Integrar un modelo de movilidad que represente de manera fiel los desplazamientos que se generan a partir del modelo físico —en su defecto se debería desarrollar un modelo de menor complejidad—.
- Mostrar al usuario de manera sencilla y visual los resultados alcanzados mediante la simulación.
- Integrar todos los elementos de manera compacta y que favorezca la interacción entre los usuarios y al mismo tiempo entre usuario y máquina.

Capítulo 2

Estado de la cuestión

En los siguientes apartados se trata primero el estado del arte referido a la mesa como herramienta de diseño interactivo, y en segundo lugar, el referido a la movilidad urbana.

2.1. Diseño interactivo

En el diseño interactivo y colaborativo cabe destacar como uno de los máximos referentes al MIT Media Lab del Instituto de Tecnología de Massachusetts [Lab]. Sus proyectos de investigación abarcan un amplio abanico de aplicaciones; desde la divulgación de conocimiento y tecnología entre los más jóvenes -dentro del grupo *Lifelong Kindergarten*-, hasta la utilización de realidad virtual controlada mediante gestos -dentro del grupo *Object-Based Media*-.

Dentro del MIT Media Lab, en relación a la utilización de mesas interactivas destacan dos instrumentos fundamentales -de las que derivan gran cantidad de estudios y casos de aplicación-.

La primera de ellas basa la interacción con el usuario en la utilización de pines que se trasladan verticalmente. Los pines de la mesa se accionan mediante un sistema neumático y la muestra de resultados se refuerza en ocasiones mediante la proyección de colores sobre los pines activados. Este tipo de mesa interactiva —Figura 2.1— permite al usuario transformar el espacio y mostrar figuras, superficies o recrear movimientos mediante la manipulación física [Med14]. Los últimos desarrollos de esta plataforma, además de la de muestra de figuras, contemplan su uso potencial como simulador para entrenamiento médico, la exposición física de datos o la representación de prototipos.

Por otro lado, como elemento fundamental del que nace este proyecto, cabe destacar la segunda de las mesas interactivas utilizadas por el MIT Media Lab, especialmente dentro de su sección *City Science* —Figura 2.2—. Este instrumento, utilizado en otros centros de innovación como en la HafenCity Universität Hamburg de Alemania [Ham16] o el CityScope de Andorra [And16], muestra diferentes versiones adaptadas al entorno y características propias del estudio que se pretende realizar -enfocado siempre, para las instituciones mencionadas, al análisis de



Figura 2.1: Mesa interactiva de superficie dinámica propuesta por el Tangible Media Group del MIT Media Lab. Fotograma tomado del vídeo de presentación del proyecto [Med14].

parámetros urbanos y comportamientos de la ciudadanía en campos como la movilidad, el acceso a la vivienda, etc.-. En líneas generales, este tipo de mesa interactiva utiliza visión artificial para detectar los elementos que los usuarios disponen sobre su superficie —piezas de Lego, figuras, etc.— y que sirven como datos de entrada para el sistema. La información recogida es procesada, y el usuario percibe los resultados de la simulación mediante pantallas en las que se visualizan datos o mediante proyecciones sobre las piezas y la propia superficie de la mesa, pudiendo incluir mapas de color estáticos o animaciones —congestión de calles, distinción de zonas en función del consumo energético, etc.—. Desde el punto de vista constructivo, este tipo de mesas interactivas incorporan la cámara debajo de la mesa, por lo que exige que la superficie o tablero sobre el que se depositan los elementos sea transparente para poder identificarlos. Por otro lado, cabe reseñar que el uso de proyector determina las condiciones de luminosidad del ambiente en el que se instala la mesa.

En comparación a los diseño mencionados, este proyecto pretende integrar algunas de sus características, como la posibilidad de modelar enclaves urbanos a partir de códigos de colores, o la recreación y visualización de un entorno virtual. Por otro lado, mediante este proyecto se pretende dar un paso más a la hora de aportar flexibilidad a la mesa de diseño permitiendo generar entramados urbanos desde cero mediante el uso de una herramienta sencilla como un rotulador.

2.2. Movilidad urbana

El ser humano, como entidad social y como individuo dependiente de otros —a la hora de obtener recursos tan dispares como la educación o el alimento—,



Figura 2.2: Mesa interactiva para el diseño y análisis urbano propuesta por el City Science Group del MIT Media Lab [Lab].

ha encontrado en las ciudades una forma eficiente —en mayor o menor grado— de establecerse, cooperar y prosperar. Los núcleos urbanos se han ordenado de tal forma que la oferta de bienes y servicios se concentran en localizaciones muy concretas —como un parque de bomberos— o diseminadas por la ciudad —como cafeterías y supermercados— o en grandes extensiones de terreno —como zonas residenciales o bloques de viviendas—. Toda esta estructuración atiende a factores como el radio de acción, la demanda, o el coste del suelo; pero también está condicionado por cuestiones geográficas y climáticas -ríos, elevaciones de terreno, meteorología, etc.-, o por la propia legislación asociada al territorio. Este conjunto de restricciones condicionan al ciudadano y le conducen a tener que desplazarse —o a que otros individuos lo hagan por él— para poder conseguir los bienes y servicios que oferta la ciudad. Es aquí donde surge la motivación de estudiar la movilidad urbana y comprender la necesidad de un equilibrio entre la localización de sus enclaves y los distintos modos que el usuario tiene a su alcance para desplazarse entre ellos.

A lo largo de la historia reciente, y dejando de lado los primeros asentamientos humanos, han sido varias las teorías urbanísticas que se han sucedido proponiendo soluciones a los problemas de movilidad urbana. En muchas ocasiones, estas medidas han estado íntimamente asociadas a cuestiones de salud y temática social. Una muestra de ello puede apreciarse en el desarrollo de algunas ciudades europeas, como es el caso de Londres o Lisboa. En esta última, a raíz del devastador terremoto de 1755 se reconstruyó el laberíntico entramado urbano teniéndose como directriz fundamental la libre circulación de ciudadanos y mercancías de manera rápida y segura. Para ello, las calles fueron ensanchadas y reordenadas. Estas medidas, de manera sinérgica, favorecieron de igual modo los flujos de renovación de aire, y mejoró la exposición de la ciudad y sus edificios al sol; elevando consecuentemente la calidad de vida de los ciudadanos [Mul92]. París es otra ciudad que a lo largo de los últimos siglos ha sufrido diversas y notables modificaciones en su disposición



Figura 2.3: París. Plaza de Charles de Gaulle, punto de confluencia de amplias avenidas ejecutadas en el s.XIX

urbanística. Aquellas que datan del siglo XIX trataron de aportar grandes avenidas que permitiesen el acceso al centro de la ciudad y conectara éste con las estaciones de ferrocarril [Vic11] —Figura 2.3—, mientras que, después de las Guerras, se construyeron vías de circunvalación en su periferia, atendiendo al creciente uso del automóvil de la sociedad francesa.

Posteriormente, en el año 1925, Le Corbusier, considerado como uno de los mayores genios de la arquitectura y urbanismo del siglo XX, propuso para París el *Plan Voisin* [Jea24], con el que sugería reconstruir determinados distritos de la margen derecha del Sena para construir edificios con planta en cruz y amplias avenidas en las que separaba el espacio utilizado por peatones de aquel destinado a automóviles. De estos conceptos, mezclados con los de otros autores, nacería posteriormente la llamada *Carta de Atenas* [Gir16], que abogaba por principios como la seguridad del peatón ante las elevadas velocidades de los nuevos medios de transporte -automóviles, camiones, etc.-; o la necesidad de tranquilidad y silencio en las zonas residenciales. Sin embargo, estos conceptos llegaron necesariamente acompañados de otros que anteponían el vehículo frente al peatón: *Los cruces actuales, situados a 100, 50, 20 o incluso 10 metros de distancia los unos de los otros, no resultan convenientes para la buena marcha de los vehículos mecánicos. Deberían estar separados por espacios de 200 a 400 metros —Carta de Atenas*, 1933, punto 54—. Estas ideas influyeron notablemente en el planteamiento de ciudades como Brasilia —Figura 2.4—, poniéndose de manifiesto una nueva entidad de medida para dimensionar las ciudades: el automóvil.

Tras varias décadas en las que el automóvil se ha elevado como el elemento protagonista dentro de las ciudades, impulsado por factores económicos, sociales y también, como se ha descrito, urbanísticos, se han podido experimentar las consecuencias menos agradables de su hegemonía: contaminación, colapsos, siniestralidad... Ante este escenario, imprevisto o despreciado en la concepción del modelo, han aparecido restricciones al automóvil en un notable número de ciudades



Figura 2.4: Brasilia. Eje Monumental de la capital de Brasil, con vías de seis carriles para cada sentido.

europeas —Madrid, Barcelona, París, Roma, Milán, etc.—. Por otro lado, gracias a la generación de una mayor consciencia social -aunque, por supuesto, también guiado por factores económicos-, ha aparecido un gran abanico de novedosos modos de movilidad encabezados por el *carsharing* -con sus diferentes posibilidades- y la movilidad eléctrica -patinetes, motos, etc.- que se unen a otros modos más tradicionales como el metro, el tranvía, la bicicleta, y por supuesto, el desplazamiento a pie. Frente a este escenario de incertidumbre y competencia entre los distintos modos, enmarcados en un urbanismo que no se adapta a sus necesidades -y que también debe ser formulado una vez más-, diferentes entidades recuerdan cuáles deben ser los focos de atención durante el proceso de transición hacia la movilidad del futuro: el acceso universal, la eficiencia, la sostenibilidad y la seguridad [all17].

Capítulo 3

Arquitectura del sistema

En este capítulo, en el que se introduce la arquitectura del sistema, se pretende dar una descripción general de los bloques que lo componen, los datos de entrada y de salida que necesitan y ofrecen cada uno de ellos, y la comunicación que se establece para que el sistema funcione de manera correcta.

El *software* del proyecto ha sido estructurado en cinco bloques diferentes. Cada uno de estos bloques está relacionado con partes o funciones del proyecto bien diferenciadas:

- **Programa principal**: Permite una correcta coordinación entre usuario y resto de bloques.
- **Detección de piezas**: En este apartado se detectan los patrones de colores formados con las piezas de Lego. Estos patrones modelan los diferentes edificios. Se hace uso especialmente de la librería OpenCV y del sistema de detección de objetos YOLO v3.
- **Detección de líneas**: En este bloque se detectan las líneas e intersecciones que modelan el entramado urbano. Para ello se hace uso de la librería OpenCV y de algoritmos para la extracción de datos a partir de polilíneas.
- **Simulación de movilidad**: En esta sección se definen los elementos propios Aimsun: desde la creación de los escenarios, hasta la ejecución de la simulación de manera automática.
- **Muestra de resultados**: En este último bloque se recopilan los datos de la simulación y se recrean dentro del videojuego Minecraft.

En la Figura 3.1 se expone de forma visual la integración de los bloques que se amplían y explican a lo largo de sus respectivos capítulos.

3.1. Comunicaciones entre bloques

Como se puede extraer de la Figura 3.1, los diferentes bloques del sistema tienen como entrada o salida archivos de datos que contienen sus resultados o los

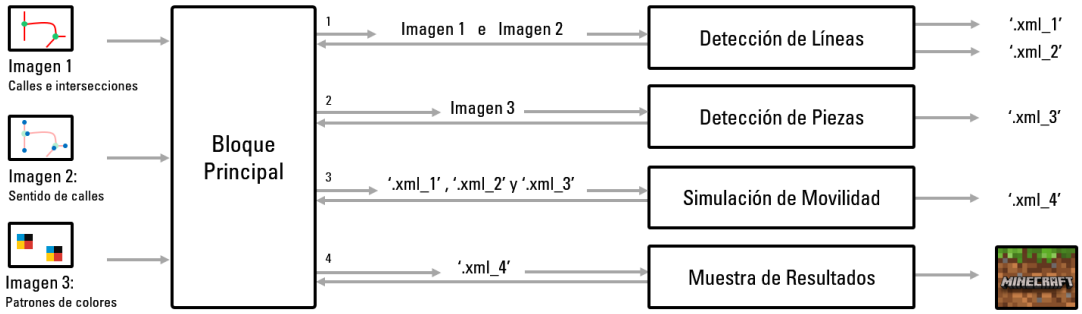


Figura 3.1: Arquitectura interna de los bloques del sistema. El bloque de simulación necesita comunicarse a través de archivos externos.

de bloques anteriores. Esto se debe a que el bloque de simulación de Aimsun se ejecuta en un proceso paralelo o externo al hilo principal del sistema.

Para lograr comunicar el resto de bloques dentro del proyecto se puede hacer uso de librerías o módulos que permiten una integración sencilla con los demás. Este es el ejemplo de Minecraft, que cuenta con una *API* en Python (definición de *API* en Apéndice B), o del módulo de reconocimiento de patrones, que utiliza un modelo entrenado externamente, pero que se integra —recibiendo y compartiendo información— de manera sencilla dentro del sistema.

Sin embargo, en el caso de Aimsun, pese a que dispone de su propia API con Python —y se hace uso de ella, tal y como se explica en el Capítulo 6—, la integración no es tan directa y se requiere que tanto los datos de entrada como los de salida de este programa se recopilen en archivos '.xml' (definición de *archivo '.xml'* en Apéndice B).

En las siguiente secciones se pretende dar una imagen ampliada de los flujos de información que se establecen entre los bloques del sistema. A modo de introducción, y con el objetivo de recrear una imagen mental en el lector, cabe mencionar que la comunicación es secuencial, por lo que la transferencia de datos mantiene una fuerte correlación con la secuencia de fases que siguen los usuarios para diseñar.

3.1.1. Detección de líneas

Los datos que recibe el bloque de detección de líneas, como se muestra en la Figura 3.2, son imágenes. Como se explica en el Capítulo 4, es necesario que este bloque reciba dos imágenes, una por cada subfase de diseño que se finalice. Estas subfases son:

- **Subfase 1:** Dibujo de calles e intersecciones.
- **Subfase 2:** Indicación de los sentidos de las calles.

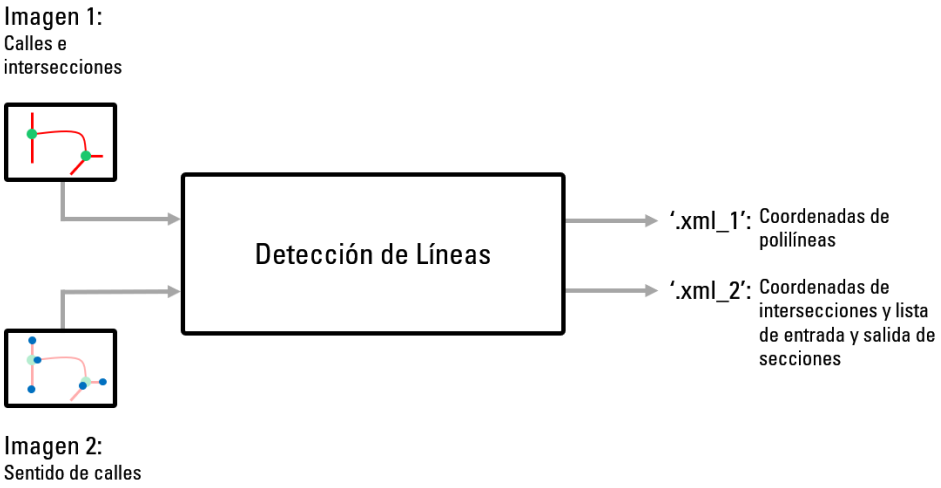


Figura 3.2: En el bloque detección de líneas entran dos imágenes para detectar toda la información necesaria de las secciones e intersecciones y se generan dos archivos de datos '.xml', uno para las secciones y otro para las intersecciones.

En primer lugar, se debe recibir una imagen de la mesa en la que se recojan todas las calles e intersecciones. En el momento en el que los usuarios han terminado de dibujar el entramado urbano y dan la señal de cambio de subfase de diseño, el Bloque Principal comparte la imagen tomada por la cámara con el bloque de Detección de Líneas.

Inmediatamente después, comienza la segunda subfase, en la que los usuarios deben determinar el sentido de las calles. La información que se debe recibir en este punto es fundamental, puesto que una mala ejecución del diseño por parte de los usuarios, conduce a que el modelo no incluirá parte del entramado urbano —por ejemplo, no indicar el sentido de una calle, o no asociarlo de forma correcta, implicará que dicha calle no aparezca en el modelo—. De nuevo, la imagen se envía desde el Bloque Principal hasta el bloque de Detección de Líneas cuando los usuarios indican que han acabado esta subfase de diseño.

Cabe reseñar, de cara a futuros desarrollos en los que se incorpore más información del entramado urbano —como semáforos, número de carriles de cada sección, etc.— que es la segunda subfase (y por tanto en la segunda imagen) donde se recomienda incluir los nuevos datos. Es muy importante poder garantizar la limpieza de la primera imagen, puesto que la forma de extraer información de ella es a través de máscaras de colores —rojas y verdes exclusivamente—. Incluir más información en esta subfase podría distorsionar la lectura de datos —si se incluyen elementos, dibujos u objetos que se superpongan al trazado de las líneas—. En en el momento en el que se toma segunda imagen, la información de las líneas ya ha sido registrado, por lo que cualquier tipo de superposición o distorsión de esa capa de dibujo es ya irrelevante. Esta es la razón por la que se hace uso de una segunda subfase de diseño y por la que se recomienda incorporar información extra en la misma.

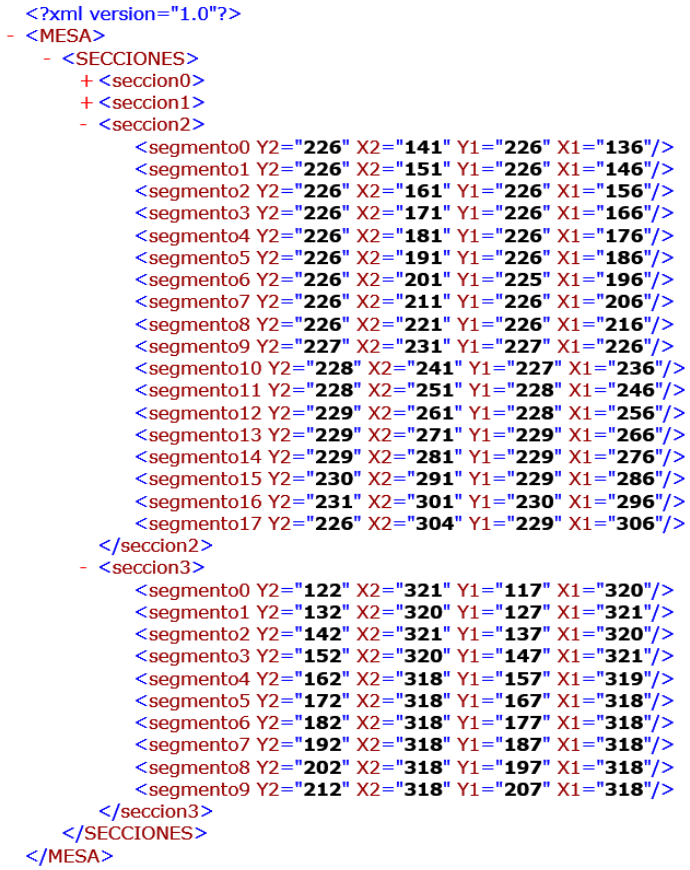


Figura 3.3: Los archivos '.xml' que se generan para guardar y comunicar información de las líneas dibujadas sobre la mesa contienen coordenadas de polilíneas. Estas polilíneas permiten recrear en Aimsun las líneas originales dibujadas en la mesa. En este caso, se han registrado cuatro líneas, pero se muestran los datos únicamente de las dos últimas.



Figura 3.4: Los archivos '.xml' que se generan para guardar y comunicar información de las intersecciones contienen las coordenadas del centro de dicha intersección. Además para cada intersección se incluye un listado de las calles —o secciones— que salen y entran a cada intersección.

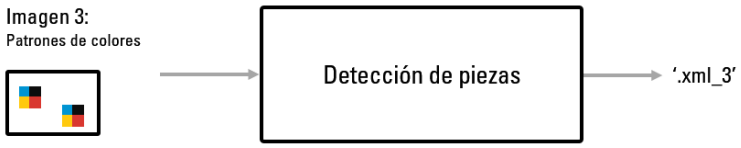


Figura 3.5: En el bloque detección de piezas entra una imagen para detectar toda la información necesaria de los patrones de colores y se genera un archivo de datos '.xml' —Figura 3.8—.

En cuanto a la salida de datos, como ya se ha señalado, se hace uso de un archivo '.xml' para recoger la información y posibilitar que pueda ser leída de forma externa por el *software* de simulación de movilidad. Por comodidad, se opta por escribir un archivo para las líneas —llamadas secciones en Aimsun— y otro para las intersecciones. En el Capítulo 4 se expone cómo se realiza el proceso de escritura de estos archivos. En las figuras 3.3 y 3.4 se muestra un ejemplo de los datos generados para recrear secciones e intersecciones respectivamente. Como se puede apreciar, en el caso de los archivos destinados a secciones —Figura 3.3—, se guardan las coordenadas que definen varios segmentos. Estos segmentos permiten definir una polilínea que recrea cualquier trazo dibujado por los usuarios en la mesa. En el caso de las intersecciones —Figura 3.4—, se guardan las coordenadas del centro de dicha intersección y un listado de las secciones de entrada y salida a la misma.

3.1.2. Detección de piezas

Como se describe en el Capítulo 5, el bloque de detección de piezas permite el seguimiento fotograma a fotograma de los diferentes patrones de colores que se disponen sobre la mesa. Sin embargo, para esta versión del proyecto, sólo se lanza una simulación cuando los usuarios determinan que ha finalizado su proceso de diseño. Es decir, sólo resulta útil —en esta versión— el último fotograma del seguimiento.

En la Figura 3.5 se muestra cómo el bloque dedicado a la detección de piezas tan solo necesita una imagen o fotograma como dato de entrada. En esta imagen de entrada de datos deben aparecer los los patrones de colores. Como el fotograma que recoge la información de los patrones se toma después del proceso de diseño de calles, en esta imagen se suelen superponer la información de ambos procesos de diseño.

Por otro lado, en la Figura 3.5 se observa que el bloque de detección de piezas genera un archivo '.xml'. En este archivo se recoge la información necesaria para implementar el modelo en Aimsun y contiene datos como el número de edificios de cada tipo, sus coordenadas o su área. En la Figura 3.8 se muestra un ejemplo de un archivo '.xml' generado por el bloque de detección de piezas.

```
<?xml version="1.0"?>
- <MESA>
    <SECCIONES/>
  - <CENTROIDES>
    - <Torre_Vivienda>
        <nodo0 Area="783.0" Y="188" X="105"/>
      </Torre_Vivienda>
    - <Hotel>
        <nodo0 Area="784.0" Y="353" X="232"/>
      </Hotel>
      <Supermercado/>
      <Centro_Comercial/>
      <Industrial/>
      <Torre_Oficina/>
      <Colegio/>
      <Biblioteca/>
      <Museo/>
      <Restaurante/>
    - <Gimnasio>
        <nodo0 Area="1240.0" Y="353" X="571"/>
        <nodo1 Area="780.0" Y="247" X="456"/>
        <nodo2 Area="784.0" Y="127" X="367"/>
      </Gimnasio>
      <Bar/>
      <Teatro/>
      <Estadio/>
      <Hospital/>
      <Parada_Bus/>
      <Estacion/>
      <Aparcamiento/>
      <Parque/>
    </CENTROIDES>
</MESA>
```

Figura 3.6: En el archivo generado se registra el tipo de edificio detectado, su área y sus coordenadas.

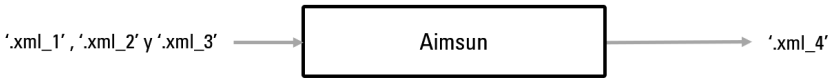


Figura 3.7: En el bloque destinado a la simulación, todos los datos referidos a las secciones, intersecciones y enclaves urbanos se introducen mediante archivos '.xml'. Una vez ejecutada la simulación, se crea otro archivo '.xml' que contiene los datos de los desplazamientos generados.

3.1.3. Simulación de movilidad

El bloque de simulación de movilidad, asociado al *software* Aimsun, recoge los archivos '.xml' mencionados en los apartados anteriores. Como se muestra en la Figura 3.7, recibe tres archivos con los que se pueden recrear, de manera virtual, todos los detalles del escenario dispuesto sobre la mesan. Por otro lado, este bloque genera otro archivo '.xml' en el que se recogen los datos de todos los desplazamientos generados en la simulación.

3.1.4. Muestra de resultados

El bloque de muestra de resultados está asociado al videojuego Minecraft. Como se observa en la Figura 3.9, este bloque sólo recoge los datos de la simulación de Aimsun, es decir, el archivo '.xml' número cuatro. Como salida de datos, a los usuarios del sistema se les ofrece por pantalla una representación de los

```
<FRAME TIME="00:00:16.000">
        <CREATED>
                <VEH ID="10000001">
                        <TYPE>154</TYPE>
                        <WIDTH>1.872514</WIDTH>
                        <HEIGHT>1.700000</HEIGHT>
                        <SHAPE2D length="4.403177">car</SHAPE2D>
                        <SHAPE3D>./shapes/cars/sports_saloon.3ds.3ds</SHAPE3D>
                        <POS X1="1510.920697" Y1="1091.452165" X2="1506.517542" Y2="1091.437961"/>
                        <SPEED>58.594690</SPEED>
                        <ACCELERATION>0.000000</ACCELERATION>
                </VEH>
        </CREATED>
</FRAME>
<FRAME TIME="00:00:16.800">
        <CHANGED>
                <VEH ID="10000001">
                        <POS X1="1523.941671" Y1="1091.494168" X2="1519.538517" Y2="1091.479964"/>
                        <SPEED>58.594690</SPEED>
                        <ACCELERATION>0.000000</ACCELERATION>
                </VEH>
        </CHANGED>
</FRAME>
<FRAME TIME="00:00:17.600">
        <CHANGED>
                <VEH ID="10000001">
                        <POS X1="1536.318792" Y1="1091.534094" X2="1531.915638" Y2="1091.519891"/>
                        <SPEED>52.799983</SPEED>
                        <ACCELERATION>-2.012051</ACCELERATION>
                        <BRAKES/>
                </VEH>
        </CHANGED>
```

Figura 3.8: Archivo '.xml' generado por el bloque de simulación, donde se guardan los datos relacionados con los desplazamientos.

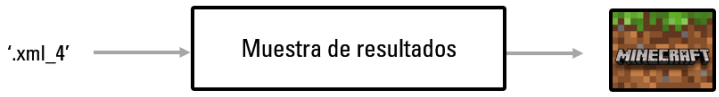


Figura 3.9: El bloque muestra de datos recibe el archivo '.xml' con el registro de la simulación ejecutada. Como salida, este bloque ofrece a los usuarios un escenario en el que se recrean los desplazamientos generados —Figura 3.10—.

desplazamientos en el entorno de Minecraft. Los diferentes vehículos se desplazan por las carreteras dibujadas y cada uno de ellos aparece representado por un grupo de bloques de color uniforme —Figura 3.10—.

3.2. Bloque principal

En el programa principal se incluyen las llamadas a los diferentes bloques que deben ejecutarse para que los usuarios puedan dar forma a sus diseños. Esta llamada atiende a una secuencia preestablecida y que siempre se ejecuta en el mismo orden. En primer lugar se ejecutan las fases de diseño. Son dos: en una primera fase se define el entramado urbano a través de las calles y sus intersecciones, mientras que en la segunda fase se disponen los edificios y enclaves de la ciudad. Una vez se han recogido todos los datos de entrada, desde el programa principal se lanza tanto la simulación de movilidad en Aimsun como la muestra de resultados por Minecraft.

Los usuarios tienen la libertad de emplear el tiempo que deseen en cada una de las dos fases de diseño —trazado de calles y disposición de edificios—, pero el paso de una a otra es irreversible en esta versión del proyecto. Para comunicarse

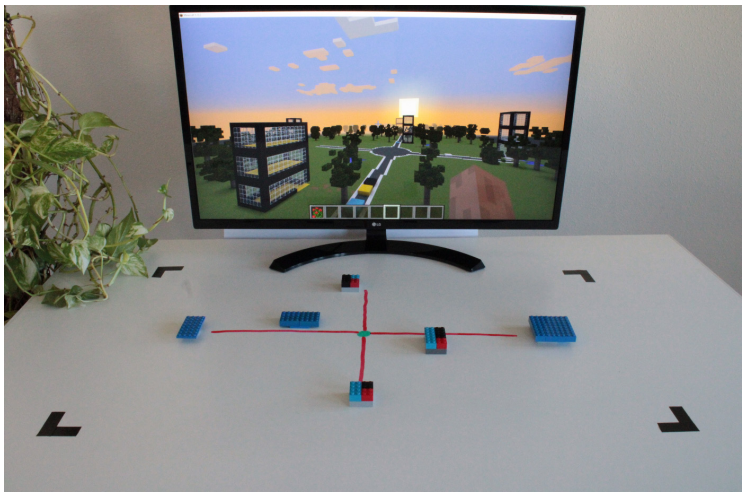


Figura 3.10: En el escenario virtual generado en Minecraft se visualizan los desplazamientos generados en la simulación. Los vehículos se representan con grupos de bloques de diferentes colores.

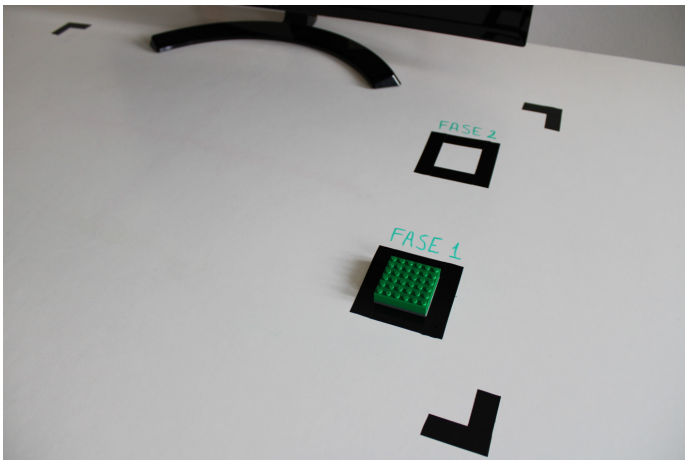


Figura 3.11: Mediante visión artificial se puede extraer información útil para el sistema a partir de objetos sencillos. Gracias a la detección de la posición de la pieza, los usuarios pueden pasar de una fase de diseño a la siguiente. La imagen corresponde a un proceso de pruebas; en el diseño final existen más posiciones.

con el sistema, los usuarios cuentan con una pieza de Lego 6x6 de color verde que se desliza sobre uno de los laterales de la mesa. Sobre este lateral, aparecen marcas que indican a qué fase corresponde cada posición de la pieza verde, que en definitiva funciona como un selector dentro de un menú de opciones. En la Figura 3.11 se muestra de forma gráfica lo explicado en este párrafo.

Una vez finalizada la fase de diseño, desde el programa principal se lanza la simulación de movilidad en Aimsun. Para ello es necesario, mediante el uso de código, abrir Aimsun, ejecutar el *script* de Python que se encuentra dentro de un archivo '.ang' preconfigurado y cerrar de nuevo el programa. En el Fragmento de código 1 se muestran los comandos necesarios para realizar las acciones descritas. En primer lugar, se importan las librerías necesarias. Posteriormente, se ejecuta el comando que permite lanzar la simulación de Aimsun. El primer argumento indica el directorio del programa que se debe ejecutar, en este caso, *Aimsun Next 8.4*.

El tercer argumento indica el archivo '.ang' preconfigurado que debe ejecutarse. La extensión '.ang' hace referencia al tipo de formato utilizado para guardar los proyectos de Aimsun. El último argumento incluye el ID del *script* guardado dentro del archivo '.ang' que debe ejecutarse. El resto de argumentos de este comando deben respetar el orden y la forma mostrados en el Fragmento de código 1. Por último, una vez esperado un tiempo para que en Aimsun se ejecute la simulación, se destruye el proceso generado. Esta acción devuelve el control al programa principal y se continúa con su ejecución. De otra forma, ésta quedaría en suspensión y no avanzaría hasta que no se cerrase manualmente la ventana de Aimsun.

```
import subprocess
import time

p = subprocess.Popen(['D:\\Aimsun Next 8.4\\Aimsun Next.exe',
"--project", 'C:\\directorio\\archivo.ang', "--command",
"--execute", "--target", "525"])

time.sleep(60)
p.kill()
```

Fragmento de código 1: Comandos para ejecutar un *script* de Aimsun.

