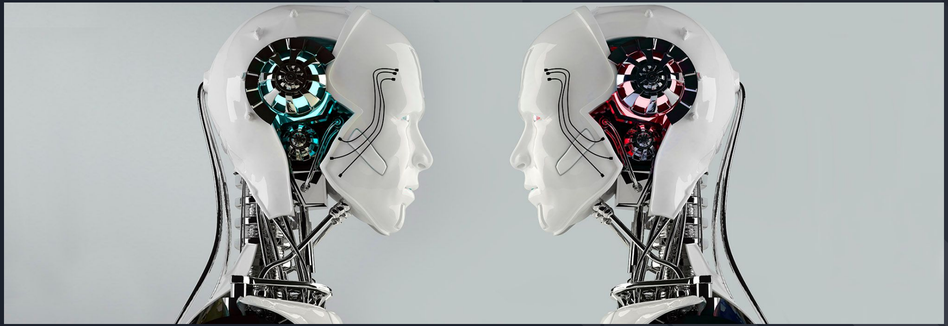
Reto Movilidad Urbana

Narhari Olalde Guajardo	A01284077
Guillermo Tafoya Milo	A01633790
Mauricio Maqueo Huerta	A01620649
Emilio Yoltic Martinez	A01620000

Introducción



Contexto del reto

La movilidad urbana, se define como la habilidad de transportarse de un lugar a otro y es fundamental para el desarrollo económico y social y la calidad de vida de los habitantes de una ciudad. Desde hace un tiempo, asociar la movilidad con el uso del automóvil ha sido un signo distintivo de progreso. Sin embargo, esta asociación ya no es posible hoy.

El crecimiento y uso indiscriminado del automóvil —que fomenta políticas públicas erróneamente asociadas con la movilidad sostenible—genera efectos negativos enormes en los niveles económico, ambiental y social en México. Como ya se mencionó anteriormente, durante las últimas décadas, ha existido una tendencia alarmante de un incremento en el uso de automóviles en México.

Los Kilómetros-Auto Recorridos (VKT por sus siglas en Inglés) se han triplicado, de 106 millones en 1990, a 339 millones en 2010. Ésto se correlaciona simultáneamente con un incremento en los impactos negativos asociados a los autos, como el smog, accidentes, enfermedades y congestión vehicular.

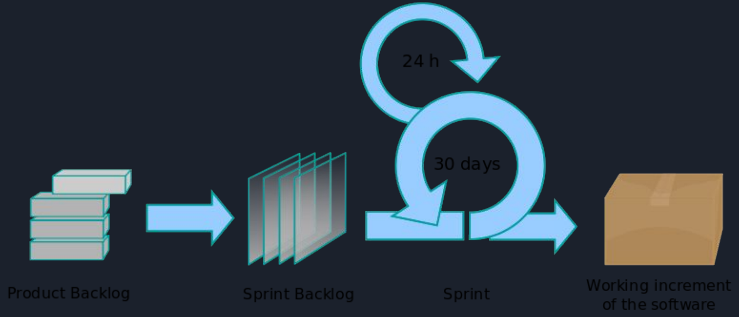
Objetivo

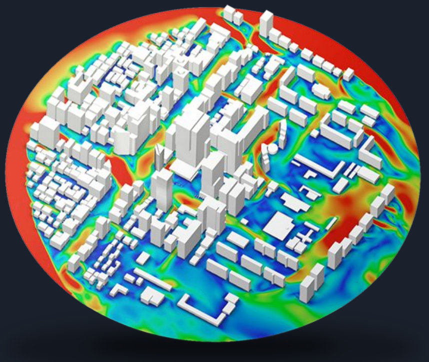
Principal: proponer una solución al problema de movilidad urbana en México, mediante un enfoque que reduzca la congestión vehicular al simular de manera gráfica el tráfico, representando la salida de un sistema multi agentes.

