

Instituto Tecnológico de Estudios Superiores de Monterrey

Campus Querétaro

**Actividad Integradora**

Docentes

Alejandro Fernández Vilchis

Denisse Lizbeth Maldonado Flores

Pedro Oscar Pérez Murrieta

**Presenta:**

Alan Patricio González Bernal - A01067546

Alan Rodrigo Castillo Sánchez – A01708668

Miguel Ángel Tena García – A01709653

Querétaro, Querétaro México 05 de septiembre del 2023

# Índice

[Índice 2](#_Toc144929430)

[Contexto 3](#_Toc144929431)

[Solución implementada 4](#_Toc144929432)

[Modelado 4](#_Toc144929433)

[Código .ipynb 4](#_Toc144929434)

[Servidor y cliente 13](#_Toc144929435)

[Unity 13](#_Toc144929436)

[Mecanismos de comunicación 14](#_Toc144929437)

[Nivel conceptual 14](#_Toc144929438)

[Nivel código 14](#_Toc144929439)

[Modelos 16](#_Toc144929440)

[Ejecución 16](#_Toc144929441)

[Link del Video 16](#_Toc144929442)

[Conclusión 16](#_Toc144929443)

# Contexto

El sistema es una simulación de una oficina, siendo el tamaño y datos de la oficina siendo brindados por los datos de un .txt. El txt brindará varios tipos de datos diferentes:

1. X
   1. Un obstáculo, lugar inaccesible
2. Valor numérico (0-9)
   1. La cantidad de basura que existe en esa celda
3. S
   1. El punto de partida de los robots que a partir de ahora serán llamados “aspiradoras”
4. P
   1. La papelera, punto en el que las aspiradoras deben dejar la basura antes de seguir recogiendo más
5. R
   1. La aspiradora en cuestión. Esta se identificará de las demás por un número a lado de R.

Las limitaciones del sistema son las siguientes:

1. Cada aspiradora solo puede recoger 5 unidades de basura antes de tener que ir a vaciar todo en la papelera.
2. Las aspiradoras no pueden estar en la misma celda varias a la vez, solo en el punto S esto es permitido.
3. Las aspiradoras no conocen su alrededor, por lo que tendrán que mapear todo o moverse de forma aleatoria.
4. Las aspiradoras no pueden pasar por los puntos X
5. Las aspiradoras no pueden saber la cantidad de basura que tengan las celdas vecinas, Solo pueden saber la cantidad exacta una vez estén en esa celda.

Tomando en cuenta todas estas limitaciones y lo aprendido en clase, se desarrolló una solución la cual toma en cuenta librerías de Python como Mesa, Matplotlib, numpy, pandas, etc. Para poder generar una solución que satisfaga lo que se indica y cumpla con las limitaciones brindadas. Esta solución se analizará más adelante a lo largo de este documento.

# Solución implementada

La solución generada incluye varios pasos.

## Modelado

Para la solución de este problema, primero tuvimos que pensar, diseñar y proponer algunas estrategias o algoritmos que nos permitieran mapear de una forma eficiente y posteriormente recolectar la basura e irla dejando en la papelera igualmente de una manera eficaz.

Principalmente decidimos que el movimiento de robots a celdas destino deberán ser “planeados” y no al azar, sino con un algoritmo capaz de tomar la mejor decisión en cuestión de costes para el recorrido de cada robot, y con esto llegamos a un algoritmo de A\* (A estrella) que principalmente se encarga de calcular la cercanía a el nodo destino y la del nodo inicial, y al sumarlas nos daría un tipo de coste medio total a seguir por dichos caminos, y conseguir así el que menos tenga.