Finalmente, una vez se ha ejecutado la simulación de Aimsun, desde el programa principal se ordena ejecutar el bloque de muestra de resultados, de tal forma que los usuarios pueden visualizar los resultados de la simulación. Para ello resulta útil mostrar a los usuarios la pantalla de Minecraft. En el Fragmento de código 2 se pueden observar los comandos utilizados para la gestión de las ventanas y el contenido que visualizan los usuarios por pantalla. Estos comandos son útiles para sistemas Windows, mientras que para sistemas Unix el código cambia ligeramente. En primer lugar, es necesario importar la librería asociada al comando. Posteriormente, el primer comando permite buscar una ventana por su nombre —segundo argumento—. Finalmente, en la última línea se muestra cómo maximizar la ventana seleccionada al introducir el segundo argumento: el código 3 implica maximizar, mientras que para minimizar la ventana, se debe incluir el código 6.

```
import ctypes

notepad_handle = ctypes.windll.user32.FindWindowW(
None, "Minecraft 1.12.2")
ctypes.windll.user32.ShowWindow(notepad_handle, 3)
```

Fragmento de código 2: Comandos para gestionar las ventana que se muestra por pantalla.

Capítulo 4

Detección de líneas

En este capítulo, el primero en el que se ahonda en las estrategias seguidas para la extracción de datos a partir de la información de la mesa, se pretende explicar con detalle los pasos que se siguen para definir las calles e intersecciones.

Este bloque es el que puede considerarse como más extenso, no por la complejidad de la lógica que se sigue, sino por el esfuerzo que se requiere realizar para comprender cómo extrae los datos de las imágenes un ordenador —en este caso haciendo uso de la librería OpenCV— y cómo necesita recibirlos en el programa de simulación de movilidad Aimsun.

4.1. Definición de sección e intersección

Para comprender qué es lo que se pretende conseguir en este bloque, se debe entender primero qué se considera una sección y qué una intersección. Por otro lado, también es necesario comprender cuáles son las limitaciones que puedan aparecer en esta versión inicial del proyecto.

- **Intersección**. Una intersección es un punto donde confluyen dos o más secciones. Para una correcta interpretación de los dibujos, las intersecciones deben dibujarse como círculos sólidos de color verde y estar en contacto con los extremos de las líneas o calles que confluyen en dicha intersección.
- **Sección**. Una sección es el nombre que recibe en Aimsun cualquier vía para el paso de automóviles, es decir, puede ser una calle, un acceso o una autopista entre otras opciones. Es importante tener en cuenta que en esta versión del proyecto no se ha integrado un módulo para determinar el número de carriles por sección. Esto requeriría la adquisición de información extra, tal y como se indica en el Capítulo 3 y más adelante en este capítulo también —*4.3.3. Secciones: Manipulación y extracción de datos de la segunda imagen*—. De esta forma, para esta primera versión se ha decidido que cada sección implique un único carril con un único sentido —que debe ser definido por los usuarios—.

Las secciones son dibujadas mediante líneas —en esta versión, de color rojo—, deben tener dos extremos —por lo que quedan excluidos diseños que contengan



Figura 4.1: El algoritmo implementado posibilita detectar una gran variedad de tipos de líneas: rectas con cualquier pendiente, curvadas, trazos sinuosos, etc. En la imagen se muestra una representación en planta de la mesa, donde las escuadras negras delimitan el campo de visión de la cámara.

figuras geométricas cerradas sin extremos—, y deben tener radios de curvatura de al menos dos veces el grosor del trazo de la línea —se explican las razones más adelante en este capítulo—. Pese a estas restricciones, el algoritmo que se implementa es lo suficientemente flexible como para recoger la mayoría de los trazados que pueden encontrarse en las ciudades. En la Figura 4.1 se recoge una muestra de trazados que pueden incluirse sin problema en cualquier diseño. Por otro lado, en la Figura 4.2 se explica cómo pueden superarse las restricciones que se mencionaban anteriormente utilizando el recurso de las intersecciones, es decir, dividir una sección compleja en dos o más secciones sencillas.

- **Rotondas**: Una circunferencia o figura geométrica cerrada no contiene extremos. El algoritmo que se implementa para la detección de las líneas necesita encontrar los extremos de las mismas. Por eso es fundamental que las figuras geométricas cerradas se subdividan en líneas más sencillas. Por ejemplo, a la hora de diseñar una rotonda, la circunferencia puede incluir intersecciones donde confluyen los accesos, de tal forma que la rotanda queda definida por varios arcos de circunferencia.

- **Curvas en horquilla**: Como se explica en las secciones siguientes, el algoritmo necesita detectar los extremos de las líneas. Para ello se sirve de variaciones abruptas del ángulo que forman sus contornos —explicación detallada del algoritmo más adelante—. Un trazo con un radio de curvatura muy reducido puede llevar al algoritmo a detectar falsos positivos. Incorporar una intersección en el ápice de la curva hace que no sea necesario incluirla. De esta forma, la disposición queda simplificada a dos líneas y una intersección.

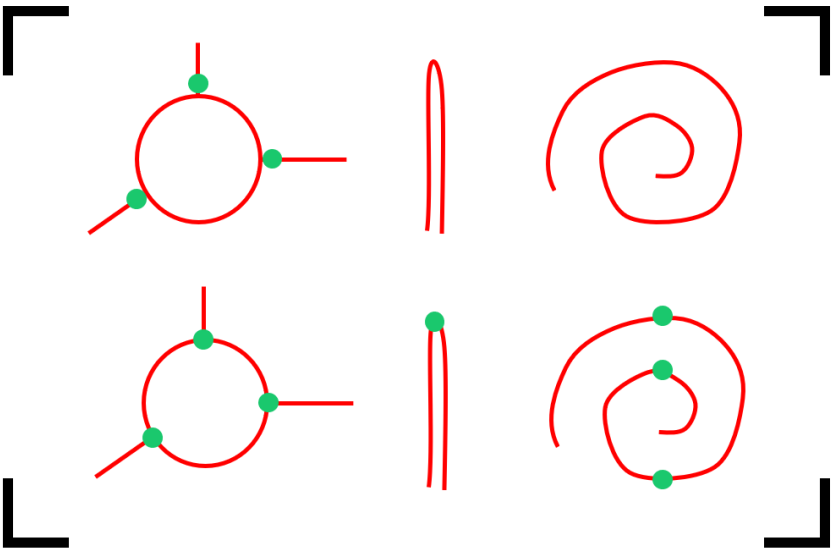


Figura 4.2: Como se explica en la definición de sección, existen ciertas limitaciones que pueden ser solventadas al fragmentar las líneas. En la parte superior de la imagen se muestran disposiciones problemáticas, mientras que en la parte inferior, se aportan soluciones coherentes para cada una de ellas.

- **Espirales**: Aunque suponen una disposición poco habitual, es posible que aparezcan en algún diseño. El problema que aparece con este tipo de curvas viene derivado nuevamente de la lectura de ángulos. Una curva en espiral puede completar varias vueltas, lo que haría trabajar al algoritmo con valores superiores a los 360 grados cuando su rango de detección está limitado de 0 a 360 grados. Como en los casos anteriores, la solución pasa por fragmentar la espiral en varias curvas consecutivas.

4.2. Especificaciones para una correcta comunicación con Aimsun

Como se menciona en el Capítulo 3, en el que se hace referencia a las comunicaciones que se establecen entre los distintos bloques, los datos que comparte el bloque Detección de Líneas con el de Simulación de Movilidad difieren para el tipo de elemento detectado. El tipo de dato que queda registrado en los archivos '.xml' queda perfectamente preparado para que en el Bloque de Simulación (Capítulo 6) los datos apenas tengan que ser tratados:

- **Calles**. Las calles son fragmentadas en segmentos que quedan definidos por su punto inicial y final. Al unir los segmentos, cada calle queda definida por una polilínea. Cada calle debe tener un registro individual. Se muestra un ejemplo en la Figura 3.3.

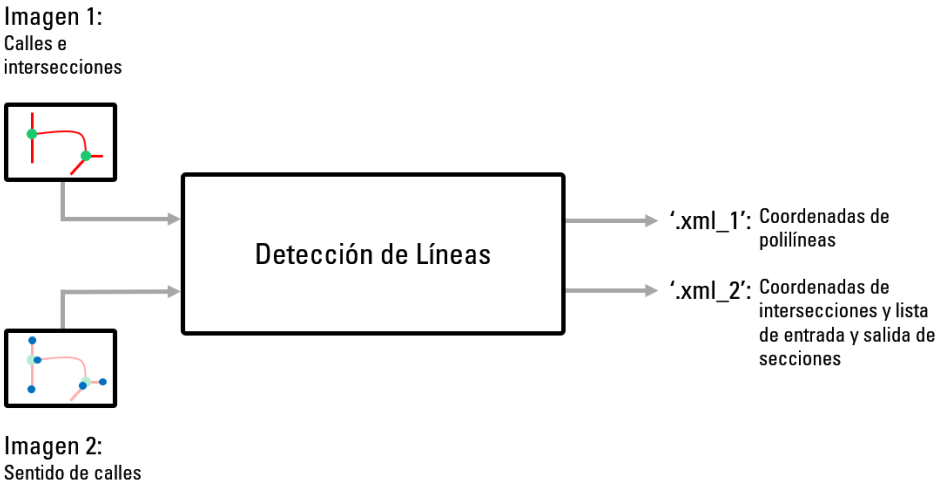


Figura 4.3: Elementos implicados en el bloque de detección de líneas.

- **Intersecciones**: Para la arquitectura diseñada, se requiere que Aimsun disponga de un listado de las calles que salen y entran a cada una de las intersecciones. De forma adicional, se incluyen las coordenadas del centro de cada intersección —aunque no es un dato con el que se trabaje en Aimsun, es un dato útil a la hora de ubicar los elementos del mapa durante el diseño del *software*—. Se muestra un ejemplo en la Figura 3.4.

4.3. Algoritmo. Técnicas para la manipulación de imágenes, extracción de información y su comunicación

Aunque ya se han avanzado ciertas características sobre la mecánica del algoritmo, en esta parte del capítulo se hace una descripción de cada una de las acciones necesarias para detectar correctamente las líneas de la mesa. En la Figura 4.3 de nuevo se ofrece una vista general elementos implicados.

Durante la descripción de todo este capítulo, se trabaja sobre un caso sencillo. La disposición se muestra en la Figura 4.4, y está compuesto por una intersección que conecta cuatro calles. Antes de dar paso a la exposición de los detalles, cabe mencionar que la explicación se ha subdividido en varias cápsulas para facilitar su comprensión. Por su mayor complejidad, las primeras cápsulas explican el proceso seguido para la manipulación y extracción de información de las secciones o calles. La cápsula final hace referencia a las intersecciones, con un proceso mucho más breve y sencillo.

- **Secciones: Manipulación de la primera imagen**. Se realizan las operaciones adecuadas para separar la información de las líneas rojas.
- **Secciones: Extracción de información de la primera imagen**. Se desarrolla el algoritmo para la detección de extremos de las líneas rojas.

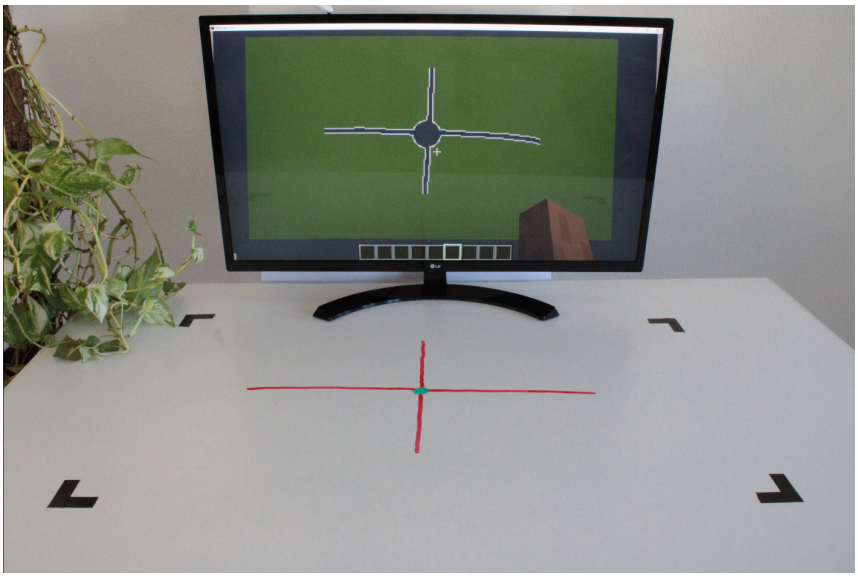


Figura 4.4: Para desarrollar las explicaciones sobre el algoritmo, se utiliza el caso base de la imagen. Contiene cuatro calles y una intersección. Además, se muestran en Minecraft —pantalla— los resultados de la detección.

- **Secciones: Manipulación y extracción de datos de la segunda imagen**. Se extrae la información necesaria para determinar los sentidos de cada sección.
- **Secciones: Preparación de datos de salida**. Minecraft y Aimsun requieren variaciones específicas a la hora de comunicar los datos.
- **Intersecciones**. Manipulación de imagen, extracción de información y preparación de la misma para ser comunicada.

4.3.1. Secciones: Manipulación de la primera imagen

Tal y como se menciona en el capítulo dedicado a las comunicaciones —Capítulo 3—, al algoritmo de detección de líneas se le deben proporcionar dos imágenes. La primera de ellas debe contener los trazos de las secciones o calles junto con las intersecciones. En la Figura 4.5 se muestra la imagen en planta que capta la cámara Intel.

Esta imagen contiene la mayor parte de los datos que se necesitan para definir las líneas: los trazos de color rojo de las cuatro secciones. Para conseguir extraer cada una de estas secciones por separado se debe hacer uso de funciones que proporciona la librería OpenCV —disponible en Python— que permite efectuar una gran cantidad de operaciones [Ope20].

En primer lugar, a esta imagen se le debe aplicar una máscara que extraiga los tonos rojizos de la imagen. Para ello, tal y como ocurre en cualquier proceso de este proyecto que requiera el uso de máscaras de colores, se deben realizar varias acciones:

- **Pasar la imagen a espacio de color HSV**. El espacio de color HSV muestra un comportamiento más robusto ante cambios de iluminación, por lo que es

recomendable trabajar con este sistema de interpretación de colores. En el Fragmento de código 3 se muestra cómo efectuar el cambio de espacio de color en una imagen. El primer argumento es la imagen base, mientras que el segundo indica el espacio de color de origen y el de destino, es decir, de RGB —BGR en OpenCV— a HSV.

```
hsv = cv2.cvtColor(self._imagen, cv2.COLOR_BGR2HSV)
```

Fragmento de código 3: Conversión de imagen a espacio de color HSV.

- **Definir en espacio de color HSV el intervalo deseado**. Es decir, se debe definir un máximo y un mínimo de aquello que se considera como *color rojo* en la imagen tomada por la cámara Intel. Esta tarea puede verse facilitada con conversores online de espacio de color RGB a HSV [Col20]. En este caso, los valores utilizados son:

$$\text{Límite inferior} = (0, 80, 50)$$
$$\text{Límite superior} = (7, 255, 255)$$

- **Aplicar máscara de color**. Sobre la imagen en espacio de color HSV se aplica la máscara de color definida por el intervalo anterior. En el Fragmento de códidgo 4 se muestra cómo aplicar una máscara sobre una imagen. El primer argumento lo constituye la imagen en espacio de color HSV. Los argumentos finales son el límite inferior y superior —en HSV— para el color deseado. Los resultados de estas operaciones se muestran en la Figura 4.6.

```
imagen_rojo = cv2.inRange(hsv, limite_inferior,
limite_superior)
```

Fragmento de código 4: Aplicación de máscara de color.

Las operaciones descritas hasta este punto del capítulo permiten separar y extraer regiones de la imagen —en el caso base, un total de cuatro regiones o líneas blancas—. El nombre que reciben cada una de estas agrupaciones de píxeles que comparten una misma propiedad —color en esta ocasión— es *blob* o *mancha* en castellano. Sin embargo, pese a que se ha conseguido discriminar información útil de la imagen inicial, todavía no se dispone de datos definitivos como son las coordenadas de las manchas. Para ello, se requiere hacer uso de los comandos que se incluyen en el Fragmento de código 5, que permite recoger las coordenadas de los contornos de las manchas observadas en la Figura 4.6. Cabe mencionar en este punto que se podrían utilizar otras técnicas como la esqueletonización [Sta17]. En este proyecto, sin embargo, tras varias pruebas en las que los resultados de la esqueletonización ofrecen como resultado figuras muy erosionadas consumiendo un tiempo de procesamiento considerable, se ha optado por las técnicas que se describen en este capítulo.

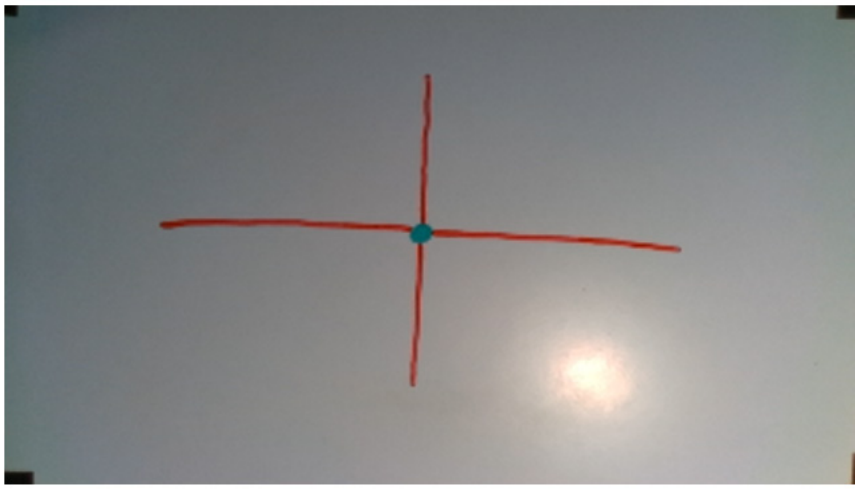


Figura 4.5: Imagen en planta tomada por la cámara Intel —integrada en el proyecto—. Se muestra el caso mostrado en la Figura 4.4, pero en esta ocasión como imagen de entrada al sistema.

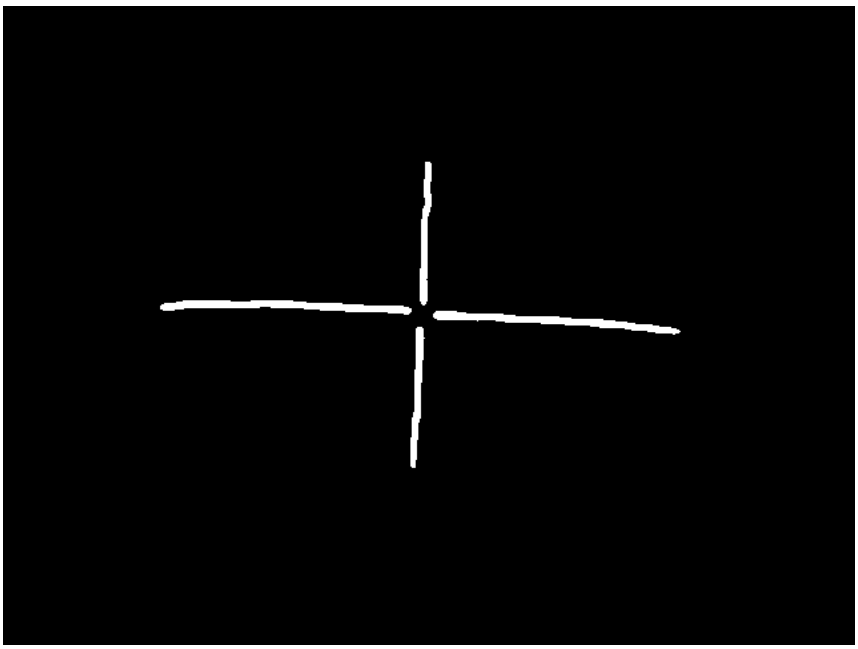


Figura 4.6: La máscara de *color rojo* aplicada sobre el caso de ejemplo arroja una imagen en negativo en la que los píxeles del color objetivo —rojo— se representan en blanco. El resto de píxeles son de color negro.

```
contornos_rojos = cv2.findContours (imagen_rojo,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE) contornos_rojos =
imutils.grab_contours(contornos_rojos)
```

Fragmento de código 5: Extracción de las coordenadas de los contornos.

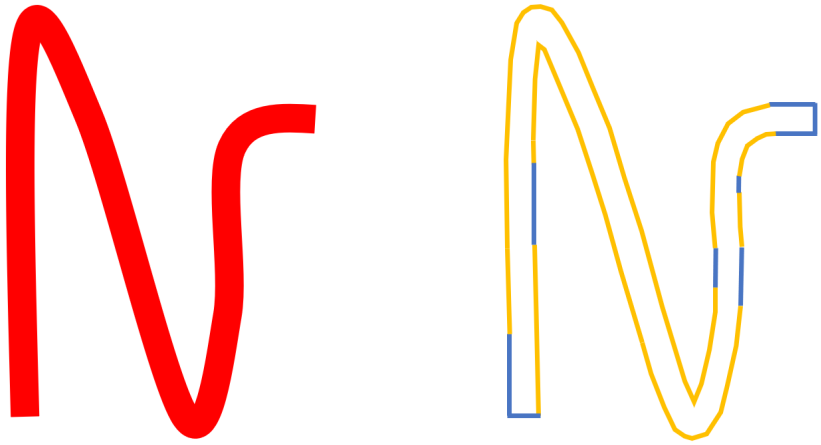


Figura 4.7: Resultados gráficos del Fragmento de Código 5 —no da como resultado una imagen, sino una lista de coordenadas con los valores x e y de los puntos del contorno—. Los contornos rodean por completo la línea detectada.

Es importante tener en cuenta qué consiguen estos comandos, pues las técnicas que se explicarán en las secciones siguientes son una consecuencia de los datos que se extraen aquí. En la primera línea que aparece en el Fragmento de código 5, el primer argumento es la imagen resultado de la máscara, es decir, la Figura 4.6. El siguiente argumento permite que se tomen los contornos exteriores de las manchas de la Figura 4.6. El último argumento permite nutrir con todos los puntos del contorno el listado de coordenadas: frente a otras posibilidades que simplifican líneas de contorno ortogonales guardando los puntos inicial y final, `cv2.CHAIN_APPROX_NONE` registra todas los puntos que las componen.

En la Figura 4.7 se muestra un ejemplo gráfico de lo que se describe en este párrafo. A la izquierda de la imagen se muestra una línea genérica que representa una sección. A la derecha se muestra el resultado gráfico que se conseguiría al ejecutar lo mostrado en el Fragmento de Código 5. En color azul aparecen las líneas que quedarían simplificadas —las ortogonales— al no utilizar el argumento `cv2.CHAIN_APPROX_NONE`. Sin embargo, al hacer uso de dicho argumento, todos los puntos o píxeles que componen el contorno, incluidos los de las líneas azules, quedan registrados.

El objetivo con el que se aborda la manipulación y extracción de datos de las imágenes es poder aportar a Aimsun una ruta de coordenadas. Esta ruta de coordenadas debe tener un punto de inicio y final para cada sección. Sin embargo, como se puede apreciar en la Figura 4.7, con las operaciones aplicadas hasta ahora sólo se consigue un contorno o circuito cerrado. Es decir, si se comunicasen las coordenadas obtenidas hasta ahora por el algoritmo, Aimsun haría la interpretación que se muestra en la parte izquierda de la Figura 4.8. Sin embargo, los resultados a los que se aspira se muestran en la parte derecha de la misma imagen. Para conseguirlo, es necesario desarrollar más el algoritmo.

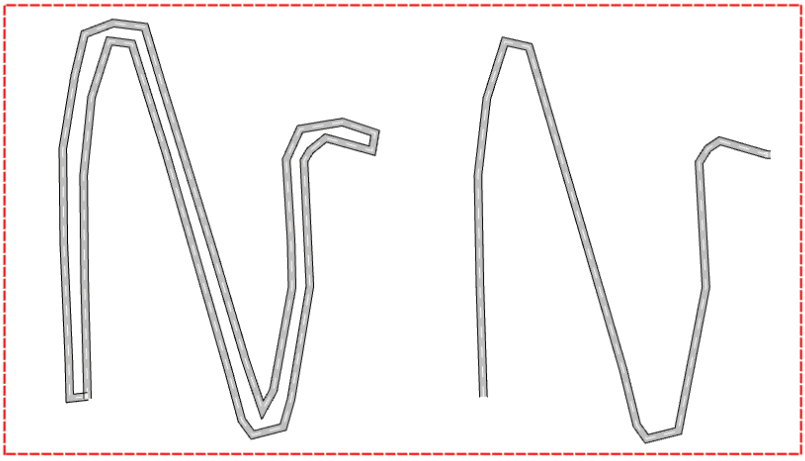


Figura 4.8: Representación en Aimsun de los resultados que se obtendrían si se incluyesen los resultados obtenidos hasta ahora. A la derecha se observa un circuito en lugar de una sección con inicio y final. A la izquierda se observa el objetivo que debe alcanzar el algoritmo que se explica en las secciones siguientes.

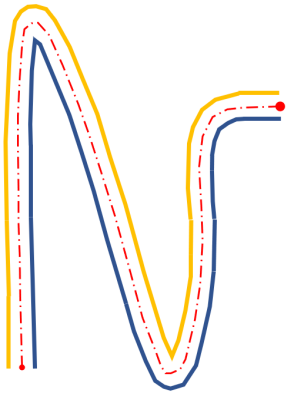


Figura 4.9: El contorno que se muestra en la Figura 4.7 puede dividirse en dos trazos: uno interior —color azul— y otro exterior —color amarillo—.

4.3.2. Secciones: Extracción de información de la primera imagen

Como se puede apreciar en la Figura 4.8, existe una notable diferencia entre lo que los usuarios desean representar y los resultados obtenidos hasta ahora por el algoritmo. La técnica que se describe a continuación resuelve el problema de los contornos —el circuito que rodea por completo a la línea detectada— y consigue registrar las coordenadas que llevan desde el punto inicial al punto final de la línea respetando su trazado.

En la Figura 4.9 se muestra cómo de un extremo a otro de una línea genérica se pueden extraer dos trazos útiles para obtener los resultados deseados. Como se puede apreciar, tanto el trazo interior de color azul, como el trazo exterior de color amarillo incluyen un desplazamiento frente al eje central —línea roja que representa el trazo ideal—. Esta distorsión es asumible y casi imperceptible en Aimsun. De

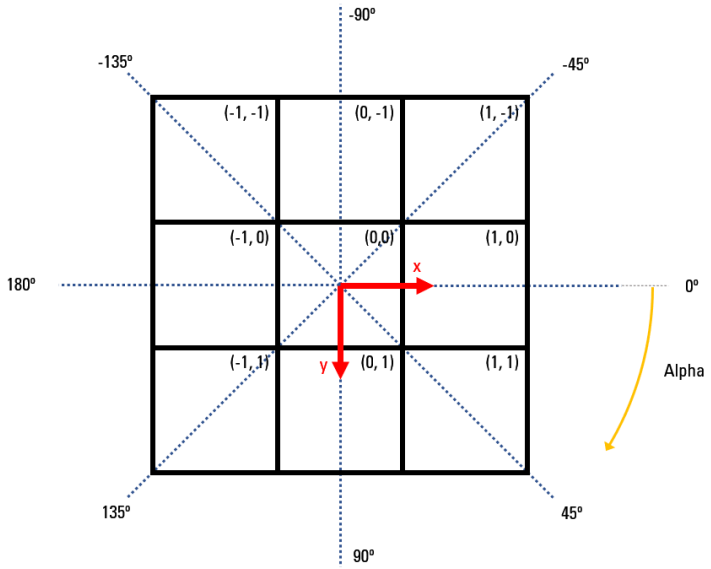


Figura 4.10: Ejemplo de la ordenación de píxeles con la definición de los ejes de coordenadas y origen del ángulo *Alpha*.

esta forma, recoger los datos de uno de los trazos —azul o amarillo— soluciona el problema planteado. Para abordarlo, resulta necesario localizar los extremos de la curva

Como se puede apreciar en el caso genérico mostrado en la Figura 4.9, los trazos azul y amarillo tienen su origen y final en los extremos de la línea original —Figura 4.7—. Si se quiere registrar la secuencia de coordenadas de cualquiera de estos trazos, es necesario implementar un algoritmo que sea capaz de detectar los extremos de cualquier línea registrada. Para entender en qué se basa el algoritmo, es necesario definir algunos conceptos primero.

En la Figura 4.10 se representan un total de nueve píxeles, cada uno de ellos etiquetado con sus coordenadas relativas x, y. En esta figura se hace una descripción gráfica de los conceptos básicos sobre los que se sustenta el algoritmo de detección de extremos y de ella se puede extraer información muy importante.

- **Eje de ordenadas**. En OpenCV, el sentido positivo del eje vertical se define de arriba hacia abajo. El eje de abscisas —horizontal— queda definido con su sentido habitual. Este cambio de sentido puede requerir reajustes —traslaciones— a la hora de enviar los datos extraídos a Aimsun, que trabaja con el eje vertical definido en sentido contrario.

- **Origen de coordenadas**. OpenCV trabaja con el origen de coordenadas situado en la esquina superior izquierda de la imagen. Como consecuencia de esto y del punto anterior, todas la coordenadas con las que se trabajan tienen valores positivos.

- **Valores de los ángulos**. Debido a la orientación de los ejes que impone OpenCV, el sentido positivo del ángulo *alpha* es el giro horario. También cabe

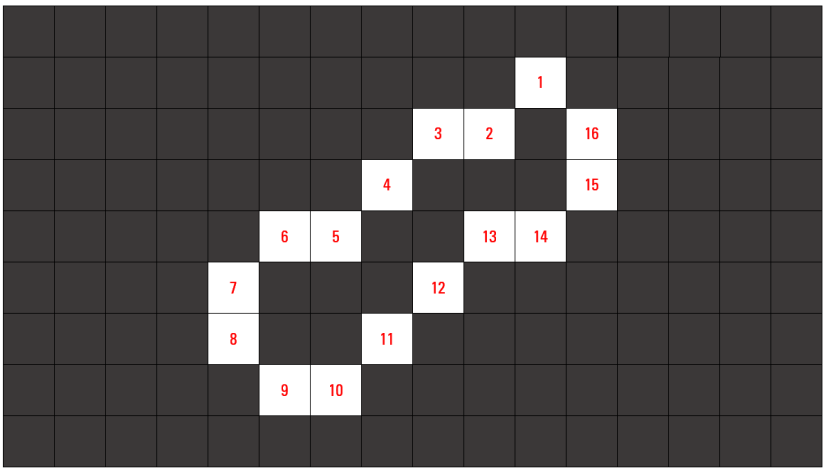


Figura 4.11: El algoritmo implementado hasta ahora extrae un contorno formado por píxeles contiguos. En esta imagen se muestra un contorno genérico.

tener en cuenta que la función utilizada en Python para extraer los valores de los ángulos —*atan2*— tan solo devuelve valores de módulos de ángulos comprendidos entre 0 y 180 grados. Esta hecho supone una limitación que el algoritmo implementado debe solventar.

Una vez definido el sistema de coordenadas y el ángulo *alpha*, se presenta la Figura 4.11. En esta imagen se representa un contorno genérico donde los píxeles blancos aparecen numerados. Esta numeración corresponde al orden de registro que siempre se cumple para cada contorno detectado. El algoritmo toma como primer píxel de cada contorno aquel que tiene menor cota y. Una vez registrado este, la secuencia continúa siempre en sentido antihorario.

En la Figura 4.12 se muestra el resultado del algoritmo al calcular los ángulos que resultan al unir píxeles consecutivos. Como se puede apreciar, de los píxeles 1 a 6 los ángulos registrados tienen valores de 180 ó 135 grados; mientras que desde el píxel 10 hasta el 16, los ángulos tienen un valor medio negativo. En el tramo del píxel 7 al 9 se localiza el extremo, es decir, donde ocurre un cambio drástico en los valores de ángulos. El algoritmo se basa en el correcto registro de los valores de ángulos y en la localización de un cambio brusco en sus valores. Cuando esto ocurra, se habrá detectado un extremo. Su mecánica es la siguiente:

- **Cálculo de ángulo**. El cálculo de los ángulos se efectúa al unir dos píxeles consecutivos —líneas azules en la Figura 4.12—. Para ello se utiliza la función *atan2*, de la librería *math*.

- **Cálculo de media móvil**. El algoritmo implementa una media móvil con los últimos n valores de ángulos calculados. La media móvil permite tener un valor de ángulos robusto, es decir, una tendencia sólida de los ángulos. Una vez registrado los n primeros ángulos, cada valor de *alpha* calculado se compara con el valor de la media móvil y se comprueba si se deben realizar correcciones en el valor del ángulo registrado. Una vez realizado este paso, el valor de la media móvil también es utilizado para la detección de los extremos.

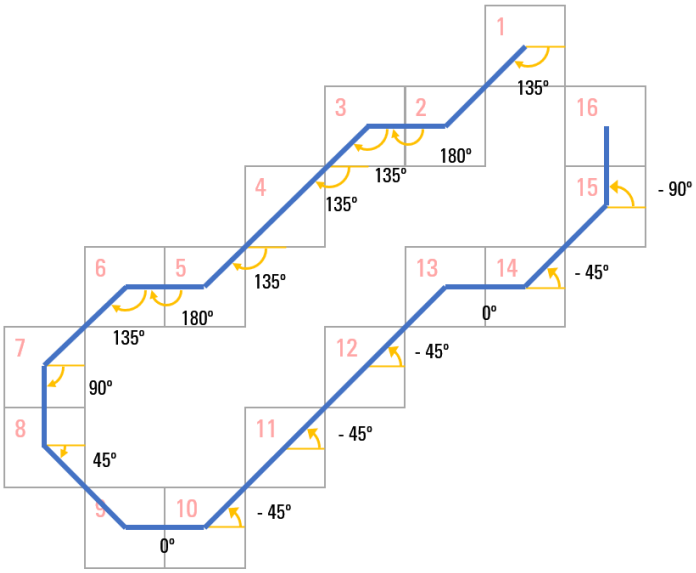


Figura 4.12: Secuencia de ángulos para el contorno mostrado en la Figura 4.11

- **Corrección del valor del ángulo**. Como se ha apuntado anteriormente, el rango de valores de módulo que devuelve la función *atan2* está comprendido entre 0 y 180 grados. Es decir, esta función trigonométrica devuelve el valor del ángulo leído en el sentido que ofrece un módulo menor: por ejemplo, ante un ángulo de -135 grados -ó +225 grados, leyendo el ángulo *alpha* en sentido contrario, la función *atan2* siempre devuelve el valor de menor módulo, es decir, -135 grados en este caso. Aunque geométricamente un ángulo de 225 grados sea equivalente a un ángulo de -135 grados, desde el punto de vista numérico, no es lo mismo. Es decir, representados en una gráfica, aparece un salto de 360 grados entre un valor y otro que introduciría datos incorrectos al algoritmo. Para determinar qué valor debe registrarse en cada caso, se recurre al valor de la media móvil. Este valor aporta una tendencia del sentido en el que se realizan las lecturas de ángulos —horario o antihorario—. Para cada ángulo leído, se detecta el cuadrante al que pertenece y se comprueba el valor de la media móvil en ese momento. En la Figura 4.13 se representa esta explicación de forma gráfica. En el gráfico de la izquierda se muestra cómo el ángulo *alpha*, con independencia del valor de la media móvil, puede ser leído en sentido positivo o negativo. En el gráfico central y de la derecha se muestra cómo el valor de la media móvil —marcado con línea de color azul— afecta a la interpretación del valor del ángulo. Cuando la media móvil es negativa, se interpreta que el ángulo es negativo, mientras que cuando la media móvil es positiva, el ángulo se lee en sentido positivo. Con el algoritmo implementado, las correcciones de ángulos permiten tener un rango de valores entre 0 y 360 grados —una vuelta completa—, lo que, como se apuntaba al principio de este capítulo, excluye la detección de espirales —que completan más de una vuelta y superan los límites de ángulos marcados por este algoritmo—.

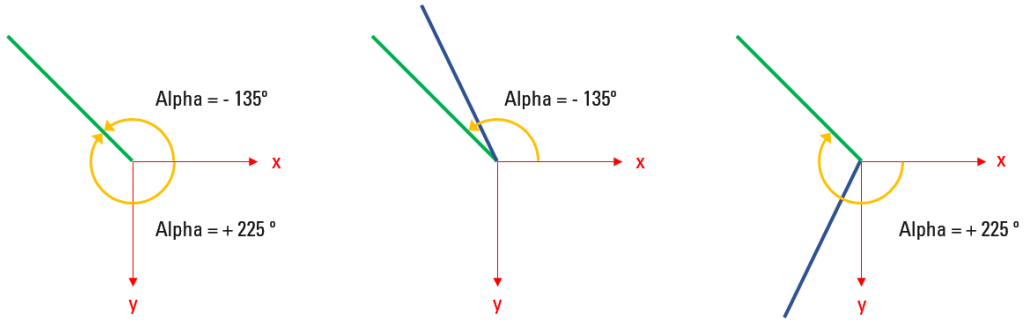


Figura 4.13: Interpretación del ángulo en función del valor de la media móvil.

- **Detección de extremos**. Además de ayudar al correcto registro de los valores de ángulos, el valor de la media móvil es crucial para la detección de los extremos en las líneas. Como se ha mencionado en su apartado, la media móvil aportar un valor robusto de la tendencia de los ángulos del contorno. Por ello, una vez realizada la corrección del ángulo, se comprueba si entre el valor de *alpha* corregido y el valor de la media móvil existe una diferencia notable. Es decir, se comprueba lo establecido en la Ecuación 4.1. Si la diferencia es superior al umbral definido, establecido en 33 grados dentro del algoritmo, entonces se considera que el algoritmo ha detectado uno de los extremos de la línea dibujada por los usuarios.

$$\text{si } |\alpha| - |Media| > Umbral \rightarrow \text{Extremo detectado} \tag{4.1}$$

  En la Figura 4.14 se muestra la gráfica correspondiente a la curva horizontal situada a la derecha en la Figura 4.5, junto con una representación de la curva segmentada en varios tramos coloreados. En la gráfica se muestra la evolución de los datos necesarios para la detección de los extremos. En primer lugar, las marcas amarillas muestran el valor del ángulo *theta* sin corregir, es decir, el valor de *theta* que devuelve la función *atan2*. La curva azul de la gráfica muestra la evolución del ángulo *theta* corregido, es decir, una vez comprobado —y corregido en caso necesario— el sentido de lectura del ángulo. Como se puede apreciar, en las muestras en las que no es necesaria incluir ninguna corrección, las marcas amarillas coinciden con las azules. En color rojo aparece la evolución de la media móvil. Como se puede observar, su comportamiento es mucho más suave que el de la curva azul, pero como cabría esperar, sigue sus tendencias. Por otro lado, en la gráfica también se representa la curva verde, que solo tiene dos estados: extremo no detectado, y extremo detectado. Una vez introducida la gráfica y las curvas que en ella se representan, se explica de nuevo la mecánica del algoritmo, esta vez apoyado en un caso de detección real.

  Las curvas de la gráfica sobre las que se debe centrar la atención son la curva roja —media móvil— y la azul —*theta* corregido—. En la fase inicial de la media móvil —curva roja—, el valor parte de de aproximadamente 20 grados y tiende hacia 0. Estos registros son los correspondientes al tramo amarillo, situado en el extremo izquierdo de la curva analizada. El algoritmo, como se

mostraba en la Figura 4.11, comienza la lectura de píxeles por el de menor cota vertical y continúa su recorrido en sentido antihorario. Como el algoritmo comienza la lectura justo en un extremo, no puede detectarlo puesto que no existen muestras previas con las que realizar una comparativa. Esto no es un problema, puesto todas las líneas contienen dos extremos, aunque no se haya detectado este por falta de información previa, cuando el algoritmo analice las muestras del segundo extremo sí habrá muestras suficientes como para realizar su detección.

Desde aproximadamente la muestra 50 hasta la 180 de la gráfica, la media móvil —curva roja— tiene un valor establecido en 0 grados —lógico si se atiende a la definición de ejes de coordenadas y origen de ángulo *alpha*—. Este tramo de la gráfica corresponde al trazo inferior del contorno de la curva analizada —trazo azul, recorrido de izquierda a derecha—.

Aproximadamente en la muestra 180, el algoritmo comienza a leer los píxeles correspondientes al extremo situado a la derecha —tramo coloreado en rojo—. Si se analiza la gráfica, en este punto existe una caída repentina del valor de *theta* corregido —curva azul—. En este tramo de la curva, si se aplica la Ecuación 4.1, se puede comprobar cómo se cumple la condición establecida y por tanto, se detecta el extremo. De nuevo en la gráfica, se puede ver cómo la curva verde —detección de extremo— salta a su estado activo cuando existe un cambio brusco en los valores registrados.

Por último, siguiendo la lectura del contorno en sentido antihorario, la media móvil se establece cercana a los -180 grados al recorrer de izquierda da derecha el tramo superior de la curva horizontal, de color verde.

Dentro del algoritmo de detección de extremos, existen varias opciones. La primera de ellas es que el algoritmo no detecte ningún extremo. En este caso, la calle representada por los usuarios no sería incluida en el modelo. La segunda de ellas es que el algoritmo, como en el caso de la Figura 4.14, sólo detecte un extremo. En tal caso, a la hora de realizar el registro de las coordenadas que definen la calle, se incluyen todos los píxeles del contorno desde el punto reconocido como extremo hasta último píxel del contorno, es decir, se registran todas las coordenadas que definen el tramo verde del contorno de la Figura 4.14. La última opción es que el algoritmo reconozca los dos extremos, como ocurre en el ejemplo mostrado en la Figura 4.16. En esta imagen se estudia el mismo dibujo de calle que en la Figura 4.14, pero se incluyen modificaciones del parámetro n, con el que se puede conseguir una mayor sensibilidad de detección. En los casos en los que se detectan los dos extremos de la calle, se registran todas las coordenadas de los píxeles comprendidos entre los dos extremos, es decir, en el caso de la Figura 4.14, se registrarían las coordenadas de los puntos perteneciente al tramo inferior de color azul.

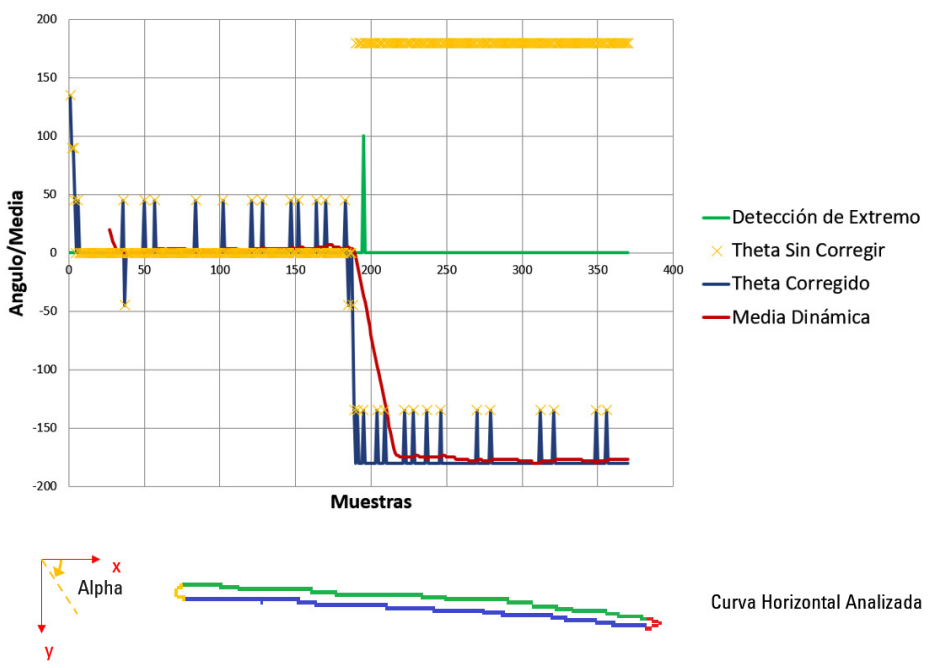


Figura 4.14: Gráfica de la curva horizontal situada a la derecha en la Figura 4.5. En este tipo de gráficas que se generan automáticamente en Excel para cada línea, se representan las curvas con la información necesaria para la detección de los extremos de cada línea.

Antes de dar paso a las siguientes secciones de este capítulo, cabe mencionar algunas observaciones relevantes sobre la mecánica del algoritmo:

- **Observación 1: Detector de valores atípicos**. En versiones iniciales del algoritmo se incluye un detector de valores atípicos. Esta implementación atiende a la posibilidad de que la extracción de los contornos no sea limpia, es decir, la disposición de los píxeles sea demasiado accidentada y por tanto aparezcan ángulos que no describen la realidad. La implementación de este sistema es redundante cuando en el Fragmento de código 4 se define un rango de color lo suficiente amplio como para captar el gradiente de color que aparece en en el entorno de la línea. En la parte derecha de la Figura 4.15, se muestra una imagen con los píxeles que componen el extremo de una línea genérica. Dicha imagen ha sido tomada por la cámara Intel RealSense D435 y ampliada con un zoom del 2000 %. Si se traza el contorno considerando exclusivamente los píxeles más rojizos, se obtiene el contorno de color azul. Por el contrario, si se ampía el rango considerado como *color rojo* y se incluyen los píxeles de tonos rosados, entonces el contorno es mucho más suave —contorno verde—. En el caso del color azul podrían aparecer valores atípicos de ángulos debido a lo escarpado del contorno, al contrario de lo que ocurre en el contorno verde, más suave y por lo tanto con una tendencia de ángulos más sólida. Además de permitir obtener unos registros de ángulos más suaves, ampliar el rango que define el *color rojo* permite que se eviten discontinuidades dentro de una misma línea —es decir, que cualquiera de las calles mostradas en la Figura 4.6 apareciese como una sucesión de manchas blancas inconexas—, lo que arruinaría por completo el funcionamiento del

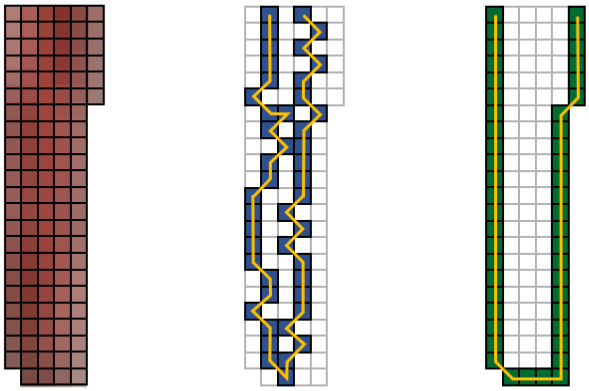


Figura 4.15: Un espectro de *color rojo* amplio permite obtener contornos más suaves y por tanto valores de ángulos más sólidos.

algoritmo.

- **Observación 2: Grosor de línea**. Del punto anterior se puede extraer una consecuencia que supone una limitación para el algoritmo. Para la detección correcta de las líneas y disponer de una base sólida de píxeles, es necesario implementar un rango de *color rojo* amplio. Esto hace que el algoritmo tome píxeles que con una ligera tonalidad rosada en los extremos, aumentando así el grosor de las líneas y produciendo que en ocasiones, dos líneas rojas o calles pintadas por los usuarios con muy poca separación entre ellas, contacten y se fusionen en algún punto al aplicar la máscara de color rojo.

- **Observación 3: Radios de curvatura**. Como se mencionó en la parte dedicada a la definición de secciones e intersecciones de este capítulo, los radios de curvatura tienen un límite inferior. Es decir, no se pueden detectar calles con radios de curvatura excesivamente pequeños, pues el algoritmo, basado en la detección de cambios bruscos del ángulo *theta*, confundiría la curva con un extremo al producirse el mismo fenómeno en ambos casos.

- **Observación 4: Número de muestras n para el cálculo de la media móvil**. A la hora de calcular la media móvil, cabe recordar que cuanto mayor sea el valor de n, más robusto es el algoritmo, pero mayor es el desfase entre la señal de la media móvil y la señal de ángulo detectado, lo que implica que cuanto mayor es el valor de n, mayor es el número de muestras procesadas hasta que se produce la detección del extremo. En la Figura 4.16 se muestra un ejemplo del efecto de n en el desfase de ambas señales. Para la misma curva horizontal analizada en la Figura 4.14, se realiza el estudio con un valor de muestras de la media móvil, n, de 10 y 25. Cuando el valor de n es menor, la media móvil presenta más rizado y es más sensible a cambios en el ángulo *theta* —corregido—. Esta sensibilidad lleva al algoritmo a ser capaz

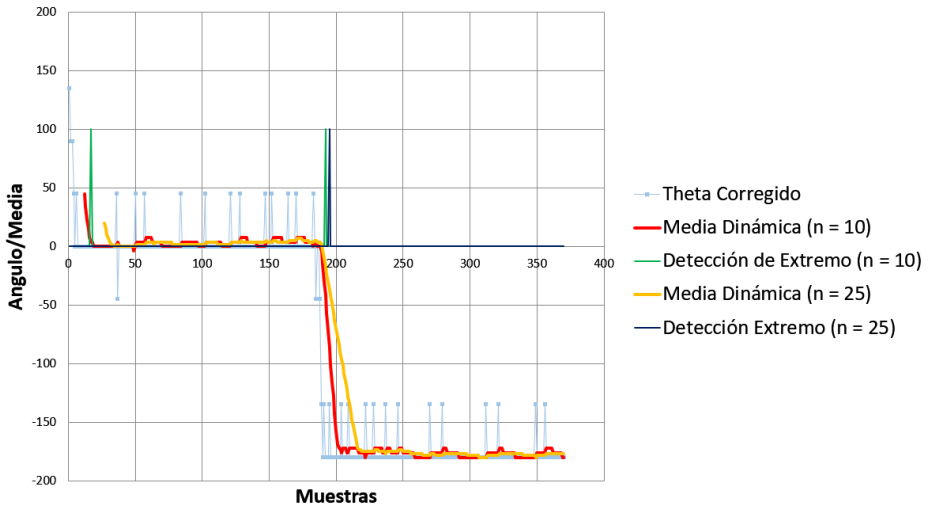


Figura 4.16: El número de muestras n consideradas para el cálculo de la media móvil afecta a la robustez y sensibilidad del algoritmo. Valores de n cercanos a 25 ofrecen un buen compromiso entre sensibilidad y robustez.

incluso de detectar el primer extremo de la curva —se detectan dos extremos, como muestra la curva verde de la gráfica—, aunque en otros cosos podría conducir a la aparición de falsos positivos. Cuando n es mayor, se puede ver un retraso más acentuado entre las señales —es decir, entre el ángulo *theta* y la señal de la media móvil— En un compromiso entre sensibilidad y robustez —corroborando con otro tipo de curvas— se opta por establecer el número de muestras de la media móvil en 25.

- **Observación 5: Proceso de depuración del algoritmo: Excel**. Para poder depurar el algoritmo de manera eficiente e independiente de Python se recomienda exportar los datos a Excel. Esto permite disponer de los datos de forma aislada y permanente sin necesidad de ejecutar de nuevo el algoritmo —lo que podría hacer variar los datos debido a la iluminación con la que se captan las imágenes o cualquier otra variable externa—. Aunque existe documentación para la comunicación entre Python y Excel, en los Fragmentos de Código 6 a 9 se muestran los comandos utilizados en este proyecto. En ellos se expone cómo importar las librerías necesarias, cómo activar una hoja Excel y acceder a una celda, cómo crear un gráfico y cómo guardar el documento.

```
from openpyxl import Workbook
from openpyxl.chart import (
ScatterChart,
Reference,
Series,
)
```

Fragmento de código 6: Comandos para la comunicación Python - Excel. Importación de librerías.

Para cada curva, se registra el valor del ángulo *theta* corregido, así como el valor de la media móvil actualizado para cada muestra de entrada. Estos datos se registran ordenados en columnas y cada dato se introduce en una celda tal y como se muestra en el Fragmento de código 7.

```
book = Workbook()
sheet = book.active

sheet.cell(row=fila, column=columna).value = dato
```

Fragmento de código 7: Comandos para la comunicación Python - Excel. Activación de hoja Excel y acceso a celda.

Una vez se ha han registrado todos los datos en sus respectivas celdas, se puede dar paso a la elaboración de los gráficos. Estos gráficos permiten depurar el algoritmo al ofrecer los datos de manera visual. Cada gráfico debe incluir la definición de su eje horizontal y vertical. Como se muestra en el Fragmento de código 8, cada serie de datos debe recoger los datos del *eje x* y del *eje y*. Mediante el comando `chart.series.append` se pueden incluir varias series en un mismo gráfico. En el comando de la última línea, el segundo argumento indica la celda donde se dibuja el gráfico.

```
chart = ScatterChart()
chart.title = Çurva"
chart.x_axis.title = 'Número de Muestra'
chart.y_axis.title = 'Angulo'

datos_x = Reference(sheet, min_col=columna_1,
min_row=fila_1,
max_row=fila_2)
datos_y = Reference(sheet, min_col=columna_2,
min_row=fila_1,
max_row=fila_2)

serie_datos = Series(datos_y, datos_x,
title_from_data=True)
chart.series.append(serie_datos)

sheet.add_chart(chart, "D10")
```

Fragmento de código 8: Comandos para la comunicación Python - Excel. Elaboración de un gráfico.

Por último, en el Fragmento de Código 9, se muestra cómo guardar el documento que incluye el registro de datos y los gráficos creados. Para ello,

se debe incluir un directorio con el nombre del archivo. En caso de que no se indique un directorio, el archivo aparece en la carpeta *Usuario* del PC.

```
book.save(Directorio + 'nombre_documento.xlsx')
```

Fragmento de código 9: Comandos para la comunicación Python - Excel. Cómo guardar el documento.

4.3.3. Secciones: Manipulación y extracción de datos de la segunda imagen

Con la información registrada hasta ahora —coordenadas que definen las calles pintadas en la mesa— se podría empezar a recrear un escenario virtual con las calles en Minecraf o Aimsun, pero aún no sería posible formar un entramado completo que uniese varias secciones. Para ello es fundamental definir el sentido de cada sección, pues en Aimsun, la circulación de los vehículos de unos puntos a otros, como el lógico, está asociada a los sentidos que ofrecen las secciones.

En este apartado del capítulo se explica cómo mediante el uso casi exclusivo de funciones básicas de la librería OpenCV se puede extraer la información necesaria para determinar el sentido de las secciones y enriquecer el modelo.

La imagen que se recibe en esta ocasión es la misma que en Figura 4.4, pero incluyendo marcas que indican el sentido. Las marcas deben situarse sobre uno de los extremos de cada línea, de tal forma que el sentido de circulación de cada sección se define como el recorrido desde el extremo sin marca hasta el extremo con marca. En la Figura 4.17 se muestra un ejemplo.

Aplicando una máscara de color —en este caso verde—, se pueden extraer las coordenadas de las marcas realizadas por los usuarios para determinar los sentidos. Una vez se han extraído las coordenadas —se realiza el mismo proceso que se explica en el último apartado de este capítulo—, se realiza un cruce de información entre las coordenadas de los extremos de cada sección y las coordenadas de las marcas de sentido recién registradas. Es decir, se comprueba a qué extremo corresponde cada marca de sentido, comparando las coordenadas de ambos elementos.

Una vez se ha detectado el sentido de la sección, se debe corroborar si el orden en el que están registradas las coordenadas es el correcto. Si no es así, el algoritmo debe actuar para invertir el orden de los puntos que definen la sección. Para facilitar la comprensión de este proceso se vuelve al ejemplo mostrado en la Figura 4.14, donde se había demostrado cómo el algoritmo había detectado un sólo extremo. Al detectarse un sólo extremo, y teniendo en cuenta el sentido antihorario con el que se recorre el contorno de la línea, el algoritmo extrae el tramo superior —de color verde— ordenando de derecha a izquierda los puntos que lo componen. Si se observa la Figura 4.17, se puede apreciar cómo la marca de la línea horizontal de la derecha indica el sentido contrario. Por lo tanto, es necesario invertir el orden de registro. En la imagen Figura 4.18 se incluye el proceso gráfico de

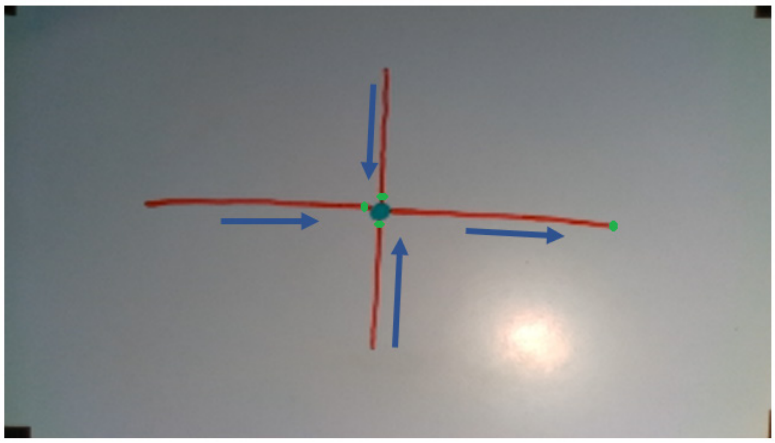


Figura 4.17: En esta imagen los sentidos se definen con marcas de color verde. Cada sección debe tener su marca. Un vehículo recorre la sección desde el extremo sin marca hasta el extremo con marca. Las flechas azules no corresponden a ningún tipo de dato captado por la cámara y sólo pretenden facilitar la comprensión de la imagen al lector.

toda la información descrito en este párrafo y que se condensa en los siguientes pasos.

- **Detección de extremos**. En el caso explicado en la Figura 4.14 sólo se detecta un extremo. En la Figura 4.18 este extremo es el denominado *E-2*. El extremo *E-1* no es detectado por el algoritmo por las causas mencionadas en su apartado, pero es fácilmente localizable al ser inicio y final del contorno.

- **Extracción del tramo de contorno adecuado**. En los casos en los que sólo se localiza un contorno, el registro se realiza desde el punto etiquetado como extremo —*E-2*— hasta el último punto que define el contorno —el final—. Esta estrategia permite definir adecuadamente cualquier sección de las expuestas en la Figura 4.1.

- **Detección de marcas de sentido**. Como se indica en los párrafos anteriores, se comprueba si en el entorno de cada extremo existe una marca de sentido. En el extremo *E-2* aparece una marca de sentido dentro de su área de influencia. No ocurre lo mismo en el caso del extremo *E-1*.

- **Corrección del orden del registro de puntos**. La marca de sentido registrada en el extremo *E-2* hace que sea necesario cambiar el orden de registro de los puntos de la curva. El sentido correcto —indicado por los usuarios— es de izquierda a derecha.

Este proceso se realiza con todos las secciones detectadas, de tal forma que al finalizar, cada sección tiene asociado un registro con los puntos ordenados

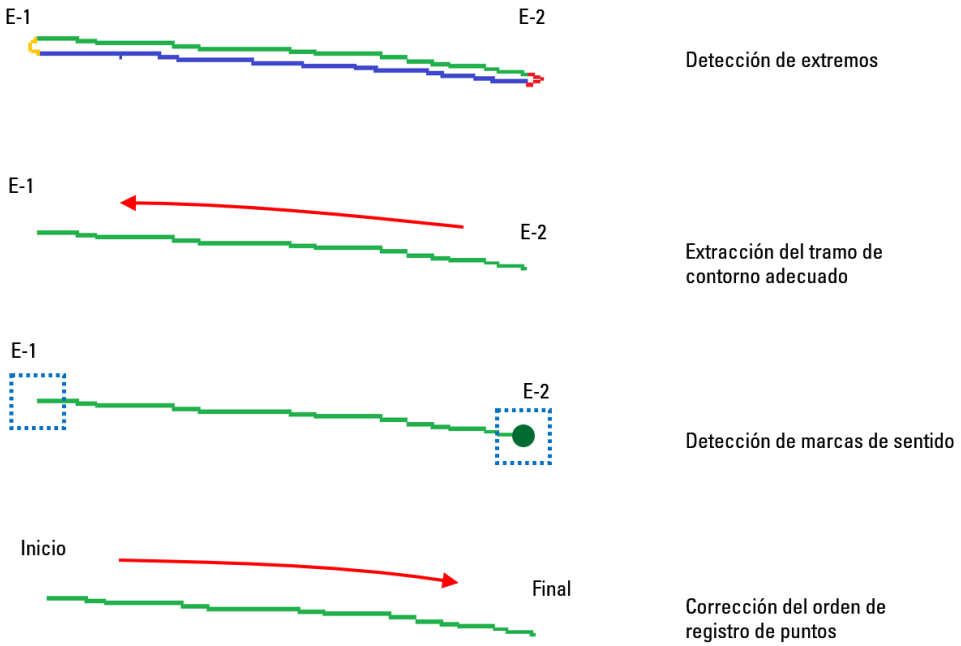


Figura 4.18: Proceso de transformación que sufre el contorno de una línea hasta convertirse en una cadena de coordenadas registradas en el orden correcto.

correctamente.

4.3.4. Secciones: Preparación de datos de salida

Los datos —coordenadas— que se extraen de este bloque deben ser enviados tanto a Minecraft como a Aimsun. Cada uno de estos programas tiene sus propias peculiaridades —por ejemplo Minecraft impone la utilización de cubos, una característica que lo define por completo—. Además las APIs Python-Minecraft y Python-Aimsun incluyen herramientas y comandos a los que hay que adaptarse a la hora de realizar las comunicaciones. Aunque se hace una exposición más detallada de cómo se trabaja con las coordenadas en Aimsun y Minecraft en sus respectivos capítulos, a continuación se explica cómo procesar el registro de coordenadas según el programa que recibe la información.

- **Minecraft**. Los comandos y opciones de Minecraft posibilitan realizar barridos de bloques. En estos barridos se unen dos puntos —definidos por las coordenadas extraídas por el algoritmo— y se colocan hileras de bloques de color negro con un bloque blanco en cada extremo. Para conseguir la mayor precisión posible y adaptarse a las curvas que aparecen, Minecraft requiere que se envíen las coordenadas de todos los puntos registrados, de tal forma que se produce un barrido entre cada dos puntos registrados hasta completar la calle. En la Figura 4.19 se muestra un ejemplo sencillo. En el Capítulo 7 se aporta un mayor nivel de detalle en la descripción del algoritmo de Python que recrea los escenarios virtuales y las carreteras.
- **Aimsun**. De modo similar a lo que ocurría en Minecraft, en Aimsun, una sección queda definida por su punto inicial y su punto final. En este

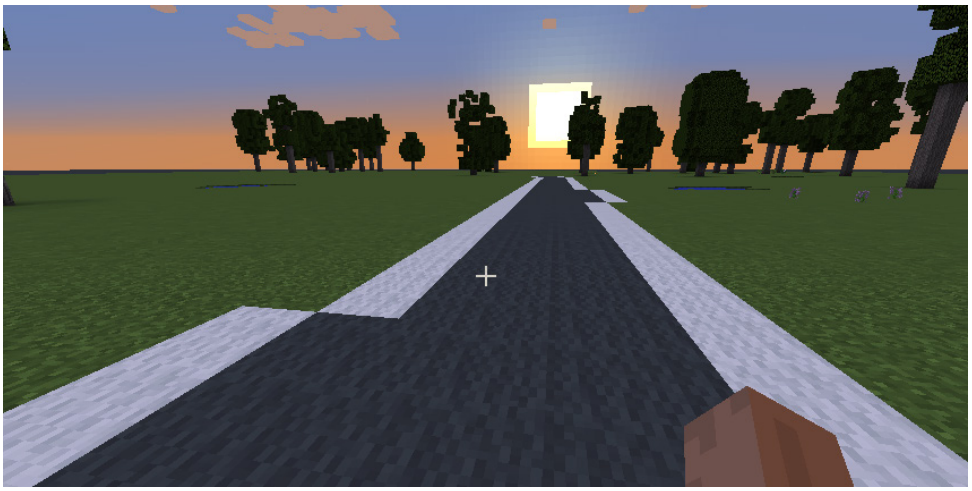


Figura 4.19: Vista a pie de calle de una carretera trazada en Minecraft.