Para la parte del mapeo nos surgió idea basada básicamente en una forma en la cual se aprovechan (en este caso) las celdas de los laterales del robot que puede ver, para usar al máximo posible el rango de visión y mapear con respecto a ellos, intentando separar a cada agente por 2 celdas, permitiendo que estas las mapeen cada agente al avanzar en paralelo barriendo el área de arriba hacia abajo (la idea principal es esta). Luego nos dimos cuenta de que no van a moverse en líneas rectas hacia abajo debido a los obstáculos, sin embargo notamos que igualmente era una manera eficiente, puesto que al buscar como llegar hasta lo más abajo posible, igualmente recorrerá celdas que otros agente no y los demás igual. Posteriormente nos topamos con un problema por la cantidad de columnas del mapa, ya que no todas eran perfectamente divisibles entre 3, por lo que no podía aprovecharse al máximo esta eficacia del campo de visión de los robots, sin embargo, notamos que podríamos hacer un desplazamiento de 2 en 2 en lugar de 3 (dejando solo 1 celda entre cada agente) y de igual manera aprovechar, aunque no en su totalidad, el mapeo de cada agente.

En cuanto a la recolección de la basura básicamente decidimos que cada robot escogiera la celda más cercana con basura, y se dirigiera a ella mediante un camino determinado por el algoritmo de A\*, y que una vez seleccionada la “celda objetivo”, ningún otro robot pueda tomarla como objetivo, y así evitar colisiones, bloqueos o ineficiencia por la torpeza en el movimiento de los robots.

## Código .ipynb

Utilizando las libretas de jupyter, se comenzó el desarrollo para la solución de este problema. Principalmente, se utilizó para testeo el siguiente archivo .txt

mapa.txt

6 5

S 4 0 6 0

6 X 0 X 4

0 X 7 X 0

5 0 0 X 8

0 0 X 0 0

0 0 0 0 P

Tomando en cuenta este archivo de entrada, se generó la solución. Primero se realizaron los imports y las lecturas de mapa.txt. Aquí se generaron dos veces la lectura del mapa:

def leer\_mapa(*filename*):

    with open(*filename*, 'r') as f:

        lines = f.readlines()

        mapaREAL = [list(line.replace(" ", "").strip()) for line in lines[1:]]

    filas = len(mapaREAL)

    columnas = len(mapaREAL[0])

    mapaUNKNOWN = [fila[:] for fila in mapaREAL]  *# Crear una copia de mapaREAL*

    for i in range(filas):

        for j in range(columnas):

            if mapaREAL[i][j] not in ['S', 'P']:

                mapaUNKNOWN[i][j] = '?'

    return mapaREAL, mapaUNKNOWN

La función **leer\_mapa** toma un archivo de texto como entrada y genera dos versiones del mapa: **mapaREAL** y **mapaUNKNOWN**. El primero es un mapa con toda la información, y el segundo es lo que las aspiradoras ven inicialmente: todos los elementos que no son "S" o "P" se reemplazan por "?".

A partir de este punto, se realiza un grafo del mapa, siendo los puntos marcados como X nodos inexistentes. Después de esto, se implementa el algoritmo A\*, el cual le permite a las aspiradoras encontrar el camino más próximo a sus objetivos.

def construir\_grafo(*mapa*):

    grafo = {}

    filas = len(*mapa*)

    columnas = len(*mapa*[0])

    for i in range(filas):

        for j in range(columnas):

            if *mapa*[i][j] != 'X':

                nodo = (i, j)

                vecinos = []

                if i - 1 >= 0 and *mapa*[i-1][j] != 'X': vecinos.append((i-1, j))

                if i + 1 < filas and *mapa*[i+1][j] != 'X': vecinos.append((i+1, j))

                if j - 1 >= 0 and *mapa*[i][j-1] != 'X': vecinos.append((i, j-1))

                if j + 1 < columnas and *mapa*[i][j+1] != 'X': vecinos.append((i, j+1))

                if len(vecinos) > 0:

                    grafo[nodo] = vecinos

                else:

                    print(f'El nodo {nodo} no es accesible')

    return grafo

construir\_grafo toma un mapa y crea un grafo en el que cada nodo representa una posición en el mapa que no es una pared (X).