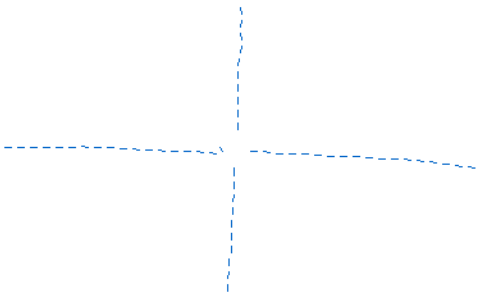


Figura 4.20: Representación gráfica de los datos extraídos de la Figura 4.5 antes de ser enviados a Aimsun. Se puede apreciar la segmentación y la aparición de huecos entre segmentos que permite suavizar los trazos de carreteras en Aimsun.

programa, la técnica utilizada se basa de nuevo en la segmentación, es decir, la representación de curvas complejas mediante la utilización de una gran cantidad de pequeñas rectas que se adaptan a la forma deseada. Sin embargo, para representar correctamente las secciones en Aimsun, no es necesario enviar la totalidad de puntos del registro. La técnica utilizada en el caso de Minecraft daría como resultado carreteras muy escabrosas. Por lo tanto, en este caso se implementa un algoritmo que posibilita la creación de pequeños segmentos. Para ello, se selecciona uno de cada cinco puntos del registro y no se permite la unión de dos segmentos consecutivos. Aunque el algoritmo implementado ofrece resultados satisfactorios, se puede complementar con el algoritmo de Ramer-Douglas-Pecker [Kar12]. En la Figura 4.20 se muestra el resultado gráfico de esta técnica —todavía sin implementarse en Aimsun—. La aparición de los huecos generados permite suavizar el trazo de las secciones en Aimsun. El mecanismo para rellenar los huecos que aparecen entre los pequeños segmentos se explica en el Capítulo 6.

Por otro lado, como se ha mencionado ya en varias ocasiones, la comunicación Python - Aimsun se realiza a través de arhivos '.xml'. A continuación, en el Fragmento de código 10 se incluyen los comandos básicos para la escritura de este tipo de archivos. En primer lugar, se importa la librería necesaria. Posteriormente, con los comandos `ET.Element` y `ET.SubElement` se crean las etiquetas y subetiquetas que aparecen en la Figura 3.3. El siguiente comando escribe los valores de las coordenadas que definen cada segmento como cadena de caracteres. Por último, se incluyen los comandos necesarios para cerrar la edición del archivo y guardarlo en el directorio deseado.

```
import xml.etree.cElementTree as ET

xmlMesa = ET.Element("MESA")
xmlSecciones = ET.SubElement(xmlMesa,"SECCIONES")
xmlIDSeccion = ET.SubElement(xmlSecciones, "seccionΓ
str(contador_lineas_detectadas))

xmlSegmento = ET.SubElement(xmlIDSeccion, "segmentoΓ
str(k), X1= str(x1), Y1= str (y1), X2= str(x2), Y2= str
(y2))
arbol = ET.ElementTree(xmlMesa)
arbol.write("C:\\Users\\Nacho\\Desktop\\secciones.xml")
```

Fragmento de código 10: Comandos para la creación de un archivo 'xml'.

4.3.5. Intersecciones

Aunque la información necesaria de las intersecciones se recoge de la primera imagen, se ha querido explicar primero todo el proceso necesario asociado al reconocimiento de las secciones para facilitar la comprensión al lector. Sin embargo, a la vez que se aplica una máscara de *color rojo* a la Figura 4.5, se debe aplicar otra máscara de *color verde* para extraer las intersecciones. En la Figura 4.21 se muestra el resultado de este proceso.

Al contrario de lo que ocurría en el caso de las secciones, para definir una intersección no se necesita un registro ordenado de coordenadas, sino tan solo las coordenadas del centro de la intersección. Para ello, la técnica utilizada cambia. En lugar de utilizar un algoritmo que registra todos los puntos del contorno que se muestra en la Figura 4.21, se emplea un comando que permite rodear todos los píxeles blancos con una circunferencia de radio mínimo. El comando utilizado se muestra en el Fragmento de código 11. Los resultados gráficos se muestran en la Figura 4.22.

Como se puede apreciar en el Fragmento de código 11, el comando utilizado devuelve las coordenadas del centro de la circunferencia dibujada. La detección de las coordenadas es un paso clave, porque de nuevo, se deben cruzar estos datos con

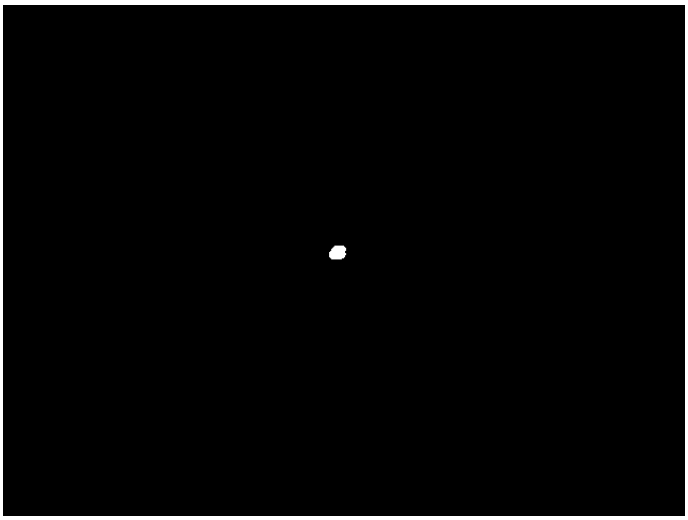


Figura 4.21: La máscara de *color verde* aplicada sobre la Figura 4.5 permite extraer el conjunto de píxeles que representan las intersecciones, en este caso, sólo aparece una intersección.

```
((x,y), radio) = cv2.minEnclosingCircle(contorno)
```

Fragmento de código 11: Comando que permite rodear un contorno con una circunferencia de radio mínimo.

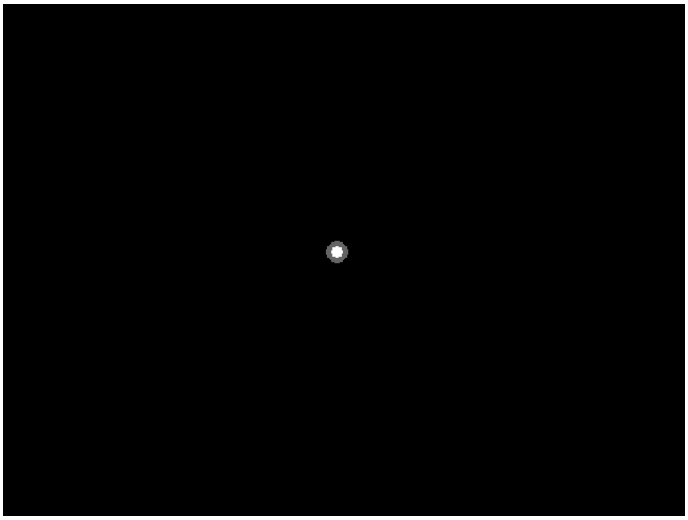


Figura 4.22: Los píxeles de la Figura 4.21 marcados en blanco se engloban en una circunferencia —color gris— de radio mínimo que permite contenerlos a todos en su interior.

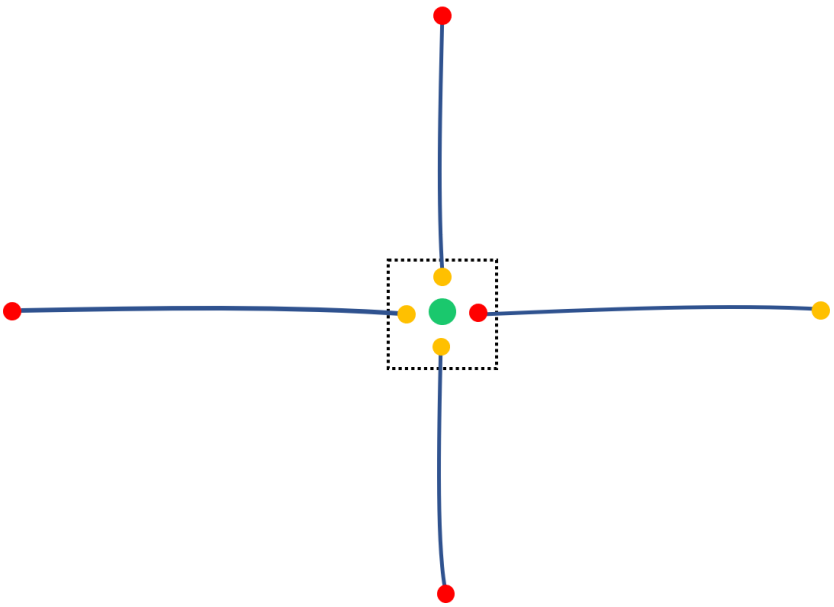


Figura 4.23: Las secciones cuyos extremos —entrantes o salientes— estén en contacto con la intersección, son registrados en el archivo 'xml' con la información de las intersecciones.

las coordenadas de los puntos extremos de las secciones. Como ya se ha visto en secciones anteriores, pueden diferenciarse dos tipos de extremos: aquellos desde los que sale el flujo de movilidad y aquellos hacia los que se dirige el flujo.

En la Figura 4.23 se muestra cómo trabaja el algoritmo. Alrededor de la intersección se incluye su área de influencia, es decir, todo extremo que se encuentre en ese área, será registrado. Los extremos de color rojo representan el extremo inicial —entrada— de cada sección. Los extremos amarillos representan el final. El ejemplo utilizado representa el mismo escenario que ha ilustrado todo este capítulo. Los resultados del registro se guardan en el archivo '.xml' mostrado en la Figura 3.4. Como se puede apreciar en ese archivo '.xml', se registran tres secciones entrantes —sección de la izquierda, sección superior y sección inferior—, y una sola sección saliente —la sección de la derecha—.

Capítulo 5

Detección de piezas

En este capítulo de la memoria se explica cómo se realiza el reconocimiento de las piezas de Lego. Para ello, primero se definen los tipos de edificios o enclaves urbanos que se pueden representar en esta versión del proyecto. Posteriormente, se aportan detalles sobre el proceso de entrenamiento y el uso del sistema de detección de objetos YOLO v3. Posteriormente, se incluyen las explicaciones necesarias para comprender el algoritmo que permite realizar el seguimiento de objetos. Finalmente, en la última sección de este capítulo se incluyen algunas observaciones generales del mismo.

5.1. Tipos de edificios y enclaves urbanos

Como se menciona en la introducción de esta memoria, la detección de piezas es necesaria para poder asociar a cada pieza un edificio o parte de la ciudad. Cada enclave detectado se convierte así en un punto de generación o atracción de movilidad en función de su tipología y tamaño.

Una ciudad está compuesta por el cúmulo casi infinito de edificios y enclaves urbanos de diferentes tipos y funciones. Sin embargo, en una primera aproximación, y sin realizar particularizaciones muy profundas, en este proyecto se distinguen un total de 19 entidades diferentes. Se han ordenado en seis grupos que permiten su clasificación según la función que desempeñan dentro de la ciudad:

- **Vivienda**. Torre de viviendas y hotel.
- **Comercial**. Supermercado y centro comercial.
- **Laboral**. Industria y torre de oficinas.
- **Educativo**. Colegio/universidad, biblioteca y museo.
- **Ocio**. Restaurante, gimnasio, bar, teatro/cine y estadio.
- **Público**. Hospital, parada de autobús, estación, aparcamiento y parque/zona verde.

5.2. Exploración de estrategias

El algoritmo o técnica de reconocimiento de piezas debe cumplir algunos requisitos para que esta tarea del proyecto ofrezca unos resultados adecuados.

- **Nº de piezas reconocibles**. Para dotar de flexibilidad al proyecto, el algoritmo implementado debe ser capaz de detectar 19 tipos de entidades, lo que implica detectar 19 agrupaciones de piezas de Lego diferentes.
- **Robustez**. Se requiere que el sistema de detección de piezas ofrezca la menor sensibilidad posible a cambios en las condiciones de iluminación.
- **Velocidad**. De forma paralela a la ejecución de este bloque, se debe representar en Minecraft el escenario virtual. Es decir, al mismo tiempo que los usuarios colocan las piezas sobre la mesa, en Minecraft deben aparecer los edificios que estas piezas representan. Para que el espacio de tiempo consumido entre entre ambas acciones —detección y representación virtual— no sea excesivo, es fundamental que el sistema de detección de objetos procese la imagen y extraiga las detecciones de manera rápida.

En las primeras pruebas e investigaciones, la técnica utilizada se basa en la diferenciación de color. Es decir, aprovechar la amplia gama de colores que ofrece Lego para asociar un tipo de edificio a un color específico, distinguiendo por ejemplo en la familia de *color azul* entre los tonos más oscuros y aquellos más claros, como se muestra en la Figura 5.1. Para realizar estas operaciones, la técnica utilizada vuelve a basarse en la librería OpenCV y la aplicación de máscaras de colores. Sin embargo, esta estrategia encuentra limitaciones bastante contundentes:

- **Detección de colores**. El uso de rangos para la definición de colores —tal y como se muestra en el Fragmento de Código 4— muestra sus limitaciones a la hora de distinguir tonalidades muy próximas entre sí. De esta forma, aplicando una máscara de color a las piezas de la Figura 5.1 para detectar la pieza más oscura, se puede observar cómo también aparecen píxeles con esta tonalidad en las piezas más claras —en sus zonas de sombra—. O viceversa, también aparecen tonos claros en las piezas oscuras —en sus zonas más iluminadas—. Como consecuencia de este comportamiento, la gama de colores correctamente detectables queda reducida a aquellos colores con tonalidades claramente distinguibles: rojo, amarillo, verde, azul y magenta.
- **Dependencia de la iluminación**. La estrategia explorada, además de ofrecer limitaciones a la hora de distinguir tonalidades de colores próximas —cambio de tonalidad en función de la iluminación recibida—, también muestra otra desventaja asociada a la iluminación. En función del tipo de iluminación utilizada —bombilla incandescente, LED, iluminación natural, etc.— el algoritmo puede llegar a mostrar falsos positivos al detectar regiones con tonalidades amarillentas, azuladas, etc. sobre el fondo idealmente blanco sobre el que se disponen las piezas de Lego. Este problema, sin embargo, puede corregirse incluyendo un balance de blancos.

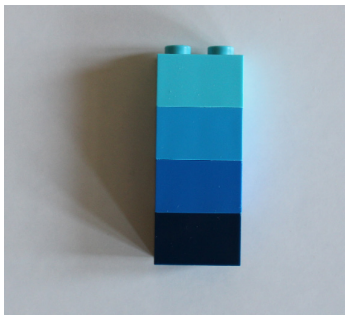


Figura 5.1: La identificación de piezas basada exclusivamente en la diferenciación por color puede ofrecer errores cuando se presentan piezas de tonos similares.

Aunque la velocidad de procesamiento de esta estrategia es más que suficiente para ofrecer un número fotogramas por segundo adecuado para la percepción de movimiento, las limitaciones descritas implican dos consecuencias que hacen que sea descartada. En primer lugar, el número de entidades —u objetos— distinguibles se reduce a cinco, es decir, un color para cada tipo de objeto hasta completar cinco tipo de edificios diferentes. Por otro lado, la necesidad de controlar de manera exhaustiva las condiciones de iluminación, hacen que la estrategia sea poco robusta.

De esta forma, se concluye que es necesario implementar un sistema más robusto, que no se vea tan afectado por las condiciones de iluminación y que ofrezca un método para reconocer un amplio espectro de objetos o entidades que puedan ser asociados a los edificios o enclaves urbanos. La estrategia que se explora y que finalmente se implementa en esta versión del proyecto es el sistema de detección de objetos YOLO en su versión 3.

5.3. Entrenamiento del modelo para la detección de piezas. YOLO

Como ya se ha mencionado en otros capítulos de esta memoria, YOLO es un sistema de detección de objetos que permite la localización de entidades empleando un menor tiempo que otros métodos basados en redes neuronales y ofreciendo una fiabilidad de detección elevada. YOLO implementa una red neuronal con 24 capas convolucionales seguidas de dos capas completamente conectadas. Al introducir una imagen en un modelo entrenado, YOLO la divide con una malla regular cuadrada. Al mismo tiempo predice contornos con los que delimita las posibles detecciones y les asocia una probabilidad de que la detección realizada sea una de las clases del modelo entrenado [Far16]. Para poder emplear su potencial en el proyecto, es necesario realizar varias tareas antes de implementarlo en el algoritmo de Python.

- **Preparación del conjunto de datos para el entrenamiento**
- **Entrenamiento del modelo en Google Colab**
- **Descarga e integración del modelo**

Para utilizar YOLO, se puede hacer uso de modelos en los que ya se ha realizado el entrenamiento. Esta opción puede resultar una gran ayuda para aquellos proyectos en los que resulta necesario localizar objetos comúnmente utilizados, por ejemplo, personas, coches, motos, etc. [Tan18]. Otra opción útil y que permite ahorrar esfuerzos a la hora de implementar YOLO en un proyecto, es la utilización de bases de datos —imágenes— en los que los objetos ya están localizados. Esto permite utilizar las bases de datos de otros proyectos que pueden resultar útiles para localizar un objeto determinado y no preparar una base de datos propia [Con20].

Sin embargo, en este proyecto, al ser necesario localizar objetos muy particulares —piezas de Lego con códigos de colores específicos—, es necesario realizar todos los pasos mencionados. Para ello, se han utilizado como referencia dos lecciones de YouTube enfocadas a la aplicación de YOLO v3 y el entrenamiento de modelos propios. Aunque en los siguientes párrafos se intenta transmitir los puntos clave para ejecutar correctamente el entrenamiento, se recomienda al lector interesado en implementar un modelo de este tipo consultar las dos lecciones para más detalles. En el primero de los vídeos —[Pys20]— se explica cómo implementar un modelo para la detección de un único objeto personalizado y la creación de una base de datos propia con imágenes etiquetadas. En el segundo vídeo —[Guy20a]— se explican los pasos necesarios para entrenar un modelo capaz de detectar varios objetos personalizados. La suma de ambas lecciones —creación de una base de datos propia con imágenes etiquetadas y el entrenamiento de un modelo capaz de detectar más de un único objeto propio— permite obtener un modelo adecuado para este proyecto.

5.3.1. Preparación del conjunto de datos para el entrenamiento del modelo

En primer lugar, es necesario crear una base de datos con imágenes en las que aparezcan los objetos que deben ser reconocidos, es decir, los 19 grupos de piezas que representan los enclaves y edificios de la ciudad.

En la Figura 5.2 se muestran los códigos de colores que han sido elegidos para representar cada edificio o espacio urbano. Nuevamente, aparecen ordenados según su función dentro de la ciudad. Los patrones de colores seleccionados se forman a partir de cinco colores: azul, amarillo, rojo, negro y puntualmente, verde.

Como se puede apreciar, todos los patrones contienen una pieza roja en su esquina inferior derecha. Esto, además de evitar que en los patrones aparezca un sólo color —lo que podría conducir a falsos positivos por la detección de piezas individuales dentro de un código—, permite reducir el número de imágenes que componen el conjunto de datos base para el entrenamiento. La marca roja de cada código indica al usuario la orientación con la que debe colocar cada patrón: la esquina inferior derecha de cada conjunto de piezas siempre debe ser de color rojo. Por lo tanto, no hace falta incluir en el conjunto de entrenamiento imágenes que incluyan piezas con diversas orientaciones. Sólo es necesario incluir los códigos de colores con la orientación que se muestra en la Figura 5.2. Si en versiones futuras

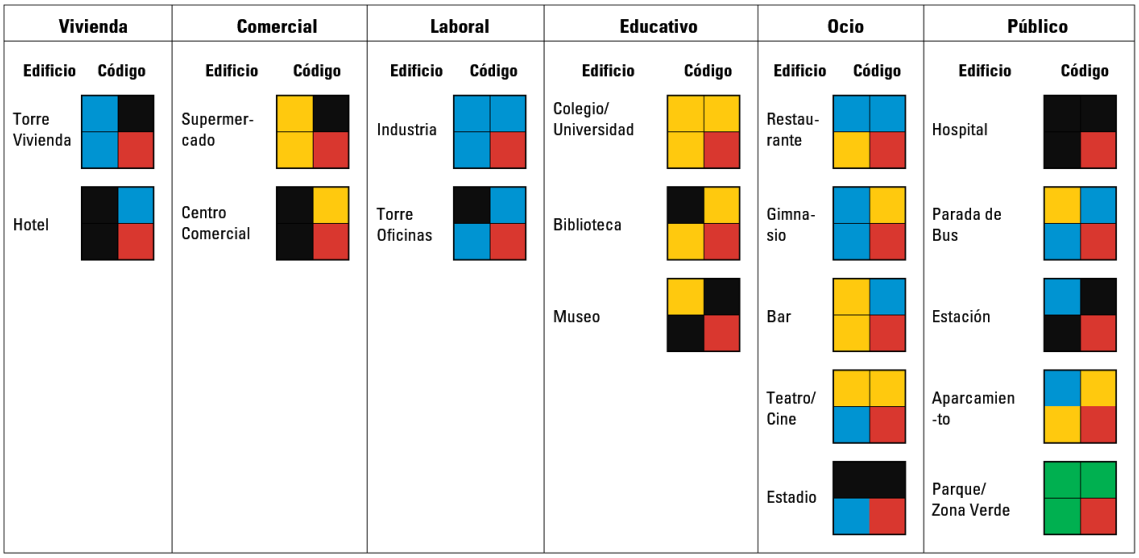


Figura 5.2: Los patrones de colores elegidos permiten localizar y distinguir un total de 19 entidades urbanas.

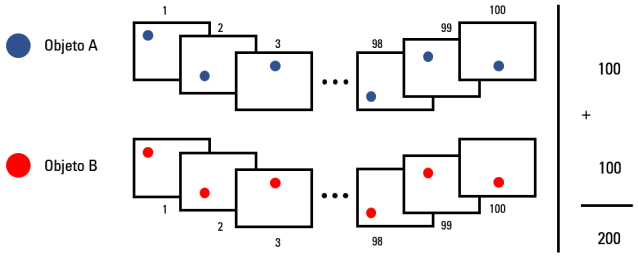


Figura 5.3: Para el conjunto de imágenes de entrenamiento, se recomienda utilizar 100 imágenes por cada objeto. Es decir, en el ejemplo mostrado, se deben incluir 100 imágenes donde aparezca el *Objeto A* —azul— y otras 100 imágenes donde aparezca el *Objeto B* —rojo—, sumando un total de 200 imágenes.

del proyecto se quisiera dotar de mayor libertad a los usuarios, permitiéndoles posicionar las piezas con cualquier orientación, entonces sería necesario incluir en el conjunto de datos de entrenamiento imágenes con todas las orientaciones posibles para cada patrón de colores.

A la hora de elaborar el conjunto de imágenes de entrenamiento, se recomienda utilizar al menos cien imágenes diferentes —[Pys20]— por cada objeto que se desee reconocer. Es decir, se cumple lo mostrado en la Figura 5.3.

Si en todas las imágenes aparecen todos los objetos que se desean reconocer, entonces el número queda reducido —tal y como se muestra en la Figura 5.4—. Esto es lo que ocurre en el modelo entrenado finalmente para esta versión del proyecto, que cuenta con un conjunto de 100 imágenes de entrenamiento en las que aparecen todos los patrones de colores. Tras varios ensayos, se concluye que los resultados obtenidos son satisfactorios para el número de imágenes incluidas.

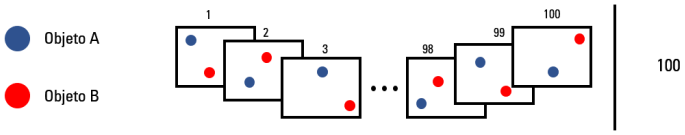


Figura 5.4: Incluir todos los objetos en una misma imagen permite reducir su número en el conjunto de entrenamiento. Esto permite ahorrar tiempo en el proceso de elaboración de la base de datos.

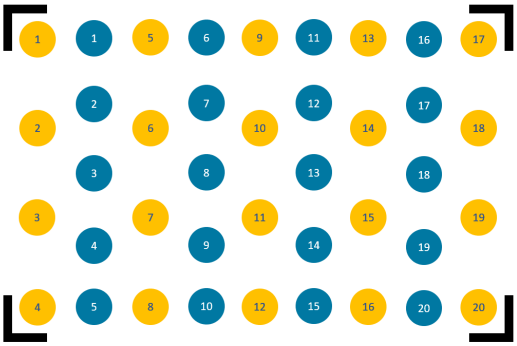


Figura 5.5: La plantilla muestra dos matrices, una de 4x5 posiciones y otra de 5x4. En total 40 posiciones. Al haber 19 piezas, se necesitan menos de 80 imágenes para que en total cada patrón haya pasado por las 40 posiciones marcadas. Las fotos restantes hasta completar la centena se realizan colocando las piezas en posiciones intermedias a las marcadas.

En ellas, aparecen todos los patrones de colores orientados de la manera deseada y repartidos por toda la superficie de la mesa.

Para cumplir con las especificaciones del entrenamiento, se crea una plantilla que permite repartir las piezas por toda la superficie hábil. Las posición de cada patrón se cambia de forma ordenada tras cada foto, lo que permite que en el conjunto total de imágenes, cada patrón aparezca en todas las posiciones de la mesa. En la Figura 5.5 se muestra la plantilla que se ha utilizado para extraer las imágenes del entrenamiento. Esta técnica permite completar la tarea de obtención de imágenes de una forma ordenada, garantizando que cada patrón de colores aparece en todas las zonas de la mesa y ahorrando tiempo.

En la Figura 5.6 se muestra una imagen tomada con cámara Intel del proyecto. Como se puede apreciar, todos las patrones están orientados de la forma descrita en los párrafos anteriores. Además, en el entrenamiento se incluyen veinte patrones —en lugar de los 19 descritos en la Figura 5.2— con el objetivo de que no dejar ninguna posición de las plantillas vacía. Para ello, se puede repetir cualquiera de los patrones, en este caso el dedicado a *Industria* aparece en dos ocasiones —en la esquina superior derecha—.

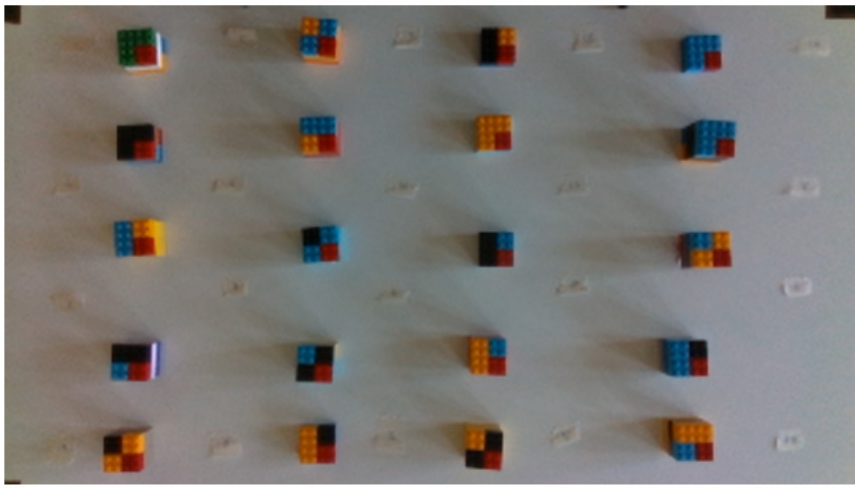


Figura 5.6: Imagen tomada para el entrenamiento del modelo. Los patrones aparecen correctamente orientados y dispuestos según la plantilla de color azul de la Figura 5.5.

Una vez realizadas todas las fotos necesarias para entrenar el modelo, se debe dar paso al proceso de etiquetado. Con el etiquetado se extraen las regiones de interés de cada foto —los patrones de colores— y se les añade información, es decir, se indica el tipo de objeto que aparece en la imagen. Este es un proceso tedioso y que requiere una inversión de tiempo considerable, pues el número total de etiquetas que deben incluirse en el conjunto de datos de entrenamiento atiende a la Ecuación 5.1. En esta ecuación, i es el número de imágenes y n es el número de patrones de colores por imagen —al que hay que sumarle el patrón repetido—.

$$\text{Etiquetas} = i \cdot (n+1) \rightarrow 100 \cdot (19+1) = 2000 \tag{5.1}$$

El proceso de etiquetado puede realizarse utilizando el *software LabelImg*. Para realizar la descarga de este *software* y configurarlo adecuadamente, se recomienda seguir la primera de las lecciones de YouTube mencionadas anteriormente —[Pys20]—. En la Figura 5.7 se muestra la misma imagen que en la Figura 5.6, pero con los patrones seleccionados y asociados a su etiqueta correspondiente.

5.3.2. Entrenamiento del modelo en Google Colab

Una vez se ha elaborado el conjunto de imágenes de entrenamiento con sus respectivas etiquetas, se puede dar paso al entrenamiento del modelo en Google Colab. La ventaja fundamental de realizar el entrenamiento en esta plataforma es hacer uso de una GPU, que permite realizar el entrenamiento de forma mucho más rápida que lo que permitiría una CPU convencional. Este servicio, aunque tiene una limitación de horas consecutivas de uso, es totalmente gratuito, y se recomienda seguir las instrucciones de la segunda lección de YouTube mencionada —[Guy20a]—,

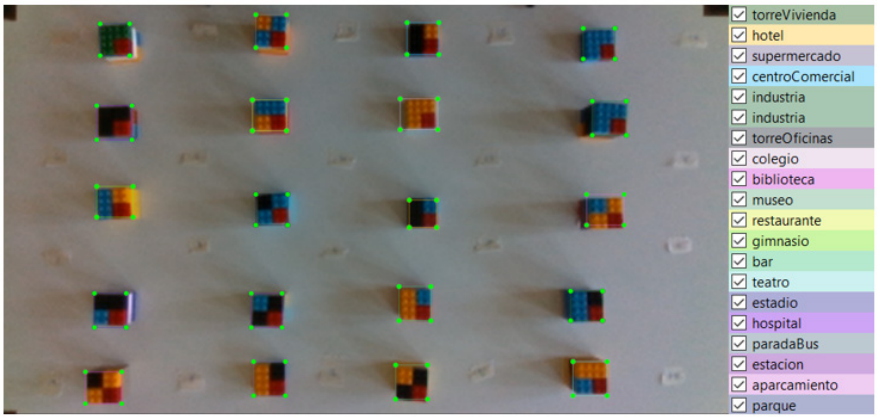


Figura 5.7: Los patrones mostrados en la Figura 5.6 aparecen marcados en el *software LabelImg*. Cada patrón tiene su etiqueta en función de la entidad que representa. Las etiquetas se muestran en la parte derecha de la figura.

donde se puede encontrar el código necesario para realizar el entrenamiento en línea. Para entrenar el modelo deseado, es necesario incluir algunos archivos que describen sus características. Estos archivos pueden cargarse en Google Drive y ser leídos desde Google Colab gracias a que se puede establecer comunicación entre ambas plataformas. Es importante respetar el nombre de los archivos que se indican, pues de otra forma, sería necesario implementar cambios en el código para poder localizarlos. Para más información sobre el tipo de archivos que se describe a continuación, se recomienda visitar el repositorio de GitHub del mismo autor de la lección de YouTube mencionada. En este repositorio —[Guy20b]— se pueden encontrar los archivos utilizados por el autor para su propio proyecto, por lo que se necesitarían incluir las modificaciones necesarias para adaptarlo al proyecto deseado.

- **Archivo 'obj.zip'**. Cuando se finaliza el trabajo con LabelImg, cada imagen tiene asociado un archivo '.txt' donde aparecen los datos y coordenadas de las etiquetas realizadas en cada imagen. Es decir, para un total de 100 imágenes, deben generarse 100 archivos. En el archivo 'obj.zip' se deben incluir todas las imágenes y todos los archivos '.txt' generados automáticamente en el proceso de etiquetado.

- **Archivo 'obj.names'**. Es un archivo de texto —puede crearse utilizando el bloc de notas— donde se incluye en cada renglón el nombre de un objeto que se desee localizar. En el caso de este proyecto, se incluyen los nombres que se muestran a la izquierda de la Figura 5.7. En el repositorio —[Guy20b]— se puede encontrar un ejemplo de este tipo de archivo.

- **Archivo 'obj.data'**. En un archivo de texto donde se incluye información relacionada con los directorios de carpetas útiles para el entrenamiento. Por ejemplo, para el caso del archivo anterior, se indica su directorio como: *names*

= *data/obj.names*. Para más detalles sobre el contenido de este archivo, se recuerda que se puede recurrir al repositorio mencionado anteriormente —[Guy20b]— donde se puede encontrar un ejemplo de este tipo de archivo.

- **Archivo 'yolov3_custom.cfg'**. Incluye las especificaciones técnicas del entrenamiento: número de objetos que se desean detectar, número de filtros, etc.. En la lección de YouTube —[Guy20a]— se dan instrucciones precisas sobre las modificaciones que deben realizarse sobre este archivo según el número de objetos que se deseen identificar. Nuevamente, en el repositorio mencionado —[Guy20b]—, se puede encontrar un ejemplo.
- **Archivo 'generate_train.py'**. Es un archivo de Python que permite generar de manera automática un archivo que recoge todos los directorio de todas las imágenes del entrenamiento cargadas en Google Colab. Es importante revisar que los directorios que se indican en este archivo reflejan de manera correcta la localización de cada archivo que utiliza, pues de otra forma, como es lógico, aparecerán errores. También es importante tener en cuenta el tipo de archivo de las imágenes. Es decir, el archivo 'generate_train.py' puede esperar imágenes en formato '.jpg' mientras que las imágenes pueden ser '.png'. Simplemente modificando el código, este problema de coordinación queda resuelto. En el repositorio —[Guy20b]— se puede encontrar un ejemplo de este tipo de archivo.

Además de crear e incluir los archivos mencionados en una cuenta de Google Drive —para estar a disposición del entrenamiento en Google Colab—, resulta necesario crear una carpeta donde guardar los archivos que genera automáticamente el entrenamiento. Cada 1000 iteraciones, el algoritmo ejecutado en Google Colab genera y guarda un archivo llamado *yolov3_custom_n.weights*, donde n es el número de iteraciones ejecutadas. En este proyecto, se ha comprobado que es suficiente con alcanzar las 2000 iteraciones para conseguir unos resultados satisfactorios.

5.3.3. Descarga e integración

Una vez se ha entrenado el modelo de detección, se puede dar paso a su integración en el algoritmo del proyecto. Para ello, se recomienda seguir una nueva lección de YouTube, en la que se explica cómo utilizar YOLO v3 para detectar objetos con un modelo previamente entrenado [Pys19].

Para poder cargar el modelo entrenado en el proyecto, se vuelve hacer uso de la librería *OpenCV*. Se deben incluir dos archivos para leer correctamente la red neuronal. El primero de ellos es el archivo que genera el algoritmo de entrenamiento mencionado en la sección anterior —*yolov3_custom_n.weights*— y que incluye los pesos que afectan a la red neuronal. El segundo, es el archivo *yolov3_custom.cfg*, es decir, el archivo que incluye las especificaciones técnicas del modelo entrenado —descrito en los párrafos de la sección anterior—. En el Fragmento de código 12 se muestra cómo realizar este proceso.

Una vez cargado el modelo, es importante utilizarlo de una manera eficiente. Cuando se envía una imagen al modelo para que la procese, éste es capaz de extraer

```
self.net = cv2.dnn.readNet("C:\\Users\\Nacho\\Desktop\\Modelo\\
yolov3_custom_last.weights", "C:\\Users\\Nacho\\Desktop\\Modelo\\
yolov3_custom.cfg")
```

Fragmento de código 12: Comandos para cargar el modelo entrenado.

todas las clases —objetos— que se encuentran en la imagen y devolver la misma imagen con los objetos localizados, recuadrados y etiquetados.

Sin embargo, debido a la arquitectura del sistema y al algoritmo de seguimiento de piezas —descrito en la siguiente sección de este capítulo—, se requiere generar, para cada tipo de patrón, una imagen diferente. Es decir, al existir en el modelo un total de 19 patrones o códigos de color distintos, es necesario generar —a partir de la imagen examinada— 19 imágenes diferentes, incluso de aquellos patrones que no aparezcan en la imagen inicial.

En la Figura 5.8 se muestra un ejemplo del tipo de imágenes que genera el algoritmo. A partir de la imagen inicial que procesa el modelo —imagen superior izquierda—, se generan 19 imágenes binarias. Para aquellos tipos de patrón que no aparecen en la imagen inicial, se genera un imagen completamente negra —imagen superior derecha—. En cambio, para aquellos patrones que sí aparecen en la imagen inicial —restaurante e industria—, se genera una imagen en la que los patrones aparecen representados por un cuadro blanco. El área del cuadro blanco corresponde al área del patrón detectado y respeta sus coordenadas. Para el ejemplo mostrado en la Figura 5.8, se puede apreciar cómo en la imagen inferior izquierda se muestra la imagen correspondiente al patrón de restaurante, mientras que en la imagen inferior derecha se muestra la imagen del patrón asociado a industria. Cabe tener en cuenta, que en el caso de detectarse varios patrones de un mismo tipo, la imagen generada para ese tipo de patrón contendría un número de marcas blancas igual al número de patrones detectados, manteniendo las propiedades explicadas —área y posición—.

Como se puede apreciar, al generar las imágenes binarias mostradas en la Figura 5.8, se sigue la misma estrategia que en el caso explicado en el Capítulo 4, es decir, extraer de una imagen inicial el contenido que es relevante de ella y aislarlo para trabajar con la información que aporta. La única diferencia es el tipo de herramienta utilizada para conseguir las imágenes binarias. En el capítulo dedicado a la detección de líneas, se utilizan máscaras de colores. En la detección de patrones de colores, como se ha mostrado en este capítulo, utiliza un modelo de redes neuronales entrenado expresamente para detectar los 19 patrones diferentes de piezas de Lego.

5.4. Seguimiento de piezas

Una vez se cuenta con las 19 imágenes que posibilitan seleccionar y clasificar la información de los patrones localizados sobre la mesa, se da paso a la explicación

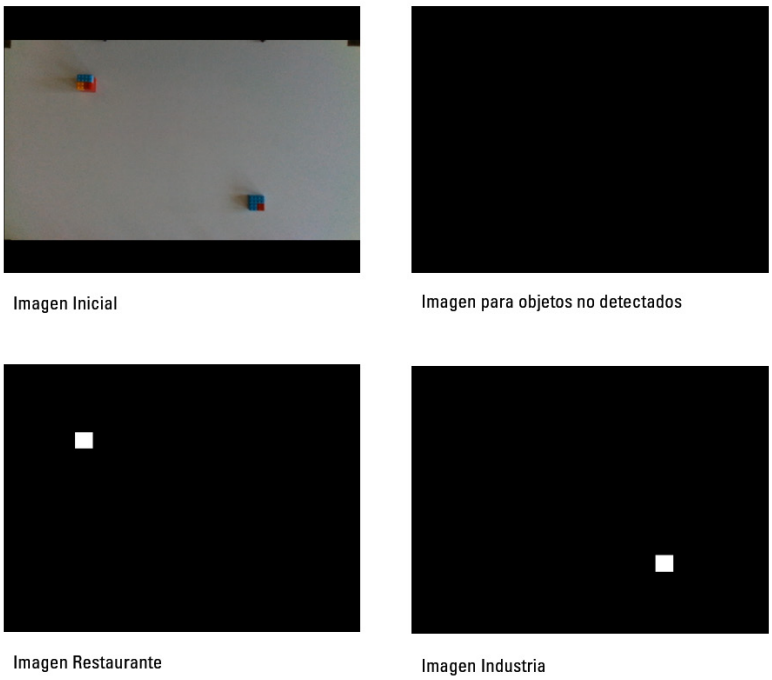


Figura 5.8: El algoritmo implementado genera una imagen para cada tipo de patrón detectado. Las imágenes generadas son binarias e incluyen un cuadro blanco allí donde se detecta un patrón. En consecuencia, cuando no se detecta ningún patrón, la imagen no incluye ningún cuadro blanco.

del algoritmo que realiza el seguimiento de las mismas. Con este algoritmo, se permite distinguir y seguir fotograma a fotograma cualquier patrón detectado, tal y como se muestra en la Figura 5.9. En la imagen superior izquierda se muestran tres patrones que representan tres aparcamientos. Cada uno de ellos está etiquetado con su nombre y número —1, 2 y 3— que lo identifica de forma física y virtual. Es decir, existe una etiqueta física en la que se ha escrito un número del 1 al 3 con la que se identifica a cada patrón de color. Al mismo tiempo, el algoritmo incluye una etiqueta virtual que añade en los fotogramas que devuelve por pantalla, de tal forma que el número de la etiqueta física y de la etiqueta virtual coinciden. En las imágenes sucesivas, correspondientes a instantes de tiempo posteriores, se muestra cómo pese al cambio de posición, se mantiene la correspondencia entre los números de las etiquetas físicas y virtuales.

Pese a que esta función del proyecto no repercute en la simulación en Aimsun ni en los resultados obtenidos —no genera ningún tipo de dato relevante para la simulación—, sí facilita el proceso de depurado del algoritmo, pues muestra en tiempo real las detecciones y sus propiedades. Además de facilitar el proceso de depuración, la función que se describe en los párrafos siguientes puede ser útil para aportar realimentación a los usuarios en tiempo real y corroborar que las piezas que disponen sobre la mesa son reconocidas correctamente —permite al usuario comprobar que el patrón de color que coloca corresponde realmente al tipo de edifico que quiere incluir, permite comprobar que la orientación con la que se coloca el patrón es la adecuada, etc.—.



Figura 5.9: El algoritmo de seguimiento de patrones fotograma a fotograma permite localizar y distinguir entre sí patrones iguales.

Además, la posibilidad de seguir fotograma a fotograma los patrones de colores puede tener un gran potencial para futuros desarrollos del proyecto. Entre las funciones de Aimsun, se encuentra la posibilidad de realizar simulaciones en las que se pueden modificar los elementos que componen un escenario. Es decir, mientras se realiza una simulación, ésta se interrumpe momentáneamente para cambiar la ubicación de los elementos y valorar los resultados que implican las modificaciones en el modelo —partiendo de un nuevo estado inicial condicionado por los resultados previos—.

Una vez justificado el interés de implementar esta función —que a primera vista no aporta un valor añadido diferencial—, se da paso a la descripción del algoritmo que permite realizar el seguimiento de objetos. Cabe mencionar que el algoritmo que se implementa en este proyecto tiene como referencia las explicaciones de una lección de seguimiento de objetos aplicado al reconocimiento facial —[Ros18]—. En los párrafos siguientes se incluyen los pasos necesarios para implementar y aplicar dicho algoritmo a este proyecto, mientras que se recomienda acudir a la lección mencionada para encontrar los detalles más técnicos referidos al código.

Un algoritmo de seguimiento de objetos fotograma a a fotograma necesita o incluye varios requisitos. En el caso de este proyecto, se mencionan a continuación:

- **Detecciones de patrones**. El algoritmo que se explica en la lección mencionada [Ros18] incluye un algoritmo de detección facial. En el caso del proyecto de esta memoria, los datos de entrada al algoritmo son las imágenes binarias mostradas en al Figura 5.8.
- **Asignación de un ID inmutable a cada patrón**. Cada patrón detectado debe contar con su propio número identificativo.

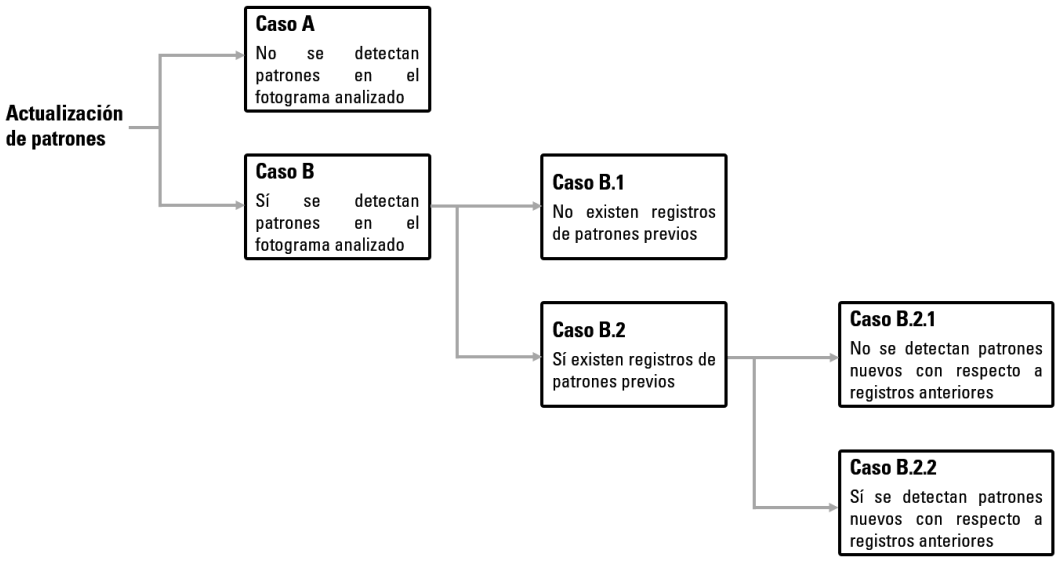


Figura 5.10: En el esquema se muestran los diferentes escenarios que pueden plantearse a la hora de actualizar la información del estado de los patrones.

- **Seguimiento de los patrones fotograma a fotograma**. Durante la evolución del vídeo, se debe realizar el seguimiento de los patrones manteniendo su ID inicial.

Para conseguir completar los requisitos mencionados en los puntos anteriores, es necesario que el algoritmo implemente un mecanismo para actualizar el estado de los patrones. Esta función constituye el corazón del algoritmo al recibir la información de las detecciones de patrones fotograma a fotograma y se encargarse de su gestión. En la Figura 5.10 se muestra un esquema de los diferentes escenarios que se pueden plantear.

Caso A: No se detectan patrones en el fotograma analizado . Este escenario puede aparecer cuando el proceso de diseño acaba de comenzar —no se han introducido piezas— o cuando después de disponer varios patrones, éstos se retiran por cualquier circunstancia del proceso de diseño de la maqueta. En la primera opción, no existe ningún registro de IDs de patrones previo —pues la fase de diseño con patrones acaba de comenzar—, por lo que no es necesario realizar ninguna acción. En el segundo caso, existe un registro de IDs de patrones que han aparecido previamente, por lo que es necesario implementar un contador que registre el número de fotogramas transcurridos desde la última vez que apareció cada patrón. Cada patrón debe contar con su propio contador de fotogramas desaparecido asociado a su ID.

En la Figura 5.11 se muestran gráficamente la primera opción descrita en el punto anterior. Se muestran dos fotogramas: el primero, tomado en el instante *t–1*, muestra la mesa vacía sin patrones. El segundo, tomado inmediatamente después, en el instante *t*, muestra también la mesa vacía. Ante la falta de registros de IDs

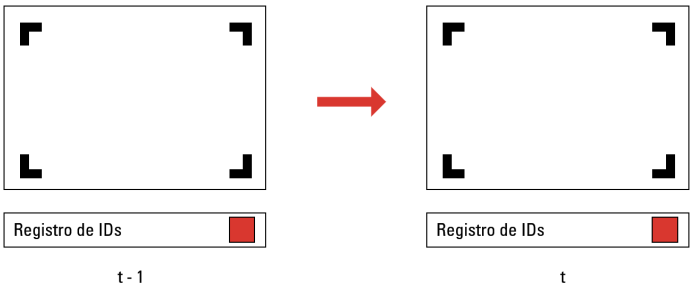


Figura 5.11: Caso A. En ambos instantes de tiempo —*t–1* y *t*— no aparecen patrones en los fotogramas. Al no existir un registro de IDs previo, el algoritmo no realiza ninguna acción. El cuadro del registro de IDs en color rojo señala que no existe un registro.

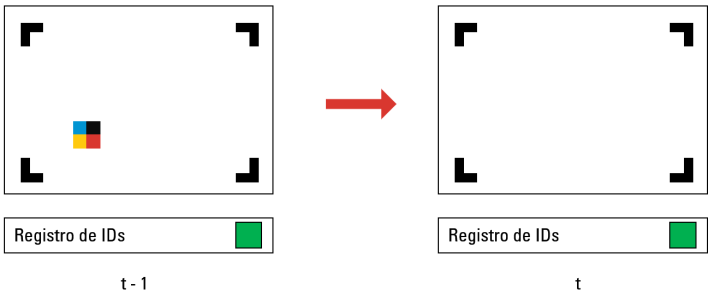


Figura 5.12: Caso A. Al existir un registro de IDs previo y no detectarse ningún patrón en el fotograma actual —t—, el algoritmo comienza a contar el número de fotogramas consecutivos en los que la figura no aparece. Si se excede un límite de fotogramas fijado, la figura queda eliminada del registro y se considera que ha desaparecido. El cuadro del verde indica que sí ha registro.

previos y de nuevas detecciones, el algoritmo no realiza ninguna acción.

Por otro lado, en la Figura 5.12, se muestra cómo en el instante de tiempo *t–1* aparece un patrón. Éste se detecta correctamente, por lo que existe un registro con su ID asociado. En el instante de tiempo *t*, el patrón desaparece de la escena, por lo que se activa su contador de fotogramas consecutivos desaparecido. Este contador, que es particular para cada una de las figuras detectadas, incrementa en una unidad por cada fotograma analizado en el que no aparezca la pieza asociada al ID. De esta forma, si se excede un número predefinido de fotogramas sin que aparezca de nuevo la pieza, se da por desaparecida y se borra su ID del registro. Como generalmente el número de fotogramas necesarios para borrar del registro un ID es superior a 0, en el instante *t* el registro de IDs todavía permanece vigente, tal y como se muestra en la Figura 5.12.

Caso B: Sí se detectan patrones en el fotograma analizado . Cuando sí se detectan patrones en el fotograma, la casuística queda dividida en dos opciones: Caso B.1 y Caso B.2.

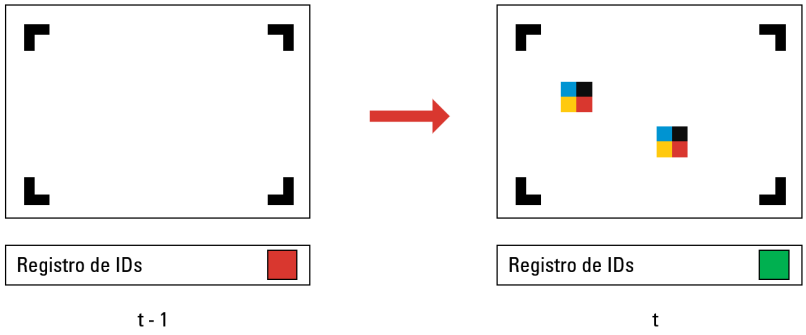


Figura 5.13: Caso B.1. Al no existir un registro de IDs previo, cuando se detectan patrones por primera vez se debe crear dicho registro en el instante *t*.

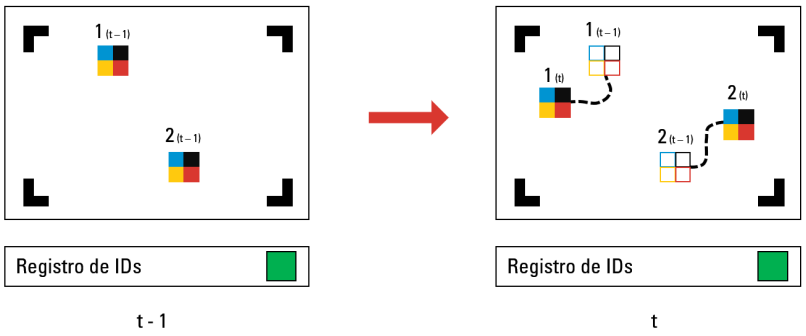


Figura 5.14: Caso B.2.1. En este tipo de escenarios, el algoritmo debe ser capaz de realizar correctamente la correspondencia entre los dos fotogramas analizados.

- **Caso B.1: No existen registros de IDs de patrones previos**. Cuando no existe un registro de patrones previos, el algoritmo se limita a registrar los nuevos patrones detectados. Un ejemplo de este tipo de escenarios se muestra en la Figura 5.13. En el instante *t–1* se muestra cómo no existe un registro de IDs y al mismo tiempo no se detectan figuras sobre la mesa. Por otro lado, en esa misma figura, en el instante *t* sí se detectan patrones, y por lo tanto, se crea un registro de IDs.

- **Caso B.2: Sí existen registros de IDs de patrones previos**. Cuando sí se detectan patrones en el fotograma analizado, y además, sí existen registros de patrones previos, de nuevo aparecen dos casos: Caso B.2.1 y Caso B.2.2.

- **Caso B.2.1: No se detectan patrones nuevos con respecto a los registros anteriores** Cuando existe un registro de IDs y se detectan los mismos patrones en el fotograma *t* que en el fotograma *t–1*, el algoritmo debe ser capaz de encontrar la correspondencia entre ambos fotogramas y mantener los IDs de las piezas. En la Figura 5.14 se muestra gráficamente este tipo de escenarios. En el instante de tiempo *t–1* aparecen dos patrones etiquetados con los números 1 y 2. En el instante *t*, se muestra cómo los patrones se han desplazado, pero sus IDs se mantienen.

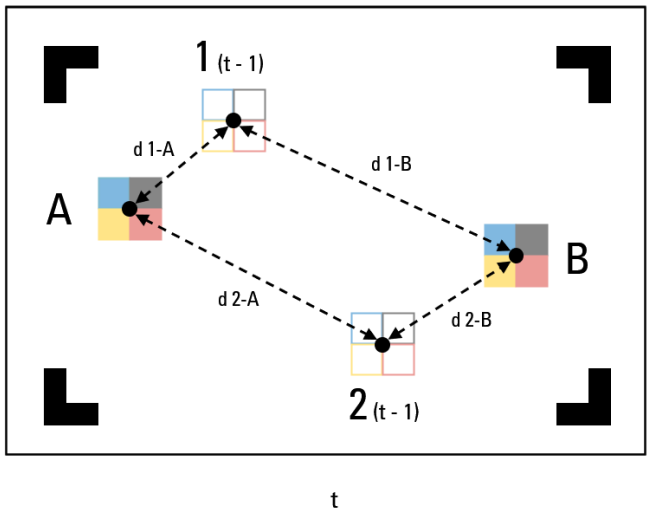


Figura 5.15: Caso B.2.1. El algoritmo de seguimiento calcula las distancias entre las antiguas detecciones y los nuevos candidatos. Para cada antigua detección, elige el nuevo candidato que presenta una distancia mínima. Es decir: para la detección *1* en *t–1*, el patrón que minimiza la distancia es el *A*. Mientras que para la detección *2* en *t–1*, el patrón que minimiza la distancia es el *B*.

El mecanismo que permite seguir y mantener los IDs de los patrones de un fotograma a otro se basa en la comparación de distancias. En la Figura 5.15 se muestra cómo actúa el algoritmo. Cuando el algoritmo analiza el fotograma tomado en el instante *t*, también cuenta con la información registrada en el instante *t–1*. En los registros, ademas de incluir los IDs, se incluyen las coordenadas de la pieza asociada a cada ID. De esta forma, al contar con las coordenadas de cada pieza tanto en el fotograma actual como en inmediatamente anterior, el algoritmo puede calcular las posibles distancias de desplazamiento de cada pieza. En la imagen, se muestran las posiciones de la pieza 1 y 2 en el instante *t–1* sin colorear. Por otro lado, se muestran con colores sólidos los dos patrones detectados en instante *t*.

Como se muestra en la Figura 5.15, para establecer la correspondencia entre las posiciones en *t–1* y *t*, el algoritmo calcula las distancias que separan cada patrón sin colorear con cada candidato —patrones coloreados—. Una vez calculadas las distancias, el algoritmo elige para cada antigua detección aquel candidato que presenta una menor distancia. De esta forma, en la Figura 5.15 se cumple que:

$$\begin{aligned} \textbf{Patrón 1}_{\,(t-1)}: \quad & \mathrm{d}_{\,(1-A)} < \mathrm{d}_{\,(1-B)} \quad \longrightarrow \quad \textbf{Patrón 1}_{\,(t-1)} \leftrightarrow \textbf{Patrón A} \\ \textbf{Patrón 2}_{\,(t-1)}: \quad & \mathrm{d}_{\,(2-B)} < \mathrm{d}_{\,(2-A)} \quad \longrightarrow \quad \textbf{Patrón 2}_{\,(t-1)} \leftrightarrow \textbf{Patrón B} \end{aligned}$$

- **Caso B.2.2: Sí se detectan patrones nuevos con respecto a los registros anteriores**. En este tipo de escenarios aparecen las mismas figuras

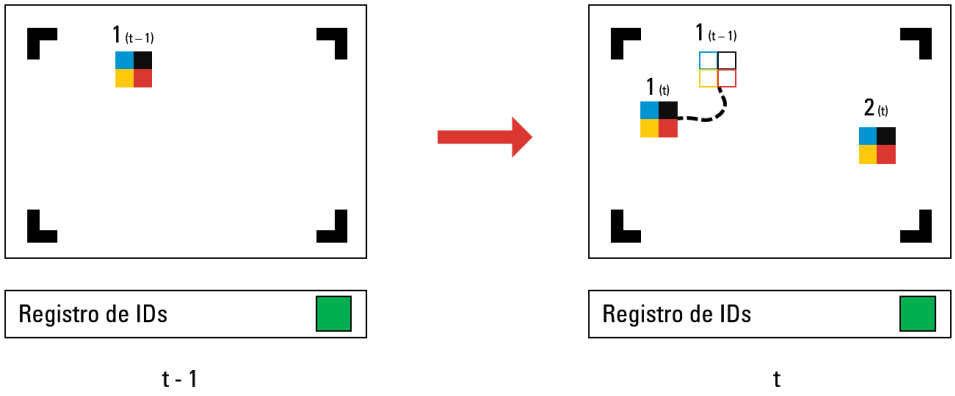


Figura 5.16: Caso B.2.2. En el instante *t* aparece un nuevo patrón que se une al registro previo que ya existía en *t–1*.

que en fotogramas anteriores, pero además, aparecen nuevos patrones. En la Figura 5.16, se muestra un ejemplo. En el fotograma *t–1* aparece un sólo patrón detectado. En el fotograma *t*, aparece el mismo patrón —en su nueva posición— y además, una nueva figura que irrumpe en el modelo.

El mecanismo que permite el seguimiento de viejos patrones y la incorporación de nuevas figuras —mientras se mantiene los IDs— se basa en lo explicado en la Figura 5.15. En la Figura 5.17 se muestra cómo desde el patrón registrado en *t–1* se computan las distancias a los nuevos candidatos —patrones *A* y *B*—. Una vez calculadas se determina que:

Patrón 1 $_{(t-1)}$: $\mathrm{d}_{(1-A)} < \mathrm{d}_{(1-B)} \longrightarrow$ **Patrón 1** $_{(t-1)} \leftrightarrow$ **Patrón A**
Patrón B : No asociado a ID previo $\longrightarrow$ **Patrón B** $\leftrightarrow$ **Nuevo registro**

En el algoritmo explicado en la lección [Ros18], se hace referencia a la detección de un sólo tipo de objeto: rostros. Sin embargo, en este proyecto se incluyen 19 tipos diferentes de códigos de colores. Extender la aplicación del algoritmo a los diferentes patrones requiere hacer una instancia del algoritmo descrito para cada tipo de patrón. Es decir, se necesita realizar un total de 19 instancias. En el Fragmento de código 13 se muestra cómo se realiza la instancia de la clase *CentroidTracker* para el patrón de aparcamiento y parque —para incluir los demás patrones, se debe repetir la operación hasta completar una instancia por cada código de colores—. Además de la instancia inicial, es necesario renovar la información fotograma a fotograma. Por ello, en el Fragmento de código 14 se incluye la función `.actualizar` —explicada en la lección [Ros18]—. En ella, el argumento *cuadrados_xxx* incluye las coordenadas x e y de los puntos que definen los rectángulos blancos de cada una de las imágenes binarias mostradas en la Figura 5.8. Los coordenadas que incluye la variable *cuadrados_xxx* se muestran en la Figura 5.18.

Para conocer los comandos necesarios de extracción de coordenadas de las esquinas de un rectángulo y trabajar con la información que éstas aportan, se

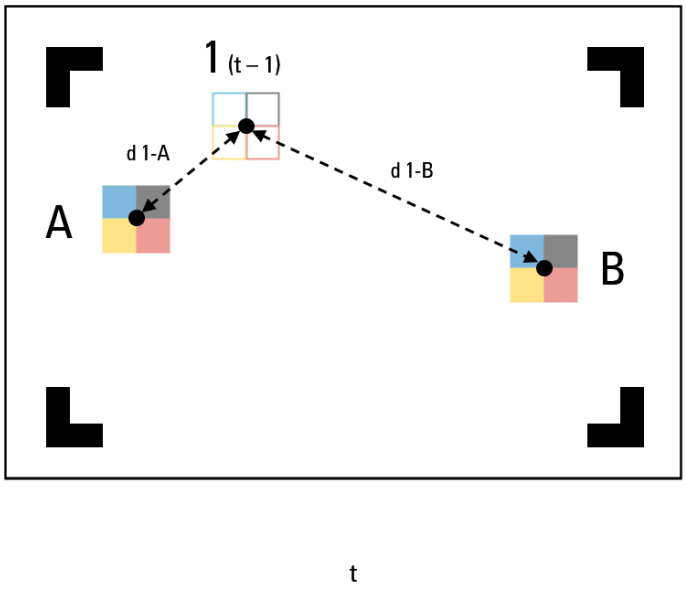


Figura 5.17: El algoritmo calcula las distancias entre el patrón detectado previamente y los nuevos candidatos A y B. A aquel patrón que no se asocia a ningún ID previo, se le da un ID nuevo y se incluye en el registro.

recomienda leer la lección [Ros16]. En el Fragmento de código 15 se incluyen dos comandos fundamentales utilizados en dicha lección. En primer lugar, la variable *bounding_box* recoge los datos necesarios para definir el rectángulo de área mínima que incluye por completo al contorno c. Como se muestra en la Figura 5.18, el contorno c corresponde a la detección de un patrón. Posteriormente, en la siguiente línea de código, la variable *bounding_box* actualiza su contenido al recoger las coordenadas de las cuatro esquinas que definen el rectángulo anterior.

```
ct_aparcamiento = CentroidTracker()
ct_parque = CentroidTracker()
```

Fragmento de código 13: Instanciación del algoritmo de seguimiento de patrones

```
piezas_aparcamiento = ct_aparcamiento.actualizar(
cuadrados_aparcamiento)
piezas_parque = ct_parque.actualizar(cuadrados_parque)
```

Fragmento de código 14: Comando para actualizar la información fotograma a fotograma

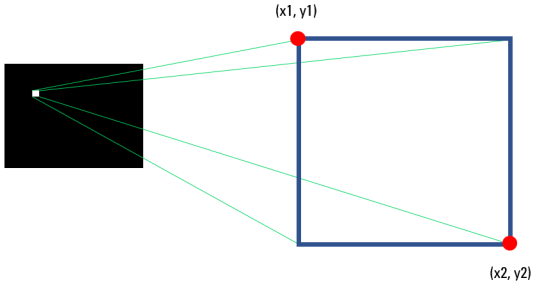


Figura 5.18: Cada detección —representado con un rectángulo blanco— queda definida por las coordenadas de la esquina superior izquierda y la esquina inferior derecha. Estas coordenadas, diferenciadas para cada tipo de patrón, son las que se incluyen como argumento en el Fragmento de código 14.

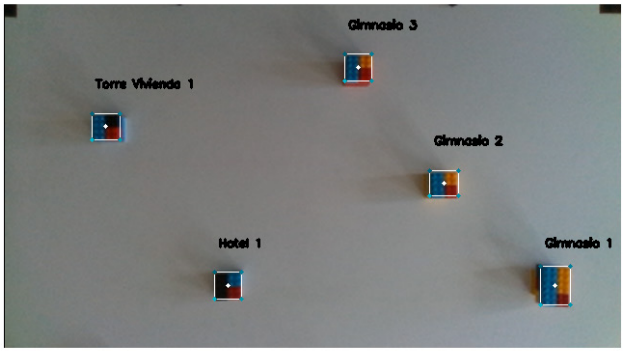
```
bounding_box = cv2.minAreaRect(c)
bounding_box = cv2.boxPoints(bounding_box)
```

Fragmento de código 15: Comandos para extraer las coordenadas de las cuatro esquinas de un rectángulo

5.5. Especificaciones para una correcta comunicación con Aimsun

Como se puede apreciar en la Figura 5.19, la información que se muestra por pantalla a los usuarios se condensa en un archivo '.xml'. Este archivo se lee posteriormente en Aimsun. Cabe recordar que la información que se incluye en el archivo '.xml' corresponde únicamente al último fotograma analizado, es decir, el fotograma en el que los usuarios dan la señal necesaria para concluir el proceso de diseño.