def heuristic(a, b):

    return abs(a[0] - b[0]) + abs(a[1] - b[1])

*# Implementación del algoritmo A\**

def a\_star\_search(graph, start, goal):

    frontier = []

    heapq.heappush(frontier, (0, start))

    came\_from = {}

    cost\_so\_far = {}

    came\_from[start] = None

    cost\_so\_far[start] = 0

    while frontier:

        \_, current = heapq.heappop(frontier)

        if current == goal:

            break

        for next in graph[current]:

            new\_cost = cost\_so\_far[current] + 1

            if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:

                cost\_so\_far[next] = new\_cost

                priority = new\_cost + heuristic(goal, next)

                heapq.heappush(frontier, (priority, next))

                came\_from[next] = current

*# Reconstruct path*

    current = goal

    path = []

    while current != start:

        path.append(current)

        current = came\_from[current]

    path.append(start)

    path.reverse()

    return path

La función heuristic se utiliza en el algoritmo A\* para estimar el costo desde un nodo hasta el objetivo. La función a\_star\_search implementa el algoritmo A\* para encontrar el camino más corto en el grafo.

Después de esto, se realiza la definición de agentes, que es donde definimos el comportamiento del agente de los robots.

class Robot(Agent):

    def \_init\_(*self*, *unique\_id*, *model*, *inicio*):

        super().\_init\_(*unique\_id*, *model*)

*self*.posicion = *inicio*

*self*.capacidad = 5

*self*.mapeoIniciado = False

*self*.mapeoIniTerminado = False

*self*.mapeoTerminado = False

*self*.pasos = 0

*self*.conteo\_fin = False

El agente cuenta con 5 definiciones: mapeoInicial(), actualizarMapa(), mover(), recogerBasura(), step().

mapeoInicial():

Las aspiradoras salen del punto de inicio, del cual se distribuyen entre las columnas para así poder realizar el mapeo de forma eficiente. Una vez que estas aspiradoras iniciales terminan el mapeo se considera que esta definición ha terminado.

 def mapeoInicial(*self*):

        mapaU = *self*.model.mapaUNKNOWN

        mapaR = *self*.model.mapaREAL

        columnas = len(mapaU[0])

        filas = len(mapaU)

        if columnas % 3 == 0:

            agentesIniciales = int(np.ceil(columnas / 3))

            desplazamiento = 3

        else:

            agentesIniciales = int(np.floor(columnas / 2))

            desplazamiento = 2

        posicionInicial = (0, 1)

        if *self*.unique\_id in range(agentesIniciales) and not *self*.mapeoIniTerminado:

            if *self*.unique\_id == 0:

                if mapaR[posicionInicial[0]][posicionInicial[1]] == 'X':

                    posicionInicial = (posicionInicial[0] + 1, posicionInicial[1])

                destino = (filas - 1, 1)

                if mapaR[destino[0]][destino[1]] == 'X':

                    destino = (destino[0] - 1, destino[1])

            else:

                posicionInicial = (posicionInicial[0], posicionInicial[1] + (desplazamiento \* *self*.unique\_id))

                if mapaR[posicionInicial[0]][posicionInicial[1]] == 'X':

                    posicionInicial = (posicionInicial[0] + 1, posicionInicial[1])

                destino = (filas - 1, posicionInicial[1])

                if mapaR[destino[0]][destino[1]] == 'X':

                    destino = (destino[0] - 1, destino[1])

            return posicionInicial, destino

        else:

            inicialesTerminados = True  *# Variable para verificar si todos los agentes iniciales terminaron*

            for agente in *self*.model.schedule.agents[:agentesIniciales]:

                if not agente.mapeoIniTerminado:  *# Si el agente no ha terminado*

                    inicialesTerminados = False  *# Cambiamos la variable a False*

                    break

            if inicialesTerminados:

*# Si todos los agentes iniciales terminaron, se verifica que no queden ? en el mapa*

                terminaMapeo = True

                camino\_corto = None

                for i in range(filas):

                    for j in range(columnas):

                        if mapaU[i][j] == '?' and mapaR[i][j] != 'X' and not any((i, j) == valor for clave, valor in *self*.model.pos\_obj.items() if clave != *self*.unique\_id):

                            terminaMapeo = False

*self*.mapeoTerminado = False

*# Si hay ? se activa un agente auxiliar para que termine el mapeo*

*self*.mapeoIniciado = True

                            caminoM = a\_star\_search(*self*.model.grafo, *self*.posicion, (i, j))

                            if camino\_corto is None or len(caminoM) < len(camino\_corto):

                                camino\_corto = caminoM

                if camino\_corto is None:

                    camino\_corto = [*self*.model.Inicio]

*# Moverse hacia la ? más cercana*

                if len(camino\_corto) > 1:

*self*.model.pos\_obj[*self*.unique\_id] = camino\_corto[-1]

                    return *self*.posicion, camino\_corto[1]

*# Si ya no hay ? se establece que el mapeo ha terminado*

                if terminaMapeo:

*self*.mapeoTerminado = True

            return None, None

actualizarMapa():

Como el nombre lo indica, se actualiza el mapa tomando en cuenta los nodos vecinos, de esta forma, actualiza la vision que tiene el agente del mapa en función de su posición actual.

    def actualizarMapa(*self*):

*self*.model.mapaUNKNOWN[*self*.posicion[0]][*self*.posicion[1]] = *self*.model.mapaREAL[*self*.posicion[0]][*self*.posicion[1]]

*# Vecinos del nodo actual (Pero incluyendo nodos con obstaculos)*

        pos\_vecinos = []

        i, j = *self*.posicion

        if i - 1 >= 0: pos\_vecinos.append((i-1, j))

        if i + 1 < len(*self*.model.mapaUNKNOWN): pos\_vecinos.append((i+1, j))

        if j - 1 >= 0: pos\_vecinos.append((i, j-1))

        if j + 1 < len(*self*.model.mapaUNKNOWN[0]): pos\_vecinos.append((i, j+1))

        for pos\_vecino in pos\_vecinos:

            if *self*.model.mapaREAL[pos\_vecino[0]][pos\_vecino[1]] == 'X':

*self*.model.mapaUNKNOWN[pos\_vecino[0]][pos\_vecino[1]] = 'X'

            elif *self*.model.mapaREAL[pos\_vecino[0]][pos\_vecino[1]] == '0':

*self*.model.mapaUNKNOWN[pos\_vecino[0]][pos\_vecino[1]] = '0'

            elif *self*.model.mapaUNKNOWN[pos\_vecino[0]][pos\_vecino[1]] == '?':

*self*.model.mapaUNKNOWN[pos\_vecino[0]][pos\_vecino[1]] = 'B'

mover():

Función que se repetirá a cada step, permite al agente cambiar de posición y aumentar 1 a su contador de pasos si esta, está disponible, además de llamar a la función de actualizar mapa.

    def mover(*self*, *nueva\_posicion*):

        if *nueva\_posicion* not in *self*.model.pos\_ocupadas:

            if *self*.posicion in *self*.model.pos\_ocupadas:

*self*.model.pos\_ocupadas.remove(*self*.posicion)

            if *self*.posicion != *self*.model.Inicio:

*self*.model.pos\_ocupadas.append(*nueva\_posicion*)

*self*.posicion = *nueva\_posicion*

*self*.pasos += 1

        else:

            for posicion\_alterna in *self*.model.grafo[*self*.posicion]:

                if posicion\_alterna != *nueva\_posicion* and posicion\_alterna not in *self*.model.pos\_ocupadas:

*self*.mover(posicion\_alterna)

*self*.actualizarMapa()

recogerBasura():

Si está en una posición donde exista basura, recoge la cantidad que le sea posible y actualiza el mapa para representar la cantidad que no se pudo recoger o en su defecto un 0.

def recogerBasura(*self*):

        if *self*.model.mapaUNKNOWN[*self*.posicion[0]][*self*.posicion[1]].isdigit() and int(*self*.model.mapaUNKNOWN[*self*.posicion[0]][*self*.posicion[1]]) > 0:

            basura = min(int(*self*.model.mapaUNKNOWN[*self*.posicion[0]][*self*.posicion[1]]), *self*.capacidad)

*self*.capacidad -= basura

*self*.model.mapaUNKNOWN[*self*.posicion[0]][*self*.posicion[1]] = str(int(*self*.model.mapaUNKNOWN[*self*.posicion[0]][*self*.posicion[1]]) - basura)

*self*.model.mapaREAL[*self*.posicion[0]][*self*.posicion[1]] =  *self*.model.mapaUNKNOWN[*self*.posicion[0]][*self*.posicion[1]]

            if int(*self*.model.mapaUNKNOWN[*self*.posicion[0]][*self*.posicion[1]]) > 0:

*self*.model.pos\_obj[*self*.unique\_id] = [] *# Elimina la posición de la basura de la lista de posiciones de objetos*

                return

step():

En esta definición se llaman las demás definiciones, de esta forma con llamar solo una función, por cada paso se llamarán las que se requieran.

    def step(*self*):

*# Mapeo inicial*

        if *self*.mapeoTerminado == False:

            posicionInicial, destino = *self*.mapeoInicial()

            if posicionInicial is not None and destino is not None:

                if *self*.mapeoIniciado == False:

                    recorrido = a\_star\_search(*self*.model.grafo, *self*.posicion, posicionInicial)

                    if len(recorrido) > 1:

*self*.mover(recorrido[1])

                    if *self*.posicion == posicionInicial:

*self*.mapeoIniciado = True

                elif *self*.posicion != destino and *self*.mapeoIniciado == True:

                    recorrido = a\_star\_search(*self*.model.grafo, *self*.posicion, destino)

                    if len(recorrido) > 1:

*self*.mover(recorrido[1])

                    if *self*.posicion == destino:

*self*.recogerBasura()

*self*.mapeoIniTerminado = True

*# Si la capacidad es mayor a 0, se dirige a la basura más cercana*

        else:

*# Si la capacidad es 0, se dirige a la papelera para vaciarse*

            if *self*.capacidad == 0:

                papelera = a\_star\_search(*self*.model.grafo, *self*.posicion, *self*.model.Papelera)

*self*.mover(papelera[1])

                if *self*.posicion == *self*.model.Papelera:

*self*.capacidad = 5

*self*.mover(*self*.model.grafo[*self*.posicion][0])

            else:

*# Encuentra la basura más cercana usando A\**

                camino\_corto = None

                for i in range(len(*self*.model.mapaUNKNOWN)):

                    for j in range(len(*self*.model.mapaUNKNOWN[0])):

                        if (*self*.model.mapaUNKNOWN[i][j] == 'B' or (*self*.model.mapaUNKNOWN[i][j].isdigit() and int(*self*.model.mapaUNKNOWN[i][j]) > 0)) and not any((i, j) == valor for clave, valor in *self*.model.pos\_obj.items() if clave != *self*.unique\_id):

                            camino = a\_star\_search(*self*.model.grafo, *self*.posicion, (i, j))

                            if camino\_corto is None or len(camino) < len(camino\_corto):

                                camino\_corto = camino

                if camino\_corto is None:

                    camino\_corto = [*self*.model.Inicio]

*# Moverse hacia la basura más cercana*

                if len(camino\_corto) > 1:

*self*.model.pos\_obj[*self*.unique\_id] = camino\_corto[-1]

*self*.mover(camino\_corto[1])

                    if *self*.posicion == camino\_corto[-1]:

*self*.recogerBasura()

A partir de aquí, se comienza la sección referente al modelo. Es en este punto donde definimos cosas como la cantidad de aspiradoras, su activación, los mapas, etc.

def update(*num*, *model*):

*model*.step()

class LimpiezaModel(Model):

    def \_init\_(*self*, *mapa\_txt*):

*self*.num\_robots = 5

*self*.schedule = RandomActivation(*self*)

*self*.mapaREAL, *self*.mapaUNKNOWN = leer\_mapa(*mapa\_txt*)

*self*.grafo = construir\_grafo(*self*.mapaREAL)

*self*.pos\_ocupadas = []

*self*.pos\_obj = {}

*# Variables para ejecución*

*self*.steps = 0

*self*.running = True

        for i in range(len(*self*.mapaREAL)):

            for j in range(len(*self*.mapaREAL[0])):

                if *self*.mapaREAL[i][j] == 'S':

*self*.Inicio = (i, j)

                elif *self*.mapaREAL[i][j] == 'P':

*self*.Papelera = (i, j)

*# Establecer el tamaño de la figura*

*self*.fig, *self*.ax = plt.subplots(*figsize*=(25, 25))

*# Desactivar los ejes*

*self*.ax.axis('off')

*# Creación de robots*

        for i in range(*self*.num\_robots):

            robot = Robot(i, *self*, *self*.Inicio)

*self*.schedule.add(robot)

*self*.pos\_obj[i] = []

El modelo cuenta con solo dos definiciones, la de draw\_map() y step().

draw\_map():

Se realiza la animación de los agentes y de los demás detalles del mapa. Se le asigna un color, tamaño, etc.

   def draw\_map(*self*):

*self*.ax.clear()

*self*.ax.set\_xlim(0, len(*self*.mapaREAL[0]))  *# Ajustar límites del eje X*

*self*.ax.set\_ylim(len(*self*.mapaREAL), 0)  *# Ajustar límites del eje Y*

        for i, row in enumerate(*self*.mapaUNKNOWN):

            for j, cell in enumerate(row):

                if cell == 'X':

*self*.ax.text(j, i, cell, *ha*='center', *va*='center', *fontsize*=45, *color*='black')

                elif cell == 'S' or cell == 'P':

*self*.ax.text(j, i, cell, *ha*='center', *va*='center', *fontsize*=45, *color*='green')

                elif cell == '?':

*self*.ax.text(j, i, cell, *ha*='center', *va*='center', *fontsize*=45, *color*='grey')

                else:

*self*.ax.text(j, i, cell, *ha*='center', *va*='center', *fontsize*=45, *color*='blue')

        for agent in *self*.schedule.agents:

*self*.ax.text(agent.posicion[1], agent.posicion[0], 'R'+ str(agent.unique\_id), *ha*='center', *va*='center', *fontsize*=35, *color*='red')

step():

Se realizan los pasos para ejecutar el programa.

    def step(*self*):

*self*.schedule.step()

*self*.draw\_map()

*self*.steps += 1

El último bloque es el responsable de la animación.

model = LimpiezaModel('mapa.txt')

ani = animation.FuncAnimation(model.fig, update, *fargs*=(model,), *frames*=50)

ani.save('Output3.gif', *writer*='imagemagick', *fps*=10)

## Servidor y cliente

Para uso de esta clase, los profesores nos brindaron un código de Python el cual actúa como servidor, desplegándose en el puerto 3535. Para Unity se nos brindó un cliente en C# el cual nos permite desplegar lo que se genera en el archivo de Python en Unity.

## Unity

En el motor gráfico de Unity se realizaron pasos muy simples. Se generaron sprites para cada elemento de la simulación y a un objeto vacío se le añadió el script de WebClient, permitiendo que el código de Python funcione en Unity.