Como se puede observar en la Figura 5.19, en la parte izquierda de la imagen se incluye el fotograma del que se extrae la información del archivo '.xml' —situado a la derecha—. En dicho fotograma, aparece un patrón del tipo torre de viviendas, otro patrón del tipo hotel y tres patrones del tipo gimnasio. En el archivo '.xml' se refleja el área y las coordenadas x e y de cada uno de ellos. Cabe tener en cuenta que en el archivo '.xml' la numeración de los patrones comienza en 0, mientras que en el fotograma en 1. Por ello, los patrones del tipo gimnasio 1, 2 y 3 tienen su correspondencia en el archivo con los número 0, 1 y 2 respectivamente. En el archivo, el patrón de gimnasio con el número 0 tiene un área superior al resto de patrones. Esto se consigue al variar la configuración del patrón como se explica en el primer punto de las observaciones del algoritmo —siguiente sección—.




```
<?xml version="1.0"?>
- <MESA>
    <SECCIONES/>
  - <CENTROIDES>
    - <Torre_Vivienda>
        <nodo0 Area="783.0" Y="188" X="105"/>
      </Torre_Vivienda>
    - <Hotel>
        <nodo0 Area="784.0" Y="353" X="232"/>
      </Hotel>
      <Supermercado/>
      <Centro_Comercial/>
      <Industrial/>
      <Torre_Oficina/>
      <Colegio/>
      <Biblioteca/>
      <Museo/>
      <Restaurante/>
    - <Gimnasio>
        <nodo0 Area="1240.0" Y="353" X="571"/>
        <nodo1 Area="780.0" Y="247" X="456"/>
        <nodo2 Area="784.0" Y="127" X="367"/>
      </Gimnasio>
      <Bar/>
      <Teatro/>
      <Estadio/>
      <Hospital/>
      <Parada_Bus/>
      <Estacion/>
      <Aparcamiento/>
      <Parque/>
    </CENTROIDES>
</MESA>
```

Figura 5.19: El archivo 'xml.' generado incluye el tipo de patrón detectado, su área y sus coordenadas.

5.6. Observaciones del algoritmo para la detección de piezas

El algoritmo descrito en este capítulo de la memoria presenta algunas peculiaridades que se describen en los puntos siguientes.

- **Aumento de área de un patrón**. Como se muestra en la Figura 5.19, en el archivo '.xml' que se genera al finalizar este bloque, se incluyen los valores de área —en píxeles cuadrados— de cada una de las detecciones. Esto puede ser un parámetro importante en Aimsun, pues puede hacer que la generación o demanda de movilidad incremente o disminuya en función del tamaño del edificio.

  Pese a que los 19 patrones que se presentan en la Figura 5.2 se definen como configuraciones perfectamente cuadradas, el modelo de detección implementado permite el reconocimiento, de manera correcta, de patrones que, presentando la misma estructura de colores definida, tienen unas dimensiones superiores a las establecidas.

  La posibilidad de detectar estos patrones sin haberse incluido estas configuraciones de mayor tamaño en el modelo de entrenamiento puede estar asociado a un modelo poco restringido. Es decir, para que el modelo sólo detectase los patrones perfectamente cuadrados —para los que se crea inicialmente el modelo—, deberían incluirse más imágenes con las que afianzar el entrenamiento.

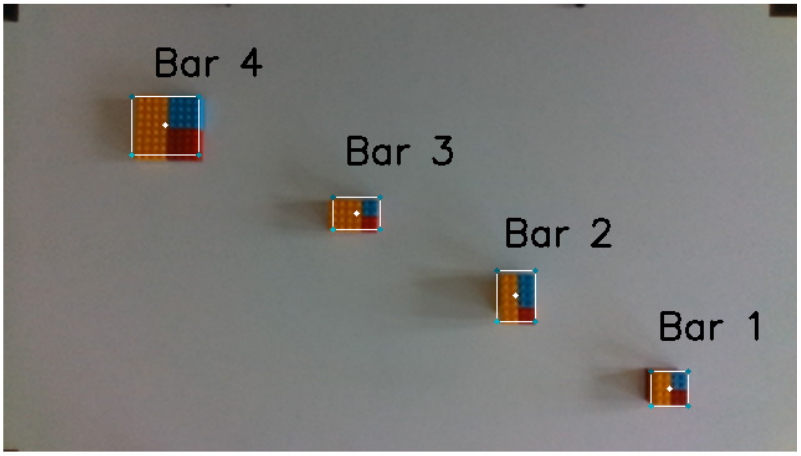


Figura 5.20: Además del tamaño estándar, —patrón etiquetado con el ID 1—, existen variaciones de mayor tamaño —detecciones con ID 2, 3 y 4—.

Este comportamiento del modelo se ha observado desde la fase de exploración asociada a las pruebas iniciales. Sin embargo, en lugar de resultar una desventaja, se ha decidido mantener esta característica y obtener provecho de esta circunstancia: el modelo es capaz de detectar correctamente patrones ligeramente distorsionados —pero que respetan el código de color definido en el entrenamiento—. De esta forma, se puede aportar flexibilidad al modelo sin necesidad de aumentar el número de fotos de entrenamiento. En la Figura 5.20 se muestra un ejemplo en el que aparecen cuatro patrones del tipo bar en escena. Cada uno de ellos tiene un tamaño diferente: la detección de ID 1 es el patrón definido como *perfectamente cuadrado*, es decir, el utilizado en el entrenamiento. Los patrones de ID 2 y 3 incluyen una distorsión vertical y horizontal, respectivamente; mientras que la detección de ID 4 respeta las proporciones, pero ocupa un área cuatro veces mayor que la del patrón de entrenamiento.

Es importante reseñar que este comportamiento no incluye ninguna desventaja notable al modelo, pues en ningún caso se han observado confusiones entre los 19 códigos de colores durante el período de pruebas.

Las distorsiones que el modelo acepta —aplicadas sobre un código completamente cuadrado— se muestran en la Figura 5.21. En primer lugar, arriba a la izquierda, se muestra el patrón de entrenamiento —cuadrado— junto a las posibles zonas de distorsión —bloques no coloreados—. Arriba a la derecha, se muestra la máxima distorsión vertical que el modelo es capaz de reconocer. Como se puede comprobar, se extienden los colores de las dos unidades superiores del patrón inicial —amarillo y azul—. Por otro lado, abajo a la derecha, se muestra un caso de distorsión horizontal, donde se prolonga el color amarillo de las piezas laterales. De esta forma, siempre que

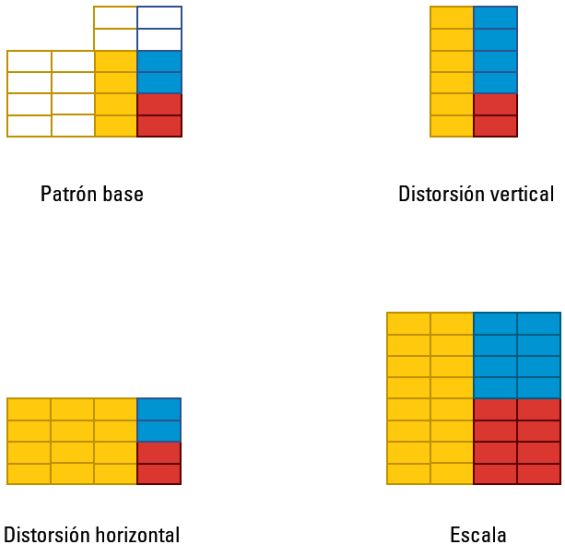


Figura 5.21: Se han examinado tres tipos de distorsión a partir de los patrones utilizados en el entrenamiento. Para mostrar las distorsiones se han utilizado patrones del tipo bar.

se realice una distorsión, se mantendrá intacta la estructura del patrón inicial y se extenderán los colores en la dirección adecuada. Finalmente, abajo a la derecha, se encuentra una pieza a la que se le ha aplicado una escala, de tal forma que el área que ocupa es cuatro veces el área del patrón inicial.

- **Número de fotogramas por segundo**. El número de fotogramas por segundo —FPS— que el algoritmo es capaz de procesar es un parámetro clave. Para el buen funcionamiento del seguimiento de patrones, es fundamental que se cuente con un muestreo lo suficientemente rápido, pues la mecánica del algoritmo recae directamente en el contraste de información de un fotograma con respecto al anterior. Si el muestreo es alto, las distancias recorridas por los patrones son más pequeñas. Si el muestreo es bajo —bajo número de FPS—, las distancias aumentan, pues se le ofrece más tiempo a los usuarios para mover las piezas sin tener ninguna entrada de información entre fotogramas procesados. Es en estos escenarios donde pueden aparecer errores en el algoritmo de seguimiento. En la Figura 5.22 se muestra cómo las piezas $1_{(t--1)}$ y $2_{(t--1)}$ siguen las trayectorias marcadas en verde, cada una por separado. A la hora de evaluar las distancias a los posibles candidatos, se observa que la distancia desde el patrón $1_{(t--1)}$ a la detección B es menor que a la detección A. Esto conduce a error, pues realmente el patrón $1_{(t--1)}$ está asociado a la detección A. Este error se evita si el número de FPS es alto, lo que permite un muestreo frecuente. De esta forma, los desplazamientos son mucho más pequeños y el algoritmo, más fiable y robusto.

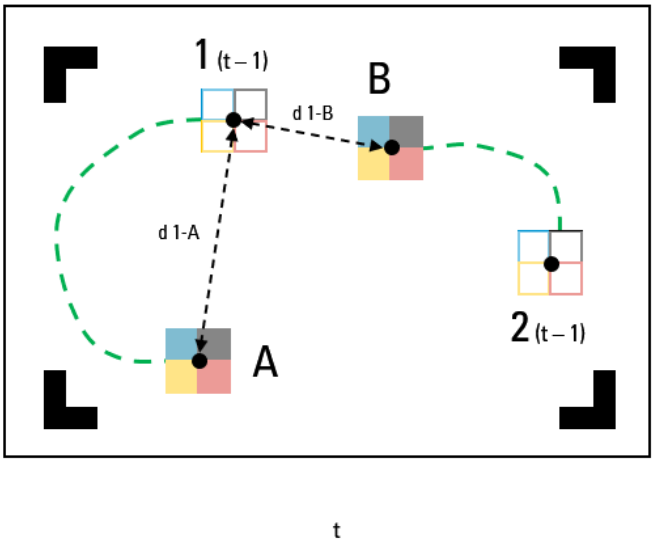


Figura 5.22: Debido a un bajo número de FPS, de las trayectorias marcadas en línea verde, sólo se extraen las posiciones mostradas. Como el algoritmo de seguimiento asocia patrones de uno y otro fotograma basándose en la mínima distancia, en este caso, aparece un erro de seguimiento.

Patrón 1 $_{(t-1)}$: d $_{(1-B)}$ < d $_{(1-A)}$ → Patrón 1 $_{(t-1)}$ ↔ Patrón B ⟶ **Error**

Capítulo 6

Simulación de movilidad: Aimsun

En este capítulo se describen las acciones implementadas en Aimsun para recrear y simular los escenarios dispuestos sobre la mesa. Estas acciones toman la información que proporcionan los archivos '.xml' descritos en los capítulos anteriores. En la Figura 6.1 se muestran los archivos que deben ser leídos desde Aimsun:

- **Archivo '.xml1'**. Contiene las coordenadas con las que se definen las secciones.
- **Archivo '.xml2'**. Incluye la lista de las intersecciones del sistema y un registro con las secciones de entrada y salida de cada una de ellas.
- **Archivo '.xml3'**. Contiene los datos extraídos de las detecciones de patrones. Posibilita la creación de puntos desde los que se generan y atraen desplazamientos.
- **Archivo '.xml4'**. Registra las trayectorias seguidas por los vehículos de la simulación.

La creación de los escenarios en Aimsun sigue un proceso metódico. Este proceso es aplicable a cualquier tipo de modelo ensayado, no solo para los creados a partir de la extracción de datos de un modelo físico, como en este proyecto. En plataformas como YouTube se pueden encontrar una notable cantidad de lecciones para comprender las nociones básicas del proceso de diseño en Aimsun [Con19].

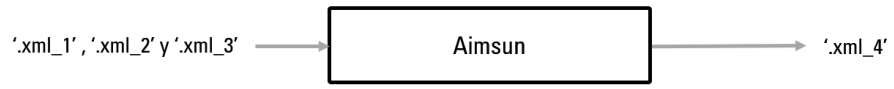


Figura 6.1: Para poder recrear los escenarios en Aimsun, se requiere la entrada de datos de las secciones, intersecciones y enclaves urbanos. Desde Aimsun también se genera un archivo con los resultados de la simulación.

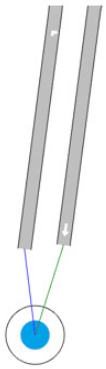


Figura 6.2: Los elementos gráficos que componen un modelo en Aimsun son las secciones y los centroides.

De estas lecciones se extrae que cualquier modelo que quiera ser simulado en Aimsun debe contener las siguientes características:

- **Secciones e intersecciones**. Conjunto de carreteras y cruces por los que los automóviles pueden desplazarse. En la Figura 6.2 se muestra una pareja de secciones.
- **Centroides**. Puntos desde los que se generan y demandan desplazamientos de automóviles. En la Figura 6.2 se puede apreciar cómo aparece un centroide asociado a la entrada y salida de una vía.
- **Escenario, experimento y replicación**. Permiten, sobre una misma geometría, aplicar diferentes condiciones. Por ejemplo, se pueden aplicar variaciones sobre los horarios de generación y demanda, incluir diferentes secuencias semafóricas, etc.

Pese a que estas acciones son relativamente sencillas de ejecutar mediante la interfaz habitual del ordenador —ratón—, se debe encontrar la forma de generar los mismos elementos a través de código. Aimsun incluye la posibilidad de generar guiones de código desde dentro del programa utilizando Python. Gracias a esta característica, se generan los guiones que deben ejecutarse para dar forma a los diferentes modelos.

Por otro lado, Aimsun cuenta con la documentación necesaria para desarrollar los algoritmos. Esta documentación puede encontrarse en la ventana de *Ayuda* $\longrightarrow$ *Documentación del Scripting*. Sin embargo, investigar y explorar esta documentación puede resultar un proceso arduo y consumir mucho tiempo. Por otro lado, además de necesitarse un proceso de estudio de la documentación, la depuración del código en Aimsun resulta mucho más compleja que en un IDE convencional como Visual Studio Code. En Aimsun, tan solo se puede mostrar al usuario información del contenido de las variables a través del uso de la función `print`. Además, este comando sólo muestra resultados cuando el contenido de la variable es numérico

—por ejemplo, una coordenada—. Cualquier otro tipo de dato más complejo —como una variable que guarda un objeto del tipo sección o centroide—, no se muestra por pantalla.

De lo mencionado en los párrafos anteriores se puede concluir que implementar en Aimsun objetos como secciones, centroides u otro elemento mediante código no es una tarea sencilla si no existen ejemplos que sirvan como referencia. Por ello, en las siguientes secciones de este capítulo, además de explicar las características cualitativas del algoritmo implementado, se proporciona con detalle la secuencia de código necesaria para la generación de los elementos nombrados.

Para crear un guión de código en Aimsun, tan sólo hay que seleccionar la venta *Proyecto* $\longrightarrow$ *Nuevo* $\longrightarrow$ *Guiones Python*. Una vez abierto un nuevo guión, se puede comenzar a crear escenarios mediante el uso de código.

6.1. Generación de una capa

En Aimsun, los elementos gráficos —secciones, intersecciones o centroides— se dibujan sobre capas. De esta forma se pueden superponer diferentes capas para mantener la información del modelo separada y bien organizada. En este proyecto, sin embargo, se utiliza una misma capa en la que se dibujan todos los elementos del modelo. Un archivo '.ang' cuenta con varias capas que pueden ser utilizadas para este propósito, pero también puede resultar de interés crear una específicamente para aquellos elementos del modelo que se originan a partir de datos extraídos de la mesa interactiva. En el Fragmento de código 16 se muestra cómo crear una nueva capa mediante código. En la última línea, la variable *layer* recoge la información de la capa creada, y puede ser utilizada cada vez que una función requiera una capa como argumento.

```
cmd = model.createNewCmd( model.getType( "GKLayer"))
model.getCommander().addCommand( cmd )
layer = cmd.createdObject()
```

Fragmento de código 16: Aimsun. Comandos para la generación de una capa en el modelo.

6.2. Generación de secciones e intersecciones

Como se ha mencionado en numerosas ocasiones en esta memoria, las secciones son las vías o calles por las que se desplazan los vehículos. Para generar una sección mediante código, es necesario incluir en el guión de Python —dentro de Aimsun— los comandos que se indican en el Fragmento de código 17.

En primer lugar, es necesario definir una variable en la que se registran una pareja de puntos. Para ello, se hace uso del objeto *GKPoints*. Posteriormente, es

necesario definir los puntos inicial y final de la sección por medio de sus coordendas *xyz*. Esto se realiza mediante el objeto *GKPoint*. Obsérvese la diferencia entre *GKPoints* —plural— y *GKPoint* —singular—. Una vez definidos los puntos iniciales y finales, éstos deben incluirse en la variable definida en la primer línea de código.

Posteriormente, en las últimas líneas de el Fragmento de código 17, se muestra cómo crear un objeto del tipo sección —*GKSection*—. A este objeto se le añaden sus características en la siguiente línea, donde se definen el número de carriles, el ancho del carril, los puntos inicial y final que restringen la recta, y por último, la capa donde se dibujan. Como se puede apreciar, en la penúltima línea se guarda el objeto creado en la variable *seccion*, lo que permite hacer uso de ella cada vez que sea necesario. Finalmente, en la última línea resulta necesario redefinir el ancho de carril.

```
points = GKPoints()
ptoInicial = GKPoint(x1, y1, 0)
ptoFinal = GKPoint(x2, y2, 0)
points.append(ptoInicial)
points.append(ptoFinal)

cmd = model.createNewCmd( model.getType( "GKSection"))
cmd.setPoints(carriles,ancho,points,layer)
model.getCommander().addCommand( cmd )
seccion = cmd.createdObject()
seccion.setLaneWidth(ancho)
```

Fragmento de código 17: Aimsun. Comandos para la generación de una sección.

El proceso que se describe en el Fragmento de código 17 debe repetirse para cada uno de los tramos que definen una calle o carretera completa, por lo que es muy útil crear una función que ejecute esos comandos cada vez que sea necesario. Sin embargo, los resultados que ofrece el código implementado hasta ahora ofrecen como resultados secciones que se conectan de una forma muy abrupta entre sí, tal y como se muestra en la parte superior de la Figura 6.3. Para intentar mejorar la conexión entre tramos, se hace uso de los giros, que permiten incluir tramos curvos que suavizan el trazado de las carreteras, tal y como se muestra en la parte inferior de la Figura 6.3. En el Fragmento de código 18 se muestra cómo crear un giro. En primer lugar, se crea un objeto del tipo *GKTurning*. Posteriormente, se introducen las dos secciones que se desean conectar. Es importante que la conexión de las secciones sea consecuente con el sentido de las mismas, es decir, no se pueden conectar dos secciones con sentidos opuestos. Por último, en la última línea se crea la variable *giro*, que guarda toda la información que define el giro recién creado.

Con respecto a las coordenadas que definen los puntos inicial y final de cada sección en el Fragmento de código 17, cabe mencionar dos observaciones importantes:

- **Corrección de coordenadas**. Como ocurre en el campo del procesamiento de imagen y se menciona en el Capítulo 4, la librería utilizada para extraer las

```
cmd = model.createNewCmd( model.getType( "GKTurning"))
cmd.setTurning( seccion2, seccion1 )
model.getCommander().addCommand( cmd )
giro = cmd.createdObject()
```

Fragmento de código 18: Aimsun. Comandos para la generación de un giro.

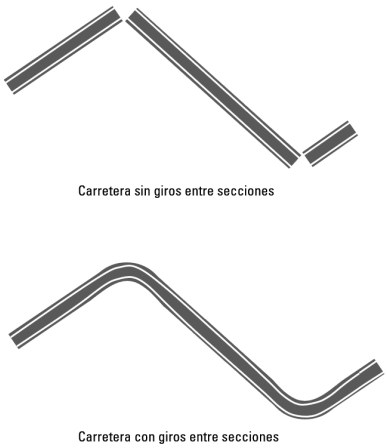


Figura 6.3: Cuando se incluyen giros para unir las secciones, los tramos de las carreteras quedan suavizados.

coordenadas de las líneas o secciones —OpenCV— trabaja con el eje vertical definido en sentido contrario al convencional. Sin embargo, Aimsun trabaja con una definición de ejes convencional. Los resultados de esta descoordinación se muestran en la Figura 6.4. Para solucionar esto, es necesario corregir los valores de la coordenada vertical de cada punto antes de trabajar con ellos en el algoritmo. Por otro lado, también cabe mencionar que se aplica una escala de amplificación a los datos recogidos. Es decir, tal y como se muestra en la Figura 6.5, las coordenadas de cada punto registrado se multiplican por 2. Así, los elementos del modelo no se aglomeran y visualmente respetan una proporción adecuada con respecto a distancias predefinidas, como la longitud de un vehículo.

- **Reducción de longitud de secciones**. Como se ha mencionado anteriormente, la incorporación de giros permite unir las secciones de una forma suave. Para que estos giros tengan un radio de curvatura mayor —y por lo tanto las uniones sean más suaves—, se reduce la longitud de las secciones que definen una calle. En la Figura 6.6 se visualizan los pasos seguidos. En primer lugar se las longitudes de las secciones se reducen. Como se puede observar, en un primer momento el punto final de una sección coincide con el punto inicial de la siguiente. Al reducir las longitudes —manteniendo la pendiente que presenta cada sección— los puntos dejan de coincidir en el plano. Finalmente, se crean las secciones de longitud reducida y se añaden los giros para unirlas.

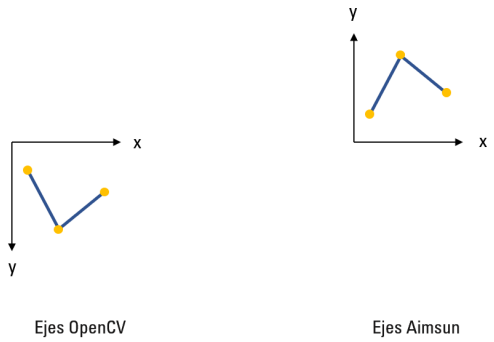


Figura 6.4: La definición opuesta del eje vertical conduce a realizar una corrección de la coordenada vertical de todos los puntos.

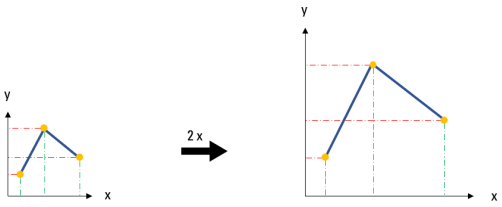


Figura 6.5: Los valores de las coordenadas de todos los puntos se multiplican por 2 para guardar una proporción adecuada con respecto a otras longitudes predefinidas como longitudes de vehículos.

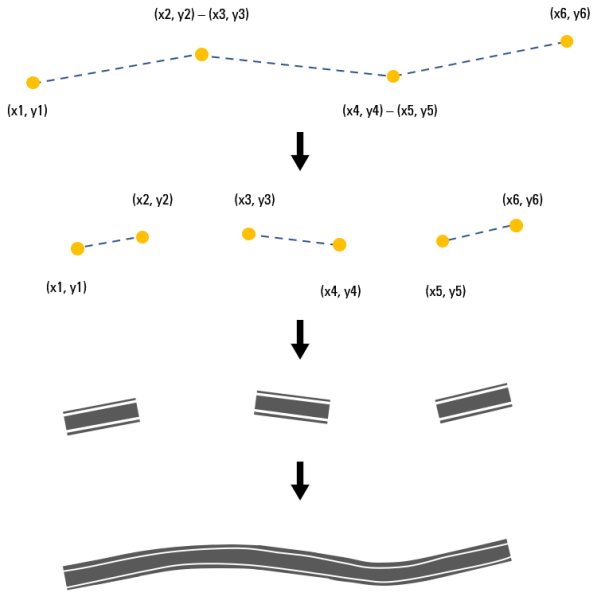


Figura 6.6: Al reducir la longitud de las secciones, los radios de curvatura de los giros aumentan y los trazos son más suaves.

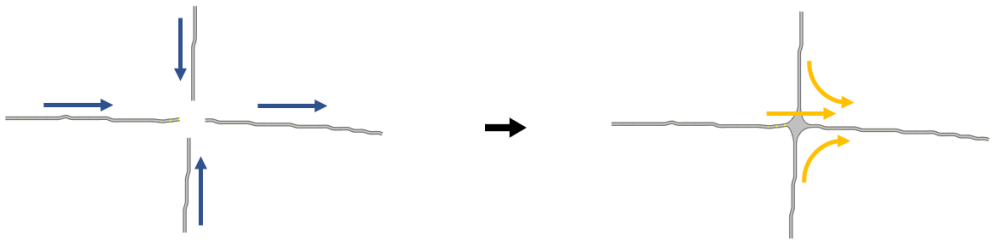


Figura 6.7: Las intersecciones en Aimsun se generan a partir de giros. Es fundamental contar con un registro de entrada y salida de cada intersección que indique qué vías deben unirse. En este caso, se generan tres giros —flechas amarillas—.

Una vez generadas todas las secciones que definen las carreteras o calles de manera independiente, es el momento de conectarlas a través de las intersecciones. En la Figura 6.7 se muestra, para el ejemplo explicado en el Capítulo 4, cómo se genera la intersección que une las cuatro calles del modelo, creando un entramado urbano. Para crear cualquier intersección, se deben implementar los mismos comandos que se muestran en el Fragmento de código 18, donde los argumentos de la segunda línea —*seccion1* y *seccion2*— corresponden a las secciones de entrada y salida a cada intersección, es decir, se debe crear un giro para cada unión de secciones, tal y como se muestra en la Figura 6.7. Para crear estos giros de manera eficiente, se hace uso del listado de vías de entrada y salida de cada intersección. Este listado se encuentra en el archivo '.xml' que recoge los datos de las intersecciones, tal y como se explica en el Capítulo 4.

6.3. Generación de centroides

Como se define al principio de este capítulo, un centroide es un punto desde el que se generan o demandan desplazamientos. Antes de poder crear un sistema de centroides mediante el uso de código, es necesario generar la estructura previa que requiere Aimsun. Esta estructura consta de:

- **Área de trabajo**. Aunque con la interfaz habitual —uso de ratón— este paso no es necesario —o no se realiza explícitamente—, resulta imprescindible definir una zona de trabajo. En el caso del proyecto, se define una rectángulo que respeta las medidas que ofrece el campo de visión de la cámara Intel. De esta forma, cada pixel de las imágenes de la cámara tiene su punto equivalente en Aimsun. En el Fragmento de código 19 se muestra cómo crear un polígono o área de trabajo. En primer lugar, como en el caso de las secciones, se define una variable —*puntosPoligono*— que guarda las coordenadas de todos los puntos. Posteriormente, se definen las coordenadas de los cuatro puntos que marcan las esquinas del rectángulo de trabajo. Nuevamente, se advierte de la diferencia entre los objetos *GKPoints* —plural— y *GKPoint* —singular—. En

las siguientes líneas del Fragmento de código 19, se muestra cómo crear un polígono o área de trabajo a partir de los puntos definidos y utilizando una capa de dibujo —Fragmento de código 16—. Por último, la variable *poligono* recoge la información del objeto creado.

```
puntosPoligono = GKPoints()
pto1 = GKPoint(0, 60, 0)
pto2 = GKPoint(0, 420, 0)
pto3 = GKPoint(640, 420, 0)
pto4 = GKPoint(640, 60, 0)
puntosPoligono.append(pto1)
puntosPoligono.append(pto2)
puntosPoligono.append(pto3)
puntosPoligono.append(pto4)

cmd = model.createNewCmd(
model.getType( "GKPolygon"))
cmd.setPoints( puntosPoligono, layer )
model.getCommander().addCommand( cmd )
poligono= cmd.createdObject()
```

Fragmento de código 19: Aimsun. Comandos para la generación de un área de trabajo.

- **Red del problema**. La red del problema es la estructura sobre la que se añaden diferentes elementos del modelo como la configuración de centroides o la demanda de tráfico. En el Fragmento de código 20 se muestra cómo añadir al modelo la red del problema. Como en ocasiones anteriores, en primer lugar se crea un objeto, en este caso, del tipo *GKProblemNet*. A continuación, en la segunda línea se incluye el área de trabajo creada en el punto anterior. Por último, la información del objeto creado se guarda en la variable *problemNet*

```
cmd = model.createNewCmd( model.getType(
"GKProblemNet"))
cmd.setData( poligono )
model.getCommander().addCommand( cmd )
problemNet = cmd.createdObject()
```

Fragmento de código 20: Aimsun. Comandos para la generación de la red del problema.

- **Configuración de centroides**. El siguiente paso requiere incluir una configuración de centroides al modelo. En el Fragmento de código 21 se muestra cómo realizar este paso. El proceso es similar a los descritos anteriormente.

Una vez creada la estructura de objetos necesaria, se puede dar paso a la generación de los centroides, cuyos datos se encuentran en el archivo '.xml' generado

```
cmd = model.createNewCmd(
model.getType( "GKCentroidConfiguration"))
cmd.setProblemNet( problemNet )
model.getCommander().addCommand( cmd )
centroidConfiguration = cmd.createdObject()
```

Fragmento de código 21: Aimsun. Comandos para la generación de una configuración de centroides.

por el bloque de detección de piezas. En el Fragmento de código 22 se muestran los comandos necesarios. En primer lugar se define la posición del centroide. Se recuerda que se requiere corregir los valores de las coordenadas verticales, tal y como se mostraba en la Figura 6.4. Posteriormente, para cada centroide se genera un objeto del tipo *GKCentroid* y en la siguiente línea se añaden sus datos: la posición y la configuración de centroides creada en los puntos anteriores. Como en otras ocasiones, la información del objeto generado se guarda en la variable *centroide*. Finalmente, en las últimas líneas del Fragmento de código 22, se muestra cómo crear una conexión entre centroide y sección. Esta conexión es fundamental, pues de otro modo los vehículos generados o demandados desde un centroide nunca entrarían en contacto con las secciones y no llegarían a desplazarse en la simulación. En primer lugar se crea un objeto del tipo *GKCenConnection*. En la siguiente línea se genera la conexión entre un centroide y sección particular. El orden con el que se introducen estos argumentos marca el sentido de la conexión: —entrada desde el centroide a la sección o salida, desde la sección al centroide—.

```
posicion = GKPoint(x, y, 0)

cmd = model.createNewCmd( model.getType( "GKCentroid"))
cmd.setData( posicion, centroidConfiguration )
model.getCommander().addCommand( cmd )
centroide = cmd.createdObject()

cmd = model.createNewCmd( model.getType( "GKCenConnection"))
cmd.setData( centroide, seccion )
model.getCommander().addCommand( cmd )
conexion = cmd.createdObject()
```

Fragmento de código 22: Aimsun. Comandos para la generación de centroides.

Una vez ejecutados los comandos de las secciones descritas hasta ahora, los resultados que se pueden alcanzar se muestran en la Figura 6.8.

6.4. Generación de matriz origen-destino

La interconexión de los diferentes centroides de un escenario da como resultado una matriz origen-destino —matriz OD—. En esta matriz, cada una de las

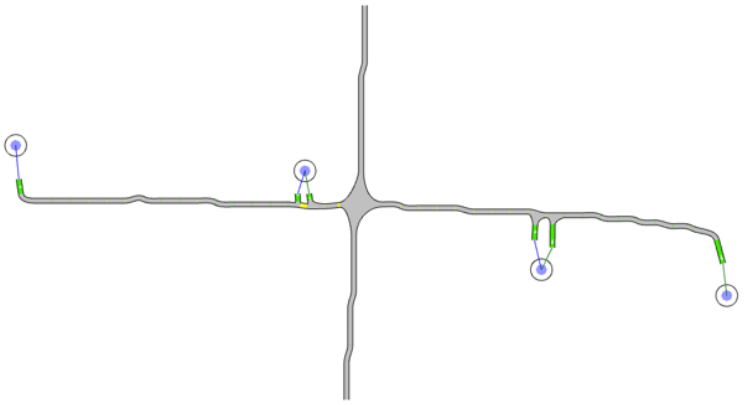


Figura 6.8: Con los comandos mostrados en los Fragmentos de código anterior se pueden formar entramados urbanos como el de esta figura.

celdas indica el número de desplazamientos que se establecen entre dos centroides diferentes.

Para establecer una conexión entre dos centroides de manera efectiva, es necesario que existan vías que comuniquen ambos centroides. Por ejemplo, en la Figura 6.8, teniendo en cuenta los sentidos de los carriles, es posible establecer una conexión desde el centroide situado más a la izquierda de la imagen hasta cualquiera de los centroides situados más a la derecha. Sin embargo, es imposible realizar la conexión inversa, pues no existen carriles con los sentidos adecuados para ello. En cualquier caso, no es necesario prestar mayor atención a esta característica, pues cuando se intenta establecer una conexión entre dos centroides que no disponen de vías para comunicarse, aparece un aviso por pantalla. Este aviso no detiene la ejecución del programa, sino que simplemente se limita a comunicar que la conexión no se ha implementado.

En el Fragmento de código 23 se muestra cómo se genera una matriz origen-destino. En primer lugar, se genera un objeto del tipo *GKODMatrix* para posteriormente, asociarle la configuración de centroides generada con los comandos del Fragmento de código 21. Posteriormente, la matriz recién configurada se guarda en la variable *MatrizOD*. A continuación, en las líneas finales del Fragmento de código 23, se incluye cómo generar un conexión entre dos centroides. El orden en el que se introducen los argumentos indica el sentido de la conexión de los centroides. Por otro lado, el argumento *n_viajes* indica el número de desplazamientos que deben generarse entre ambos centroides. Finalmente, en la última línea, se define el tipo de vehículo asociado a dichos desplazamientos.

En la última línea del Fragmento de código 23 se muestra la variable *vehiculo*. Esta variable contiene la información necesaria para asociar un coche a los trayectos de la matriz origen-destino —aunque es ampliable a otro tipo de vehículo, como autobús, camión o bicicleta, por ejemplo—. En el Fragmento de código 24 se

```
cmd = model.createNewCmd( model.getType( "GKODMatrix"))
cmd.setCentroidConfiguration( centroidConfiguration )
model.getCommander().addCommand( cmd )
MatrizOD = cmd.createdObject()

MatrizOD.setTrips(centroide_1,centroide_2,n_viajes)
MatrizOD.setVehicle(vehiculo)
```

Fragmento de código 23: Aimsun. Comandos para la generación de un matriz origen-destino.

muestra cómo seleccionar un tipo de vehículo predefinido en Aimsun. Para ello, la variable *vehicleType* guarda inicialmente todos los objetos del tipo *GKVehicle*. Posteriormente, se itera por todos los objetos de este tipo hasta encontrar el tercero de ellos —$i = 2$—, pues es la posición que ocupa el vehículo de tipo coche en el registro de Aimsun.

```
i = 0
vehicleType = model.getType( "GKVehicle")
for types in model.getCatalog().getUsedSubTypesFromType(
vehicleType ):
for vehicle in types.itervalues():
if i ==2:
vehiculo = vehicle
i = i + 1
```

Fragmento de código 24: Aimsun. Comandos para la selección de un vehículo predefinido.

Con los comandos mostrados en esta sección del capítulo, se pueden generar tantas matrices origen-destino como se deseen. De esta forma, se pueden superponer varias matrices para modelar escenarios más complejos. Por ejemplo, al superponer matrices origen-destino, se puede incluir en una misma simulación varios tipos de vehículos, o modelar escenarios más complejos que tiene asociados una generación y demanda de desplazamientos base, a la que se superponen matrices origen-destino para agregar picos de saturación al sistema.

6.5. Generación de demanda

Una vez generada la matriz origen-destino, a ésta se le debe asociar un horario de aplicación. Al asociarse un horario con una matriz OD, se dispone de los datos necesarios como para generar una demanda.

En primer lugar, en el Fragmento de código 25, se muestra cómo generar un horario. En la primera línea, se crea un objeto del tipo *GKScheduleDemandItem*. En las líneas siguientes, se le asocia la duración y el instante de inicio —ambos valores en segundos—. Posteriormente, el horario creado se asocia a la matriz de la sección anterior.

Una vez creado el horario, se genera la demanda siguiendo el mismo proceso que otros objetos mencionados en este capítulo. Posteriormente al objeto creado se le asocia la red del problema —*problemNet*— creada en el Fragmento de de código 20. Por último, se agrega a la demanda el horario recién creado en las líneas previas.

```
horario = GKScheduleDemandItem()
horario.setDuration(duracion)
horario.setFrom(inicio)
horario.setTrafficDemandItem(MatrizOD)

cmd = model.createNewCmd( model.getType( "GKTrafficDemand"))
cmd.setProblemNet( problemNet )
model.getCommander().addCommand( cmd )
demanda = cmd.createdObject()
demanda.addToSchedule(horario)
```

Fragmento de código 25: Aimsun. Comandos para la generación de un horario.

6.6. Generación de escenario

Antes de poder ejecutar la simulación, es necesario crear una estructura compuesta por: escenario, experimento y réplica. En el Fragmento de código 26 se muestra cómo crear estos objetos siguiendo la misma mecánica explicada en los fragmentos anteriores.

En primer lugar, al objeto del tipo *GKScenario* se le debe asociar la red del problema —*problemNet*— y la demanda generada en la sección anterior. Posteriormente, al objeto del tipo *GKExperiment* se le asocia el escenario creado en las líneas previas y se establece el *EngineMode*. Por último, en el Fragmento de código 26 se muestra cómo crear una replicación a partir del experimento generado en las líneas superiores.

6.7. Ejecución de simulación

Finalmente, para ejecutar la simulación con el escenario y datos cargados, tan solo hay que ejecutar el comando que se muestra en el Fragmento de código 27. Para la generación de los archivos '.xml' con los resultados generados en la simulación, resulta interesante generar en primer lugar un escenario —a partir de

```
cmd = model.createNewCmd( model.getType( "GKScenario"))
cmd.setProblemNet( problemNet )
model.getCommander().addCommand( cmd )
escenario = cmd.createdObject()
escenario.setDemand(demanda)

cmd = model.createNewCmd( model.getType( "GKExperiment"))
cmd.setScenario( escenario )
cmd.setEngineMode( 1 )
model.getCommander().addCommand( cmd )
experimento = cmd.createdObject()

cmd = model.createNewCmd( model.getType( "GKReplication"))
cmd.setExperiment( experimento )
model.getCommander().addCommand( cmd )
replica = cmd.createdObject()
```

Fragmento de código 26: Aimsun. Comandos para la generación de un escenario, un experimento y una replicación.

código o con la interfaz habitual— y seleccionar esta vez sí con la interfaz habitual la pestaña *Escenario dinámico* → *Resultados a generar* → *Guardar Animación XML*. Además, dentro de la pestaña *Resultados a generar* también es necesario seleccionar *Estadísticas* → *Emisión de contaminantes (Panis et al, 2006)*.

```
GKSystem.getSystem().executeAction( "execute", replica, [], "")
```

Fragmento de código 27: Aimsun. Comandos para la ejecución de una simulación.

Capítulo 7

Visualización de resultados: Minecraft

Este capítulo está dedicado a la recreación de un escenario virtual en el que se representan los datos recogidos de la mesa interactiva y los resultados de la simulación ejecutada en Aimsun. El programa utilizado para representar estos escenarios ha sido el videojuego Minecraft, aunque otras aplicaciones como Blender [Ble] o Unity [Uni] se han tenido en cuenta para desarrollar estas tareas. Finalmente, la posibilidad de trabajar a través de la API de Minecraft utilizando Python ha sido el factor que ha decantado la elección a favor de este conocido videojuego.

La instalación de los paquetes necesarios para utilizar la API se pueden encontrar en YouTube [OHa17]. Por otro lado, la totalidad de los comandos utilizados en los fragmentos de código de este capítulo tienen como origen el libro de referencia escrito por los autores de la API [Wha18].

Antes de dar paso a las explicaciones sobre la generación de los elementos que componen el escenario, resulta imprescindible crear una instancia para comunicar el código de Python con Minecraft. En el Fragmento de código 28 se muestra cómo realizarla. Además, en ese mismo fragmento se muestra cómo importar la librería que contiene el muestrario de bloques que se utilizan en este capítulo. Cabe señalar que siempre que se quiera ejecutar cualquier comando asociado a la instancia que se muestra en el Fragmento de código 28, un *mundo* de Minecraft debe estar abierto. De lo contrario, aparecerá un error en el terminal del IDE que se esté utilizando.

```
import mcpi.minecraft as minecraft
import mcpi.block as block

mc = minecraft.Minecraft.create()
```

Fragmento de código 28: Minecraft. Comandos para la generación de una instancia que permite comunicar el código de Python con Minecraft.

Los elementos que se incorporan al escenario y cuya generación se explica en este capítulo son:

- **Área de trabajo**. De igual forma que se define un área de trabajo en Aimsun, también se define en Minecraft. Este área de trabajo permite mostrar al usuario el campo de visión de la cámara y le ayuda a localizar, en la representación virtual, cada uno de los elementos dispuestos sobre la mesa.
- **Carreteras**. Se recrean las calles o carreteras dibujadas sobre la mesa. Sobre estas carreteras se representan los desplazamientos de los vehículos.
- **Edificios**. Se recrean los edificios representados mediante las piezas de Lego.
- **Decorados**. Se incluyen árboles, flores y estanques que se reparten de manera aleatoria por el escenario.
- **Desplazamientos**. Una vez recreado el escenario, se deben mostrar los desplazamientos que resultan de la simulación en Aimsun.

7.1. Generación del área de trabajo

Como se ha mencionado, el área de trabajo delimita el escenario generado en Aimsun. Para crear un escenario, se recomienda utilizar un *mundo extraplano*, puesto que de esta forma, se genera una superficie de bloques plana con la que es más sencillo trabajar. La creación inicial de este *mundo* puede realizarse mediante la interfaz habitual, por lo que no se requieren comandos para realizar dicha acción. Una vez creado —y abierto, aunque sea en segundo plano—, se pueden ejecutar los comandos que se muestran en el Fragmento de código 29. Los resultados de esta ejecución se muestran en la Figura 7.2. Con la primera línea del fragmento, se genera una superficie periférica de color gris oscuro, mientras que con la segunda, se genera el área de trabajo. En ambas líneas se hace uso del comando *setBlocks*, cuyos 6 primeros argumentos definen las coordenadas del volumen que se desea generar —los 3 primeros argumentos son la longitud, altura y profundidad del primer punto, mientras que los 3 siguientes definen la longitud, altura y profundidad del segundo punto—. Como se puede observar, se utilizan las variables *alto* y *largo*. Estas variables contienen las dimensiones, en píxeles, del campo de visión de la cámara —480 y 640 píxeles respectivamente, que se dividen entre 2 para tener unas dimensiones proporcionadas en Minecraft—. El último argumento de este comando define el tipo de bloque utilizado para rellenar el volumen generado.

En la Figura 7.1 se muestra cómo queda definido un volumen a partir de las coordenadas xyz de dos puntos. La terna de ejes en Minecraft queda definida tal y como se muestra en la imagen.

7.2. Generación de carreteras

En Minecraft, la generación de carreteras se realiza a través del comando *drawFace*, que en lugar de generar un volumen, genera un polígono definido por las coordenadas de las equinas que lo componen. En este caso, la estrategia seguida con este comando consiste en la formación de cuadriláteros a partir de dos puntos consecutivos. Estos puntos son los extraídos en el Capítulo 4, y deben

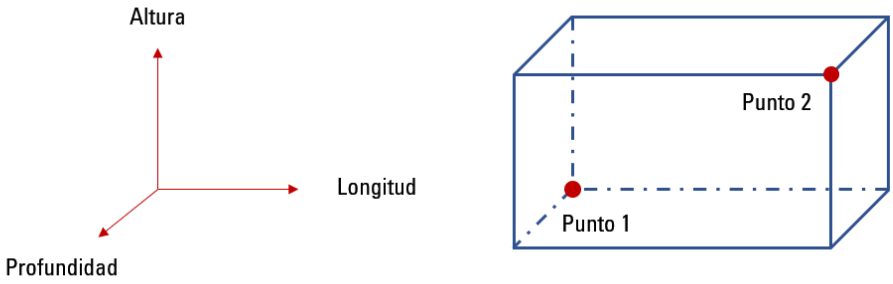


Figura 7.1: El comando *setBlocks* utiliza las tres coordenadas de los dos puntos definidos en la figura como argumentos.

```
mc.setBlocks(-100, -1, -100, largo/2+100, -1, alto/2+100,
block.WOOL.id,7)

mc.setBlocks(0, -1, 60/2, largo/2, -1, alto/2 -60/2,
block.GRASS.id)
```

Fragmento de código 29: Minecraft. Comandos para la generación de un área de trabajo.

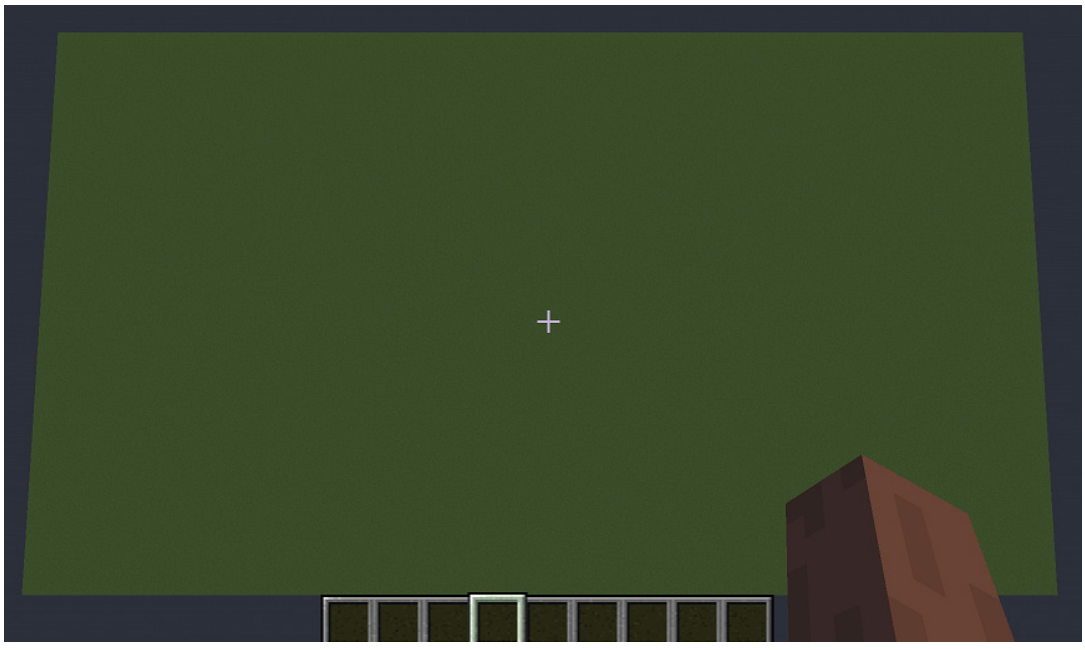


Figura 7.2: Resultados en Minecraft de la aplicación del Fragmento de código 29. Vista en planta. En color verde se puede observar el área de trabajo, es decir, la zona donde pueden aparecer elementos en el escenario virtual. La zona verde guarda las proporciones con el campo de visión de la cámara, por lo que todos los elementos detectados por la cámara se representan en la zona de color verde. Rodeando el área de trabajo aparece una zona de color gris oscuro sobra la que no se actúa.

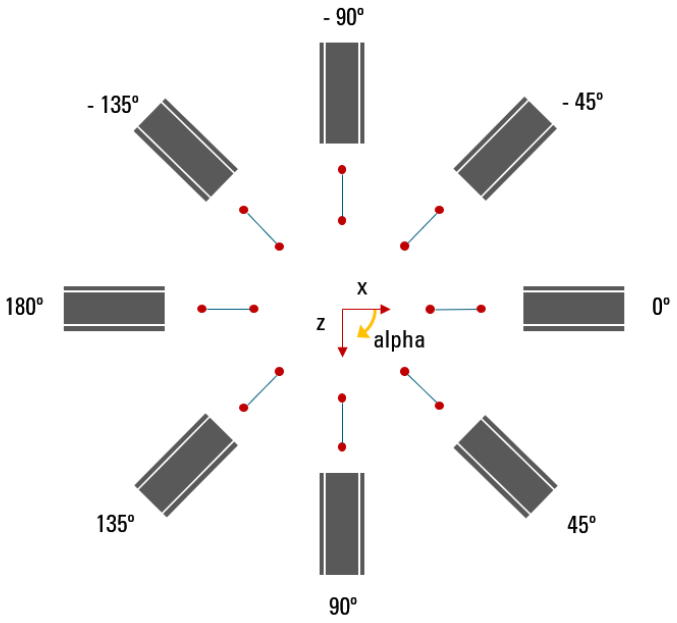


Figura 7.3: A partir de dos puntos consecutivos —color rojo—, se genera un cuadrilátero orientado en función del ángulo que formen los puntos de color rojo, que constituyen el eje de la sección.

ser consecutivos para evitar la aparición de discontinuidades en los dibujos de las carreteras —a diferencia de las necesidades de Aimsun, cuyo muestreo no necesita de la extracción de puntos consecutivos al contar con la integración de los giros, característica que aporta mucha flexibilidad a sus gráficos—. En la Figura 7.3 se muestra cómo se generan los cuatro puntos que definen el cuadrilátero en función del ángulo que forma la unión de los dos puntos consecutivos. Se implementa esta estrategia para conseguir un ancho de vía uniforme.

En el Fragmento de código 30 se muestra cómo utilizar el comando mencionado. En primer lugar, se requiere incluir las dos primeras líneas para poder utilizar el comando *drawFace*. Posteriormente, se definen los cuatro puntos que delimitan el cuadrilátero. Obsérvese que las coordenadas aparecen marcadas con asteriscos, lo que indica que están sujetas a las modificaciones necesarias por la orientación de los puntos —como se muestra en la Figura 7.3—. Por último, una vez definidos los puntos, se hace uso del comando. El primer argumento hace referencia a los puntos recién definidos, el segundo, hace que el cuadrilátero sea un polígono relleno, y no solo un contorno, mientras que el último, indica el tipo de bloque que se debe utilizar para dibujar el cuadrilátero.

Una vez generado un tramo de asfalto de la carretera, se añaden las líneas blancas que delimitan sus bordes mediante el comando que aparece en el Fragmento de código 31. Como en ocasiones anteriores, los tres primeros argumentos hacen referencia a las coordenadas del primer punto, mientras que los tres siguientes a las coordenadas del extremo final de la línea. El último argumento indica el tipo de bloque que debe ser utilizado para implementar la línea. El comando mostrado sólo dibuja la línea de un lateral, por lo que es necesario implementar la del lateral opuesto también. De

```
mc = minecraft.Minecraft.create()
mcdrawing = MinecraftDrawing(mc)
mc.create

puntos = Points()
puntos.add(x1*, -1, z1*)
puntos.add(x1*, -1, z1*)
puntos.add(x2*, -1, z2*)
puntos.add(x2*, -1, z2*)

mcdrawing.drawFace(puntos, True, block.WOOL.id, 7)
```

Fragmento de código 30: Minecraft. Comandos para la generación de un cuadrilátero o sección de carretera.

nuevo, las coordenadas aparecen marcadas con asteriscos, puesto que el en función del ángulo que describen los puntos teóricos iniciales, varían las operaciones que se deben implementar para que la línea aparezca en el lateral de la sección, y no en el eje de la misma.

```
mcdrawing.drawLine( x1*, -1, y1*, x2*, -1, y2*, block.WOOL.id, 0)
```

Fragmento de código 31: Minecraft. Comandos para la generación de una línea de bloques.

Por otro lado, para generar las intersecciones, se hace uso de nuevo del comando *drawFace*, pero en lugar de generar un cuadrilátero, se genera un octógono —que puede aproximarse a una circunferencia—. Para generar las líneas blancas que delimitan la intersección, se ejecuta de nuevo el mismo comando, con el que se dibuja con bloques blancos un octógono sin relleno, de dimensiones ligeramente inferiores.

Con los comandos mostrados en esta sección se pueden conseguir escenarios como los mostrados en la Figura 7.4, donde se puede ver en Minecraft el entramado de calles analizado en el Capítulo 4.

7.3. Generación de edificios

Los edificios que se generan en Aimsun para esta versión del proyecto son muy sencillos. Aunque sería deseable para mejorar la estética del trabajo, no se cuenta con una representación particular asociada a cada tipo de edificio, sino que todos cuentan con la misma estructura.

A partir de las coordenadas guardadas en el archivo '.xml' generado en el Capítulo 5 se genera un edificio compuesto por varios cubículos apilados. Estos

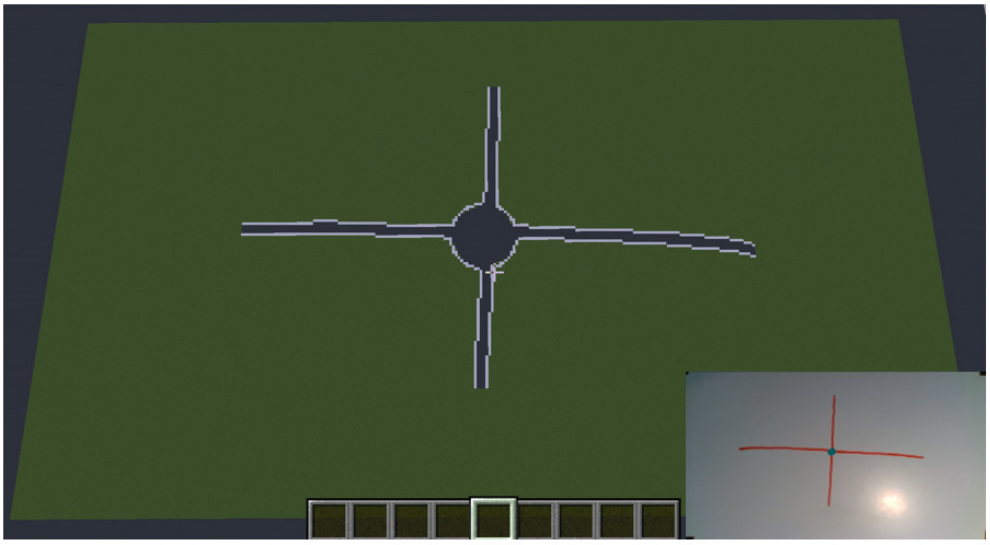


Figura 7.4: Conjunto de secciones e intersección representado en Minecraft a partir de los datos extraídos de la mesa. En la esquina inferior derecha se muestra la imagen de la que se extrae la información.

cubículos están generados con el comando *drawFace* explicado en el apartado anterior, de tal forma que se generan cuatro paredes compuestas por un marco y un ventanal. Adicionalmente, se genera otra superficie que sirve de techo para cada cubículo. Repitiendo esta fórmula tantas veces como se desee, se pueden conseguir edificios de diferente número de pisos, altura y área. En la Figura 7.5 se muestra un ejemplo de un grupo de edificios generados a partir del uso de la técnica descrita.

7.4. Generación de decorado

Los elementos que se incluyen en el modelo como decorado son tres: árboles, estanques y flores.

- **Árboles**. La técnica utilizada para recrear árboles se basa en la generación de esferas de bloques del tipo *hoja de árbol*. Mediante la generación de un tronco de altura variable con un número también variable de copas de árboles se crean diferentes morfologías de árboles. Para aportar mayor variedad, el diámetro de las copas también es variable. Los árboles se diseminan de manera aleatoria por el área de trabajo. En el Fragmento de código 32 se muestra cómo generar la copa de un árbol. Los tres primeros argumentos del comando *drawSphere* definen las coordenadas del centro de la esfera, mientras que el cuarto argumento, *r*, introduce el radio de la misma. Finalmente, el último argumento indica el tipo de bloque con el que se quiere generar el voulumend de la esfera. Para generar el tronco, se puede utilizar el comando expuesto en secciones anteriores —*drawLine*—.

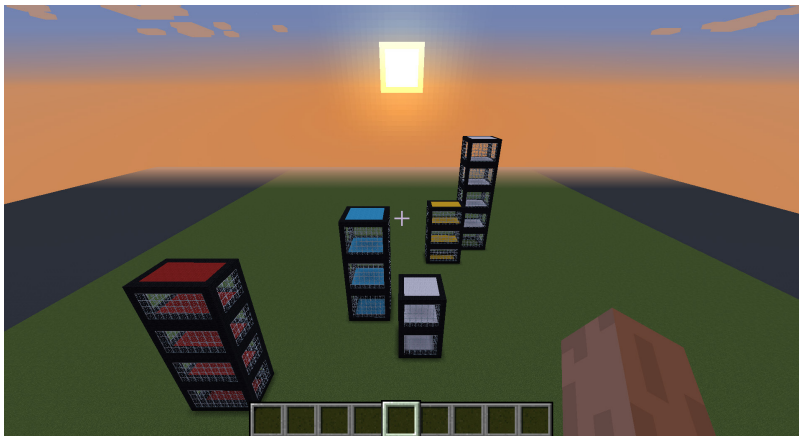


Figura 7.5: Conjunto de edificios generados en Minecraft a partir de la creación de polígonos. Gracias a ellos, para cada pared se forma un marco —cuadrilátero sin relleno— y un ventanal —cuadrilátero relleno con bloques de vidrio—. Los techos de cada piso se generan con cuadriláteros formados por bloques de colores.

```
mcdrawing.drawSphere(x, y, z, r,
block.LEAVES_SPRUCE_PERMANENT.id)
```

Fragmento de código 32: Minecraft. Comandos para la generación de una copa de árbol —esfera—.

En la Figura 7.6 se muestran los resultados de la estrategia implementada para generar los árboles del escenario.

- **Flores y estanques**. La generación de flores y estanques sigue la misma estrategia. A partir del comando *drawFace* se genera un polígono cuyos extremos no están ordenados —ver Figura 7.7— utilizando bloques de tipo agua o diferentes tipos de flores. Los grupos de flores y los estanques se generan también de forma aleatoria por el área de trabajo. En la Figura 7.8 se muestra un ejemplo, al nivel del terreno, de un escenario que integra todos los elementos descritos en esta sección.



Figura 7.6: Conjunto de árboles generados en Minecraft a partir de la combinación aleatoria de esferas y segmentos de diámetro y altura variable.

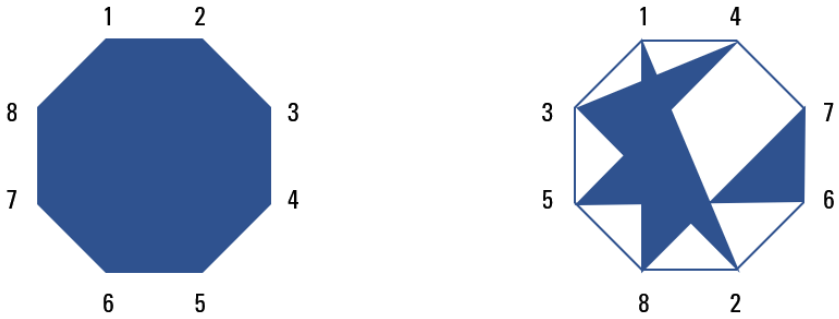


Figura 7.7: En la parte izquierda de la imagen se muestra un octógono cuyos vértices están ordenados secuencialmente. A la derecha se muestra una figura que parte inicialmente de la misma figura geométrica, pero que da un resultado completamente aleatorio y más orgánico al numerar los vértices de forma desorganizada.



Figura 7.8: Escenario con todos los elementos descritos en el capítulo: área de trabajo, carreteras, intersección, edificios, árboles, flores y estanques.

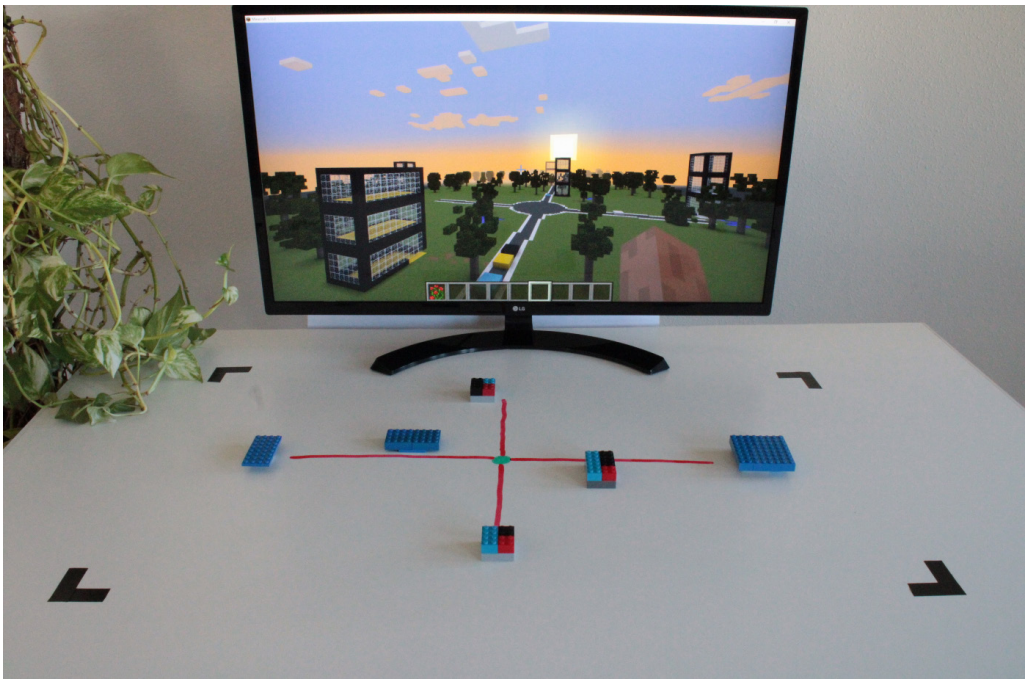


Figura 7.9: Escenario virtual con varios vehículos —volúmenes de colores— desplazándose por las carreteras.

7.5. Recreación de los desplazamientos de la simulación

La recreación de los desplazamientos en Minecraft requiere de la lectura del archivo '.xml' generado en Aimsun. En este archivo, aparecen registrados los tiempos de muestreo y las incidencias que ocurren en cada uno de ellos —aparición de vehículo, cambio de posición o llegada de vehículo a destino—. Para cada vehículo, que tiene asociado un ID particular, se selecciona un color que mantiene durante toda la representación de la simulación, facilitando su detección.

Para ejecutar la animación, en cada muestra del registro, se generan con el comando *setBlocks* los volúmenes que representan los vehículos. Posteriormente, se incluye un pequeño retardo de pocas fracciones de segundo y finalmente se borra el volumen generado. En la siguiente muestra, con las coordenadas actualizadas, el proceso se repite, y de esta manera se consigue crear un efecto visual de movimiento.

En la ejecución del proyecto se estudia la posibilidad de introducir diferentes modelos de vehículo en función de las características de la simulación. Esto requiere trabajar con las denominadas *entidades* de Minecraft, que permiten una visualización mucho más suave y agradable en el escenario virtual. Pese a que la API Python-Minecraft permite la manipulación de entidades de serie, no ofrece las mismas posibilidades con entidades personalizadas —coches, autobuses, tranvía, etc.— por lo que finalmente, se descarta esta opción. En la Figura 7.9 se muestra un ejemplo en el que aparecen volúmenes de colores —vehículos— desplazándose de acuerdo a los resultados de la simulación por el escenario recreado en Minecraft.

Capítulo 8

Resultados

En este capítulo se muestran de forma visual los resultados alcanzados por el proyecto mediante un caso de estudio sencillo que puede servir como manual de usuario.

En primer lugar, en la Figura 8.1 se pueden observar los elementos básicos que componen el proyecto: una mesa de color blanco en la que se delimita el espacio de trabajo, una pantalla en la que aparece información sobre la evolución del diseño, una cámara con la que introducir los datos del modelo físico al virtual y los dos elementos con los que trabajan los usuarios: piezas de lego y rotuladores de pizarra.



Figura 8.1: Elementos de trabajo del proyecto.

Una vez se ejecuta el código del proyecto, siempre con un mundo de Minecraft abierto, se puede dar paso a la generación del modelo, que se completa con los siguientes procesos:

- **Generación de entramado urbano**
  - **Calles**
  - **Intersecciones**
- **Generación de edificios y enclaves urbanos**

La definición del modelo comienza por la disposición del entramado urbano. La herramienta fundamental para el usuario en este proceso son los rotuladores. Con color rojo se definen las calles, mientras que con color verde se incluyen las intersecciones. En la Figura 8.2 se muestra cómo lo dibujado en la mesa tiene su reflejo tanto en el *software* de movilidad —Aimsun— como en el de visualización de datos —Minecraft—.