# Mecanismos de comunicación

Para comunicarse entre agentes, se genera un mapa “vacío”, el cual es el que se marca con ? los puntos que no se conocen, a excepción de los puntos S y P que siempre se sabrán. A partir de aquí, las aspiradoras comenzarán su mapeo, el cual actualiza ese mapa con datos reales para generar un mapa completo.

## Nivel conceptual

En el área de Servidor y cliente, nuestro código, al utilizar un grafo en lugar de una grid, el sistema tiene más facilidad de pasar las posiciones de los elementos a través de un JSON y un método POST, el cliente con un GET recibe esa información y la interpreta en Unity, brindando así la solución final de este reto.

## Nivel código

Se generan dos servidores, uno manda las posiciones xy y el id de los robots, el otro manda el mapa.

 def do\_POST(self):

        model.step()

        content\_length = int(self.headers['Content-Length'])

        post\_data = self.rfile.read(content\_length).decode('utf-8')

        response\_data = model.datos

En el Post mandamos el valor de model.datos la cual mandará específicamente la información que se comentó anteriormente. Una vez que esos datos se envían, mediante RobotController y GarbageController se definen que valores tomar exactamente y cómo trabajarlas.

GameObject robot = robots[id];

            robot.transform.position = new Vector3(x, y - 0.3f, 0);

            }

Con esta línea, se generan las copias del robot, y se le mueven en sus posiciones x, y y z.

En el área del mapa, se entrega todo el mapa como string, a partir de ahí, el GarbageController itera paso a paso ese mapa, para saber qué se hará dependiendo de la posición.

for (int i = 0; i < rowCount; i++)

                    {

                        for (int j = 0; j < colCount; j++)

                        {

                            Debug.Log(matriz[i, j]);

                            Vector3 position = new Vector3(i \* spacing, j \* spacing, 0);

                            if (matriz[i, j] >= 1 && matriz[i, j] <= 9)

                            {

                                Vector3 positionGarbage = position + new Vector3(0, 0, 0);

                                Instantiate(Garbage, position, Quaternion.identity);

                            }

                            else if (matriz[i, j] == -1)

                            {

                                Vector3 positionStarting = position + new Vector3(0, 0, 0);

                                Instantiate(Starting, position, Quaternion.identity);

                                Debug.Log("Starting");

                            }

                            else if (matriz[i, j] == -2)

                            {

                                Vector3 positionBote = position + new Vector3(0, 0, 0);

                                Instantiate(Bote, position, Quaternion.identity);

                                Debug.Log("Bote");