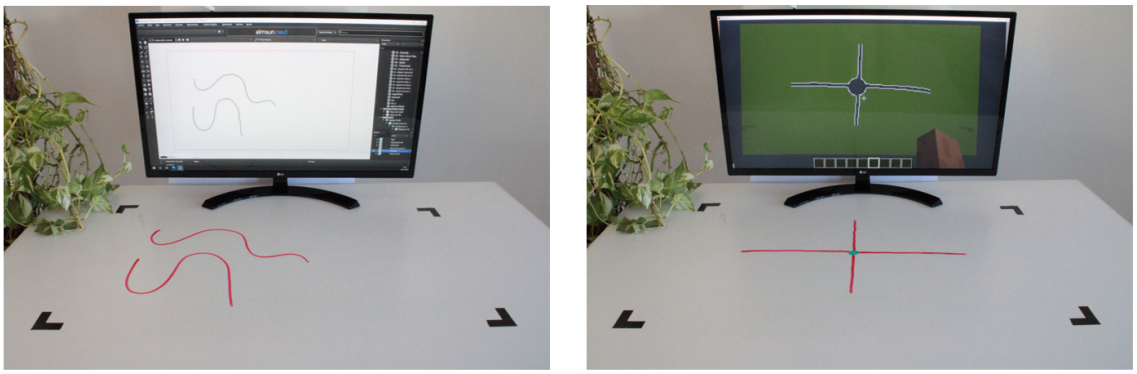


Figura 8.2: El entramado urbano diseñado sobre la mesa queda reflejado tanto en Aimsun —izquierda— como en Minecraft —derecha—. En las imágenes se muestran escenarios diferentes entre sí.

El proceso de definición del entramado urbano es secuencial. En una primea fase se realizan los dibujos de todo el entramado, y posteriormente, en una segunda fase se indican los sentidos de cada una de las calles. Para realizar el cambio de fase, los usuarios disponen de un menú muy sencillo. Moviendo la pieza de color verde que se muestra en la Figura 8.3 se indica en primer lugar el cambio de fase —casilla de *fase 1*—, y por último la finalización del diseño del entramado urbano —casilla de *fase 2*—.

Cuando el entramado urbano ha sido completamente definido, se puede dar paso a la disposición de los enclaves que componen el modelo. En la Figura 8.4 se muestran diferentes patrones dispuestos sobre la mesa. El proceso de seguimiento permite localizar y representar cada patrón también en Minecraft.

Juntando los elementos y procesos descritos, los usuarios pueden llegar a la formación de un modelo completo. En la Figura 8.5 se muestran imágenes en las que se puede apreciar el modelo físico formado sobre la mesa y diferentes perspectivas de la representación virtual que se genera en Minecraft.

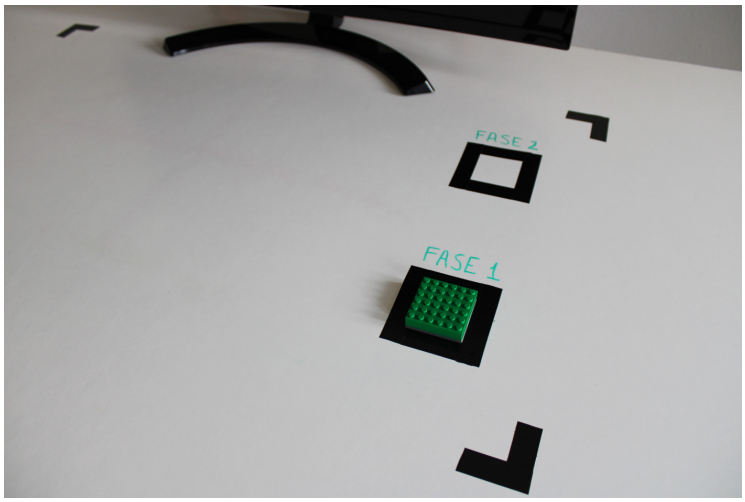


Figura 8.3: Los usuarios pueden indicar la finalización de las fases de diseño por medio del menú físico-virtual implementado.

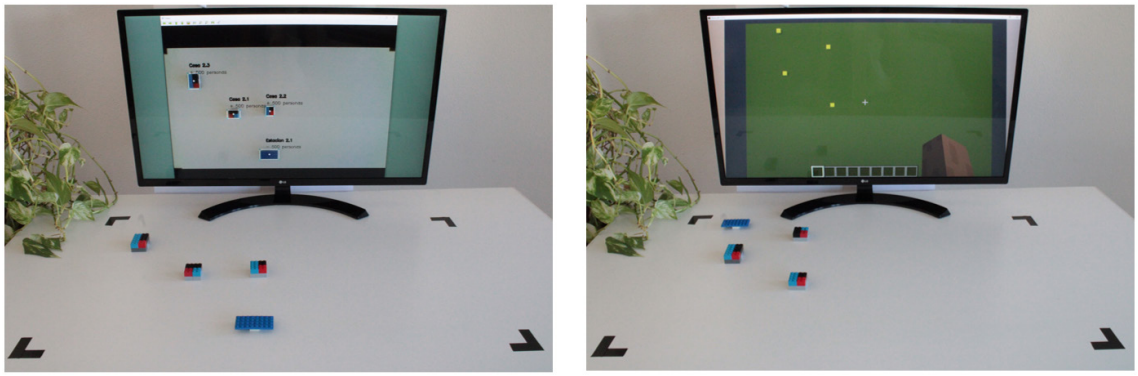


Figura 8.4: Los enclaves urbanos se definen a partir de los patrones de colores formados con piezas de Lego.

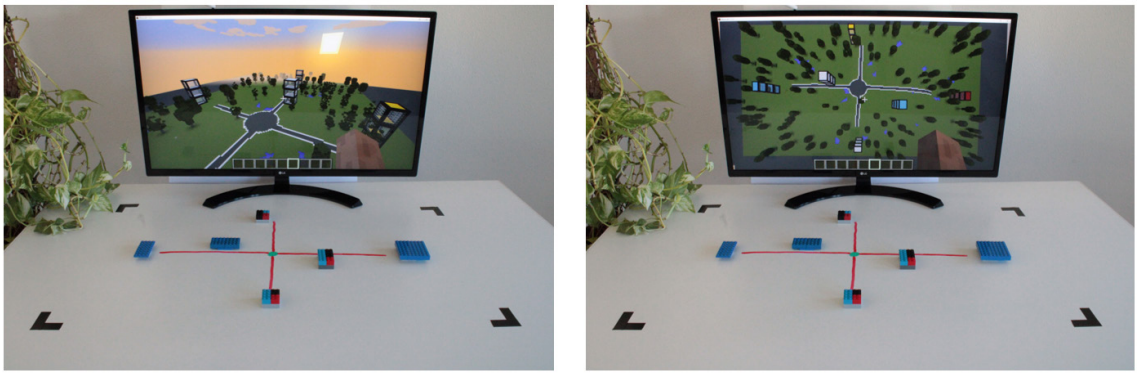


Figura 8.5: Los datos físicos recogidos sobre la mesa se reproducen en el escenario virtual. Además, se añaden otros elementos como árboles o estanques como parte del decorado.

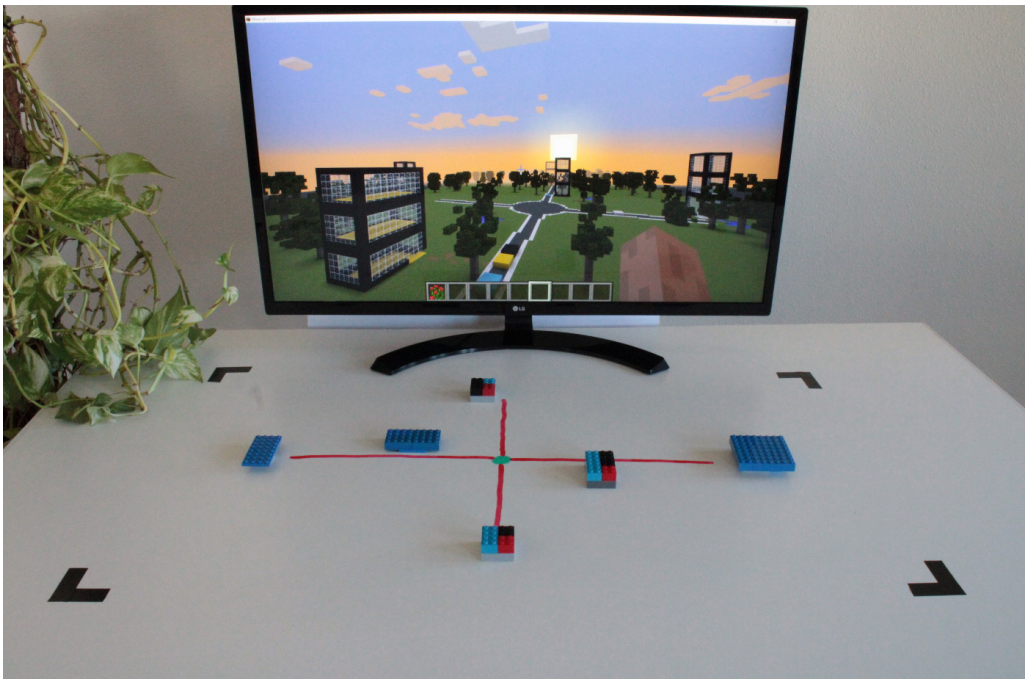


Figura 8.6: Los desplazamientos que resultan de la simulación de movilidad se representan mediante bloques de colores que se desplazan por las carreteras del escenario.

Por último, una vez acabada la definición del modelo físico y su implementación virtual, se da paso a la ejecución de la simulación de movilidad. Los resultados que ofrece Aimsun son los desplazamientos de los vehículos. Estos desplazamientos se representan en el escenario virtual, dando a cada vehículo un color particular que permite diferenciarlos entre sí mientras viajan por las carreteras del modelo. En la Figura 8.6 se muestra un ejemplo de visualización de resultados en Minecraft.

Capítulo 9

Conclusiones y futuros desarrollos

Como se puede extraer del capítulo destinado a resultados —Capítulo 8—, este trabajo posibilita utilizar el producto final para la generación y estudio de escenarios sencillos. De esta forma, se pueden extraer las siguientes conclusiones:

- **Integración**. El proyecto integra de manera adecuada los tres bloques del proyecto: entrada de datos, procesamiento y muestra de resultados.
- **Flexibilidad**. Aunque el campo de aplicación del proyecto es la movilidad urbana, el bloque dedicado al procesamiento puede ser sustituido por otro *software* diferente con el que analizar otros escenarios. Sin embargo, la estructura del proyecto seguiría siendo válida.
- **Robustez**. El proyecto aporta herramientas robustas tanto para la detección de líneas complejas como para la detección de patrones de colores.
- **Fiabilidad**. La posibilidad de integrar el *software* de análisis de movilidad —Aimsun— aporta veracidad a los resultados que pueden obtenerse de la simulación.
- **Modelos híbridos**. La posible falta de detalle a la hora de modelar el entramado urbano —falta de elementos como semáforos, pasos de peatones, etc.— se puede suplir fácilmente introduciendo un escenario mediante la interfaz habitual y trabajar así con un modelo híbrido.
- **Facilidad de uso**. Las herramientas que ofrece el proyecto para comunicarse con el sistema son fáciles de manipular, lo que permite abrir el proceso de diseño.
- **Visualización**. El proyecto ofrece la posibilidad de visualizar los resultados de una forma atractiva.

Pese a que los resultados alcanzados permiten dotar a los usuarios de las herramientas básicas para crear y simular escenarios urbanos, el proyecto puede alcanzar un mayor atractivo y funcionalidad si se desarrollan con mayor profundidad algunos bloques. A continuación se enumeran y exponen los más importantes.

- **Incluir profundidad en el reconocimiento de patrones**. La profundidad de los patrones formados con las piezas de Lego pueden aportar un nuevo dato

de entrada para modelar los edificios: la altura. Al igual que se utiliza el área de los patrones para para aumentar o disminuir la demanda de movilidad, la altura de los edificios también puede ser una variable que permita alterar dicha demanda. La cámara Intel RealSense D435 permite obtener este tipo de información.

- **Enriquecer el modelo de movilidad de Aimsun con más datos de entrada**. Se pueden conseguir modelos más elaborados si se introduce la posibilidad de incluir intersecciones semaforizadas o elegir de forma selectiva el número de carriles de cada sección. Para ello se puede recurrir de nuevo a un sistema de reconocimiento de objetos de tipo YOLO (como el utilizado para los patrones de Lego).

- **Mejorar los gráficos de Minecraft**. Minecraft ofrece mucha libertad y sencillez para representar los escenarios urbanos. Sin embargo, la calidad de los gráficos depende de los *shaders* que se incluyan. Por otro lado, durante la fase de investigación del proyecto, se estudió la posibilidad de incluir gráficos de automóviles gracias a la instalación de *mods*. Sin embargo, la integración Python - Minecraft de la que se hace uso en este proyecto no permite la manipulación de entidades -nombre que da Minecraft a elementos del videojuego que no se catalogan como bloques- provenientes de *mods*. Con estas limitaciones, puede resultar conveniente utilizar otros programas ya mencionado —como Blender o Unity— para mejorar la estética del proyecto a la hora de mostrar los resultados.

- **Flexibilizar la fase de diseño**. Actualmente la fase de diseño atiende a un proceso irreversible en el que primero se definen las calles e intersecciones del modelo y posteriormente se pasa al establecimiento de los edificios. Una vez se han definido las calles y los usuarios se encuentran definiendo los tipos de edificios del modelo, actualmente no es posible volver a la fase de diseño del entramado de calles. Implementar esta función podría flexibilizar el proceso de diseño y facilitar la majora de los modelos que se simulan.

- **Modelar los edificios con datos reales**. En esta primera versión del proyecto, los datos que se han utilizado para modelar los trayectos que se generan de unos puntos de la ciudad a otros no han sido contrastados con bases de datos reales. Para conseguir una representación real de los escenarios que se deseen modelar, este tipo de datos deberían ser incluidos en el modelo, y por tanto, se debería realizar un estudio de modelización de las matrices origen-destino que se incorporan en el bloque de simulación de Aimsun. Algunos recursos útiles pueden provenir del mismo software de simulación, en cuya versión Aimsun Live permite simular los flujos de movilidad en tiempo real en grandes áreas metropolitanas como las ciudades de San Francisco o Sydney [Aim20]. En uno de los estudios que se lleva a cabo para desarrollar esta herramienta [Lin19], se sirven de datos demográficos, encuestas, redes sociales o registros de desplazamientos para conseguir sus predicciones. Tener acceso a este tipo de datos puede resultar muy costoso en términos económicos, pero podría dar pie a sucesivas colaboraciones con Aimsun en proyectos futuros. Otra fuente disponible para mejorar el modelo son los datos de concurrencia de público como los que recoge Google -se incluye un ejemplo en la Figura 9.1,



Figura 9.1: Google muestra el estado de concurrencia de la gran mayoría de establecimientos con etiquetas cualitativas y su evolución temporal a lo largo de la jornada laboral de cada establecimiento

aunque estos datos deberían traducirse de una etiqueta cualitativa como *'un poco concurrido'* a un número cuantitativo-. Otras posibilidades como la obtención de los datos a través de estudios de ayuntamientos o entidades gubernamentales puede ser igualmente válido, aunque conlleve un proceso más tedioso, pues no existe homogeneidad ni en los datos recogidos ni en la forma de exponerlos.

- **Mejorar la comunicación de los diferentes bloques mediante métodos más eficientes**
- **Explorar técnicas de esqueletonización**. Para la detección y extracción de las coordenadas de las líneas o secciones se ha utilizado un algoritmo que puede resultar complejo. Una solución más compacta puede encontrarse en técnicas como la esqueletonización, que permite estrechar una línea con varios píxeles de grosor a una línea de un solo píxel de grosor. Sin embargo, el desafío que aparece aquí es conseguir un algoritmo que recorra y registre todos los puntos -píxeles- de la línea sin incurrir en los problemas descritos en el Capítulo 4.
- **Extracción de información a través de imágenes de Google Maps**. En lugar de crear entramados urbanos a partir los datos que aportan las líneas sobre la mesa, se pueden crear los escenarios a partir de información extraída de mapas.

Apéndice A

Objetivos de desarrollo sostenible

Este proyecto se alinea con varios de los objetivos de desarrollo sostenible marcados por la ONU [Uni15]. A continuación, se citan los más destacados y su relación con este trabajo.

- **Objetivo 11: Lograr que las ciudades sean más inclusivas, seguras, resilientes y sostenibles**. El proyecto trata de ser una herramienta que ayude al diseño de nuevas áreas urbanas o al remodelado de aquellas ya existentes. En los últimos años, el número de personas que vive en áreas urbanas ha incrementado, y se espera que más del 60 % de la población mundial viva en ciudades a partir de 2030. Para dar cabida a esta demanda —no sólo en términos de vivienda, sino también de infraestructuras de comunicación, hospitales, colegios, oficinas, etc.— se requiere que las áreas urbanas se expandan.

  La Historia ha demostrado en numerosas ocasiones cómo el entramado de las ciudades ha quedado obsoleto por cuestiones sociales, tecnológicas y incluso bélicas, por lo que es importante que a la hora de diseñar las ciudades del mañana se tengan en cuenta una gran cantidad de perspectivas que permitan dar con soluciones versátiles, sostenibles y humanas. Este proyecto pretende aportar una plataforma que sirva como un lienzo en blanco para que se produzca el diálogo entre los diferentes agentes implicados en esta tarea.

  La perspectiva con la que está implementado este proyecto es la de analizar y evaluar la movilidad asociada a un entramado urbano y sus condiciones de contorno. De esta forma, con este proyecto se pretende aportar la base para desarrollar una herramienta con la que analizar de manera colaborativa cómo se producen los desplazamientos en las ciudades y extraer las consecuencias que puedan derivarse de ellos, como niveles de contaminación, congestión, optimización de tiempos de trayecto, etc.

- **Objetivo 7: Garantizar el acceso a una energía asequible, segura, sostenible y moderna**. Este objetivo puede no centrarse exclusivamente en el acceso u origen de la energía, sino ampliarse también al uso final que se hace de ella. Las ciudades constituyen núcleos de gran demanda energética,

siendo dos pilares notables de esta demanda la construcción y la movilidad.

En ocasiones, el edificio más sostenible no se considera que es aquel que ha sido concebido bajo los estándares más estrictos de sostenibilidad, sino aquel que después de cumplir con su ciclo de vida inicial, se rehabilita y comienza un ciclo totalmente nuevo [Ech16]. El proceso de rehabilitación permite ahorrar recursos y energías asociados al proceso de demolición y reconstrucción del nuevo edificio. Este proyecto, al contar con un catálogo amplio de edificios y enclaves urbanos que pueden intercambiarse de forma fácil, puede ser una herramienta adecuada para el rediseño de zonas urbanas atendiendo a cuestiones de movilidad, de tal forma que los ayuntamientos puedan reorganizar su estructura urbana manteniendo y adaptando los edificios existentes.

Por otro lado, desde la perspectiva de la movilidad, el consumo energético tiene una gran importancia. En la actualidad, una gran cantidad de trayectos en automóvil son inferiores a 5 km y sirven para transportar dos pasajeros o menos. Si se calcula un *rendimiento* entre el consumo energético asociado al movimiento de la masa de cada automóvil —superior a los 1000 kilogramos— y la energía útil destinada al transporte de las personas, se obtienen coeficientes muy pobres. Para solucionar este comportamiento sociológico —y derroche energético—, dos agentes que juegan un papel importante son el diseño urbano, con propuestas como los barrios autocontenidos, y la oferta de trasporte público que posibilite el desplazamiento entre diferentes puntos de la ciudad de manera eficiente. La mesa interactiva puede resultar una herramienta adecuada para analizar y estudiar estos tipos de propuestas.

- **Objetivo 9: Construir infraestructuras resilientes, promover la industrialización sostenible y fomentar la innovación**. Este punto conecta de nuevo con la necesidad de construir infraestructuras que atiendan a las necesidades actuales, pero en las que se tengan en cuenta también los avances tecnológicos y económicos, o las previsiones demográficas. Como se mencionaba en el Objetivo 7, construir no tiene por qué significar exclusivamente y expandir las ciudades, sino que puede reinterpretarse como una oportunidad de reformar y flexibilizar los límites actuales de la ciudad, sus edificios y sus infraestructuras de comunicación.

- **Objetivo 16: Promover sociedades justas, pacíficas e inclusivas**. Por último, con este proyecto se pretende aportar una plataforma en la que todas las personas, sin importar condiciones como el nivel de formación, tengan acceso a la toma de decisiones y a la obtención de los datos y análisis que respaldan el diseño de la ciudad en la que viven. De esta forma, este proyecto pretende ser un elemento que aporte transparencia y democratice procesos que generalmente son opacos para la ciudadanía.

Apéndice B

Diccionario de Términos

Uno de los objetivos con los que nace este proyecto es la posibilidad de ofrecer una nueva línea de investigación en ICAI dentro del campo de la movilidad, la simulación y la visión artificial. Para ello, resulta fundamental que en el futuro se desarrollen más trabajos que mejoren y perfeccionen los resultados que se exponen en esta memoria.

Con el objetivo de facilitar la lectura y la introducción a esta temática a cualquier lector interesado, y especialmente a otros estudiantes que se acerquen por primera vez a este tipo de trabajos, se aportan las definiciones de algunos términos que podrían resultar extraños si no se tiene una cierta familiaridad con la temática del proyecto.

- **API**. Una *Application Programming Interface* es una interfaz que posibilita la conexión de dos sistemas o programas. Para ello, es necesario que ambos puedan comunicarse y *hablen* el mismo idioma. De esta forma, una API permite flexibilizar los servicios que ofrece un programa o sistema al incorporarle módulos que pueden enriquecer su funcionalidad sin necesidad de realizar modificaciones internas. Por ejemplo, en este proyecto se ha utilizado una API que permite comunicar Minecraft con Python, de tal forma que se consiguen crear los escenarios urbanos o modificar la posición de los automóviles de manera sencilla y de forma automática desde el código del proyecto.

- **Archivo '.ang'**. Un archivo '.ang' es aquel que ha sido guardado con el formato estándar de Aimsun. Para ejecutar correctamente el proyecto, es fundamental que exista un archivo de este tipo preconfigurado y con el *script* de Python preparado para ser ejecutado.

- **Archivo '.xml'**. Los archivos '.xml' recogen información mediante el uso de etiquetas anidadas. Dentro del proyecto, se utilizan archivos '.xml' tanto para compartir información con Aimsun, como para recoger los resultados de sus simulaciones. En la Figura B.1 se muestra un ejemplo de archivo '.xml' generado por Aimsun al ejecutar una simulación. Como se puede apreciar, se genera una etiqueta para cada tiempo de muestreo. En ella aparecen los vehículos que se crean, cambian su posición o finalizan trayecto. Además, se incluyen subetiquetas en las que se recogen otras características como la velocidad, o la posición.

```
<FRAME TIME="00:00:16.000">
        <CREATED>
                <VEH ID="10000001">
                        <TYPE>154</TYPE>
                        <WIDTH>1.872514</WIDTH>
                        <HEIGHT>1.700000</HEIGHT>
                        <SHAPE2D length="4.403177">car</SHAPE2D>
                        <SHAPE3D>./shapes/cars/sports_saloon.3ds.3ds</SHAPE3D>
                        <POS X1="1510.920697" Y1="1091.452165" X2="1506.517542" Y2="1091.437961"/>
                        <SPEED>58.594690</SPEED>
                        <ACCELERATION>0.000000</ACCELERATION>
                </VEH>
        </CREATED>
</FRAME>
<FRAME TIME="00:00:16.800">
        <CHANGED>
                <VEH ID="10000001">
                        <POS X1="1523.941671" Y1="1091.494168" X2="1519.538517" Y2="1091.479964"/>
                        <SPEED>58.594690</SPEED>
                        <ACCELERATION>0.000000</ACCELERATION>
                </VEH>
        </CHANGED>
</FRAME>
<FRAME TIME="00:00:17.600">
        <CHANGED>
                <VEH ID="10000001">
                        <POS X1="1536.318792" Y1="1091.534094" X2="1531.915638" Y2="1091.519891"/>
                        <SPEED>52.799983</SPEED>
                        <ACCELERATION>-2.012051</ACCELERATION>
                        <BRAKES/>
                </VEH>
        </CHANGED>
```

Figura B.1: Fragmento de archivo '.xml' que genera Aimsun para guardar los resultados de las simulaciones.

- **Espacio de color HSV**. Un espacio de color es un sistema de representación de colores. Uno de los espacios de colores más extendidos es el llamado RGB. Frente al espacio de color RGB, el HSV presenta una mayor robustez frente a cambios de luminosidad, por lo que en este proyecto las imágenes se pasan a este espacio de color para extraer información de ellas. Los tres parámetros que definen un color en este sistema son: *hue* o matiz, *saturation* o saturación y *value* o brillo.

- **Google Colab**. [Col] Es un entorno de programación gratuito —limitado a 12 horas de uso consecutivas— que posibilita trabajar con Python en la nube en proyectos asociados a *machine learning*. Google Colab ofrece la posibilidad de utilizar GPUs, lo que permite utilizar todo el potencial que ofrecen en ahorro de tiempo y potencia sin necesidad de sobrecargar la máquina del usuario que utiliza este servicio. En este proyecto, se ha utilizado Google Colab para entrenar el modelo de detección de patrones.

- **GPU**. Una GPU es una unidad de procesamiento gráfico utilizada en aplicaciones como videojuegos o procesamiento de imágenes. En este proyecto se utiliza una de las GPUs que ofrece Google Colab. Su uso permite realizar en menor tiempo el entrenamiento del modelo de detección de patrones. De cara a desarrollos futuros puede resultar interesante la utilización de una GPU en el proyecto para mejorar el número de fotogramas por segundo que actualmente es capaz de procesar al utilizar YOLO v3.

- **IDE**. El término IDE es la sigla en inglés de *integrated development envrionment*. Un IDE es un entorno de desarrollo de código que permite su generación, depuración o ejecución a través de una interfaz gráfica. Un IDE posibilita que el proceso de creación de código sea mucho más ágil con

propiedades como el autocompletado o la localización de errores de forma gráfica.

- **LabelImg**. LabelImg es un software útil para el proceso de etiquetado de objetos en imágenes. Aunque el proceso de generación de imágenes para entrenamiento de un modelo de detección resulta tedioso, el uso de este software permite agilizar la tarea. El software está disponible tanto para Windows y Linux [Lin20b], como para Mac [Lin20a].

- **OpenCV**. [Ope20] Es una librería que permite trabajar con imágenes en proyectos de visión artificial o *machine learning*. En este proyecto se utiliza en numerosas ocasiones para tareas como la aplicación de máscaras de color que permiten extraer datos de imágenes o la carga y configuración del modelo de detección de patrones de colores.

- **Visual Studio Code**. [Cod] Es el entorno de desarrollo utilizado en el proyecto. Ofrece las características mencionadas en la definición de IDE.

- **YOLO v3**. [Far16] YOLO es un sistema de detección de objetos que permite la localización de entidades empleando un menor tiempo que otros métodos basados en redes neuronales y ofreciendo una fiabilidad de detección elevada. YOLO implementa una red neuronal con 24 capas convolucionales seguidas de dos capas completamente conectadas. Al introducir una imagen en un modelo entrenado, YOLO la divide con una malla regular cuadrada. Al mismo tiempo predice contornos con los que delimita las posibles detecciones y les asocia una probabilidad de que la detección realizada sea una de las clases del modelo entrenado. YOLO dispone de las versiones YOLO v1, YOLO v2, YOLO v3 —utilizada en este proyecto— y YOLO v4.

Bibliografía

[Jea24] Charles-Edouard *Le Corbusier* Jeanneret. *Urbanisme La ciudad del futuro*. Éditions Crès, 1924.

[Mul92] John R. Mullin. "The reconstruction of Lisbon following the earthquake of 1755: a study in despotic planning". En: *The Journal of the International History of City Planning Association* Volume 45 (1992).

[Gir16] *Le Corbusier*. José Luis Sert. Jean Giraudoux. *La Charte d'Athènes. Suivi de: entretien avec les étudiants des écoles d'architecture*. Points, (Reedición) 2016.

[Vic11] Carmen Seva Victoria. *París, un recorrido por la historia del urbanismo europeo*. Universidad de Alicante, 2011.

[Kar12] Marius Karthaus. *Javascript implementation of the Ramer Douglas Peucker Algorithm*. 2012. URL: https://karthaus.nl/rdp/.

[Med14] MIT Media Lab. Tangible Media. 2014. URL: https://www.media.mit.edu/videos/labcast-72-transform/.

[Uni15] Naciones Unidas. *Objetivos de desarrollo sostenible*. 2015. URL: https://www.un.org/sustainabledevelopment/es/objetivos-de-desarrollo-sostenible/.

[And16] CityScope Andorra. 2016. URL: https://www.actua.ad/es/cityscope.

[Ech16] EchoLogis. *Construcciones ecológicas. Aprovecharlo todo*. 2016. URL: https://www.youtube.com/watch?v=KsQJkRzdLcY.

[Far16] Joseph Redmon. Santosh Divvala. Ross Girshick. Ali Farhadi. "You Only Look Once: Unified, Real-Time Object Detection". En: (2016).

[Ham16] HafenCity Universität Hamburg. 2016. URL: https://www.hcu-hamburg.de/en/research/citysciencelab/.

[Ros16] Adrian Rosebrock. *Ordering coordinates clockwise with Python and OpenCV*. 2016. URL: https://www.pyimagesearch.com/2016/03/21/ordering-coordinates-clockwise-with-python-and-opencv/.

[all17] Sustainable mobility for all. *Global mobility report 2017. Tracking sector performance*. 2017. URL: http://www.sum4all.org.

[OHa17] Martin O'Hanlon. *Adventures in Minecraft - Setting up your PC*. 2017. URL: https://www.youtube.com/watch?v=lQB5QLt0gPw.

[Sta17] StackOverflow. *Esqueletización de una imagen. ¿Cómo quitar pixeles y completar lineas?* 2017. URL: https://es.stackoverflow.com/questions/109610/esqueletizaci%5C%C3%5C%B3n-de-una-imagen-c%5C%C3%5C%B3mo-quitar-pixeles-y-completar-lineas.

[Hui18] Jonathan Hui. *Real-time Object Detection with YOLO, YOLOv2 and now YOLOv3*. 2018. URL: https://medium.com/@jonathan_hui/real-time-object-detection-with-yolo-yolov2-28b1b93e2088.

[Ros18] Adrian Rosebrock. *Simple object tracking with OpenCV*. 2018. URL: https://www.pyimagesearch.com/2018/07/23/simple-object-tracking-with-opencv/.

[Tan18] Benjamin Tan. *YOLO v3 Demo*. 2018. URL: https://www.youtube.com/watch?v=D_WXxjgQzKk.

[Wha18] Martin O'Hanlon. David Whale. *Adventures in Minecraft*. Wiley, 2018.

[Con19] Fratar Engenharia Consultiva. *Como modelar uma interseção simples no Aimsun Next 8.4*. 2019. URL: https://www.youtube.com/watch?v=K7nGOf8Af40.

[Lin19] Panchamy Krishnakumari. Hans van Lint. Tamara Djukic. Oded Cats. "A data driven method for OD matrix estimation". En: *Delft University of Technology, the Netherlands. Aimsun SL, Barcelona, Spain* (2019).

[Pys19] Pysource. *YOLO object detection using Opencv with Python*. 2019. URL: https://www.youtube.com/watch?v=h56M5iUVgGs.

[Aim20] Aimsun. *Aimsun Live*. 2020. URL: https://www.aimsun.com/aimsun-live/.

[Col20] Colorizer. *Colorizer*. 2020. URL: http://colorizer.org/.

[Con20] Common Objects in Context. 2020. URL: https://cocodataset.org/#home.

[Guy20a] The AI Guy. *YOLOv3 in the CLOUD : Install and Train Custom Object Detector (FREE GPU)*. 2020. URL: https://www.youtube.com/watch?v=10joRJt39Ns&t=1434s.

[Guy20b] The AI Guy. *YOLOv3-Cloud-Tutorial/yolov3*. 2020. URL: https://github.com/theAIGuysCode/YOLOv3-Cloud-Tutorial/tree/master/yolov3.

[Int20] Intel. *Intel RealSense Product Family D400 Series*. 2020. URL: https://www.intelrealsense.com/wp-content/uploads/2020/05/Intel-RealSense-D400-Series-Datasheet-May-2020.pdf.

[Lin20a] TzuTa Lin. *LabelImg*. 2020. URL: https://github.com/tzutalin/labelImg#macOS.

[Lin20b] TzuTa Lin. *LabelImg Download List*. 2020. URL: https://tzutalin.github.io/labelImg/.

[Ope20] OpenCV. *OpenCV*. 2020. URL: https://opencv.org/.

[Pys20] Pysource. *Train YOLO to detect a custom object (online with free GPU)*. 2020. URL: https://www.youtube.com/watch?v=_FNfRtXEbr4.

[Ble] Blender. URL: https://www.blender.org/.

[Cod] Visual Studio Code. URL: https://code.visualstudio.com/.

[Col] Google Colab. URL: https://colab.research.google.com/notebooks/intro.ipynb.

[Lab] MIT Media Lab. URL: https://www.media.mit.edu/.

[Uni] Unity. URL: https://unity.com/es.