                            }

                            else if (matriz[i, j] == -3)  *// Usando la constante definida*

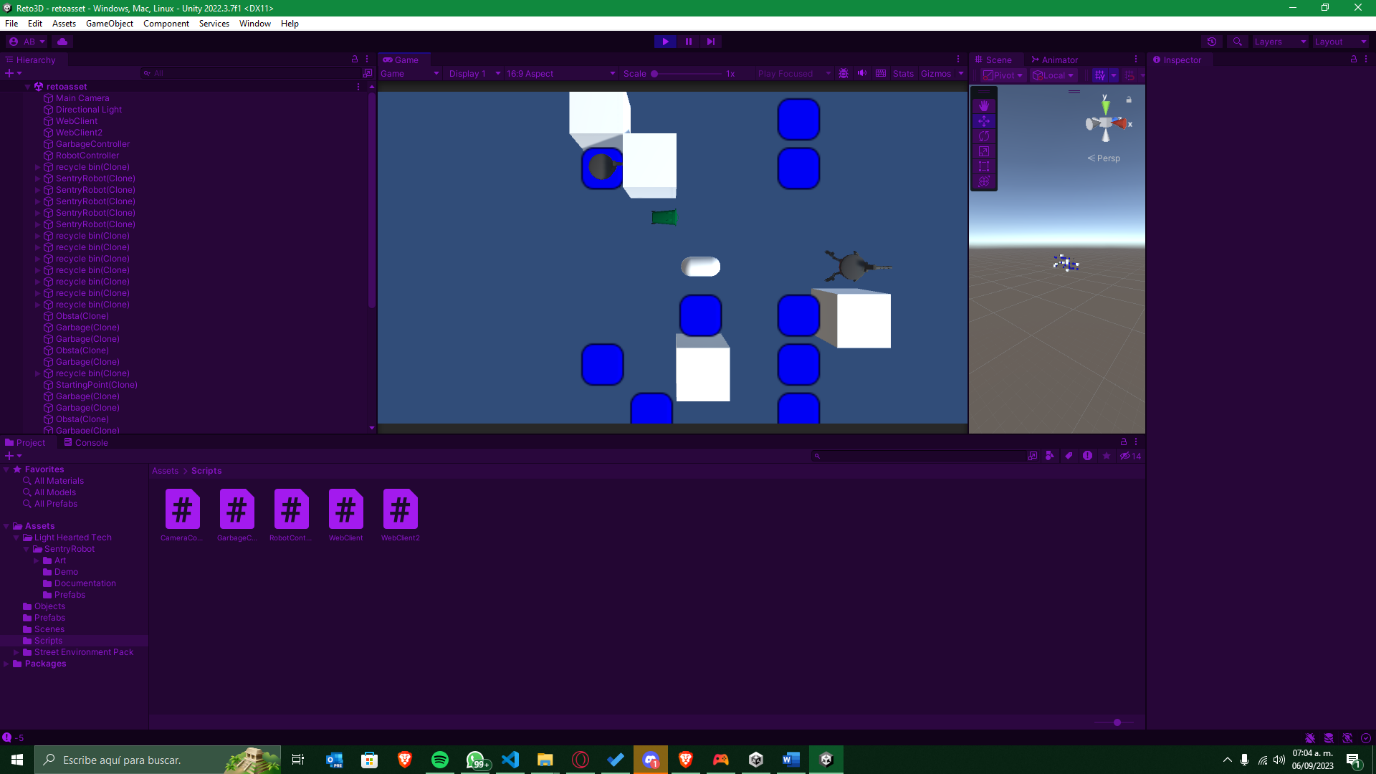
                            {

                                Vector3 positionObsta = position + new Vector3(0, 0, 0);

                                Instantiate(Obsta, position, Quaternion.identity);

                            }

Aquí el GabageController define qué objeto será cada valor numérico, con la posibilidad de varios no tener objeto.

En Unity esto se ve algo así:  
  


Todo este movimiento se genera directamente desde el código de GarbageController y RobotController. La cámara se puede mover en el modo juego con las flechas direccionales.

# Modelos

Para los objetos como el robot, la basura, el punto S, P, y los obstáculos, lo que se realiza es que se generaron dos objetos vacíos, uno se encarga de generar las instancias del robot (siendo este objeto el RobotController) y otro objeto vacío, el cual realiza las instancias del punto S, P, obstáculo y basura (Siendo este último el GarbageController). Estos objetos se generan gracias a la información brindada por los códigos de C# con su respectivo nombre.

# Ejecución

Para ejecutar el código, se debe ejecutar primeramente ambos servidores, en dos terminales distintas, se deben ejecutar estos comandos:

Python3 tc2008B\_server.py 8585

Python3 tc2008B\_server2.py 8586

Finalmente, se debe abrir el proyecto en unity, solo se debe presionar el boton start. Con las flechas direccionales se mueve la cámara y listo! Disfruta de la simulación.

# Link del Video

<https://youtu.be/dtAo3Ntw5FI>

# Conclusión

Después de generar la solución exitosa para este reto, se puede concluir que el uso de agentes y modelos nos son muy útiles para generar simulaciones, definiendo comportamientos y otras limitantes, además del uso de motores gráficos como lo es Unity, siendo este uno de los más amigables para usuarios que a penas se inician en el trabajo de este tipo de software.