- Unity y Python

Procesos

- Establecer las bases
  - Agentes
  - Diagramas
- Realizar un plan de trabajo
- Diseño de la simulación
- Diseño de los aspectos gráficos





Agentes

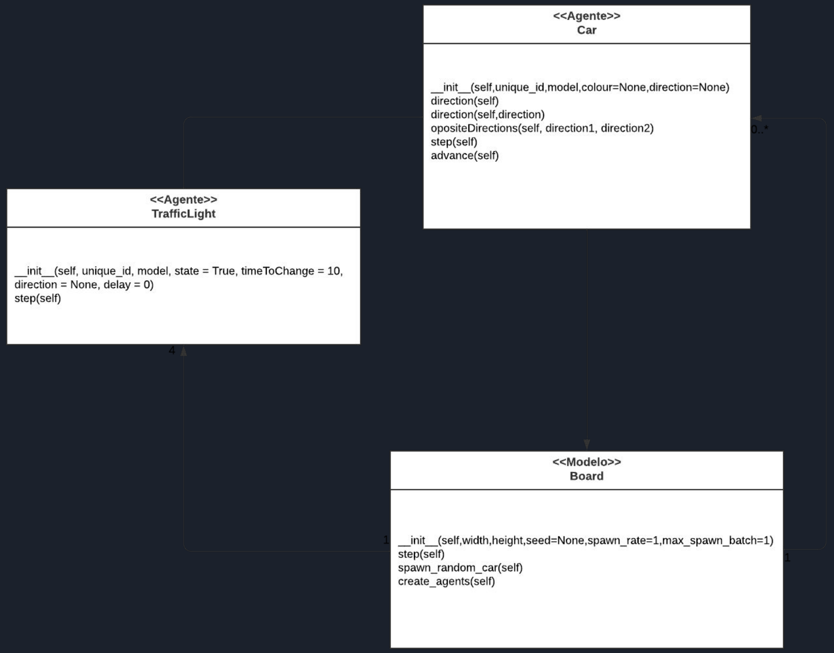
Agentes

- Agentes Involucrados
  - Autos
  - Semáforos
- Entorno
  - Calles

Diagramas

Diagrama de clase

Diagrama de Clases



<<Agente>>
Car

__init__(self,unique_id,model,colour=None,direction=None)
direction(self)
direction(self,direction)
opositeDirections(self, direction1, direction2)
step(self)
advance(self)

<<Agente>>
TrafficLight

__init__(self, unique_id, model, state = True, timeToChange = 10, direction = None, delay = 0)
step(self)

<<Modelo>>
Board

__init__(self,width,height,seed=None,spawn_rate=1,max_spawn_batch=1)
step(self)
spawn_random_car(self)
create_agents(self)

Diagrama de protocolos de interacción (Auto)

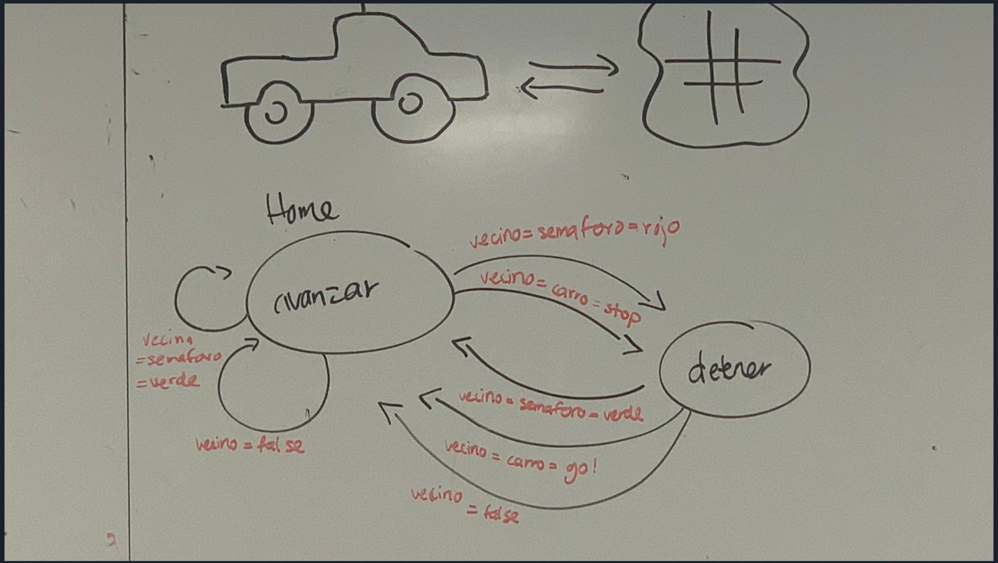
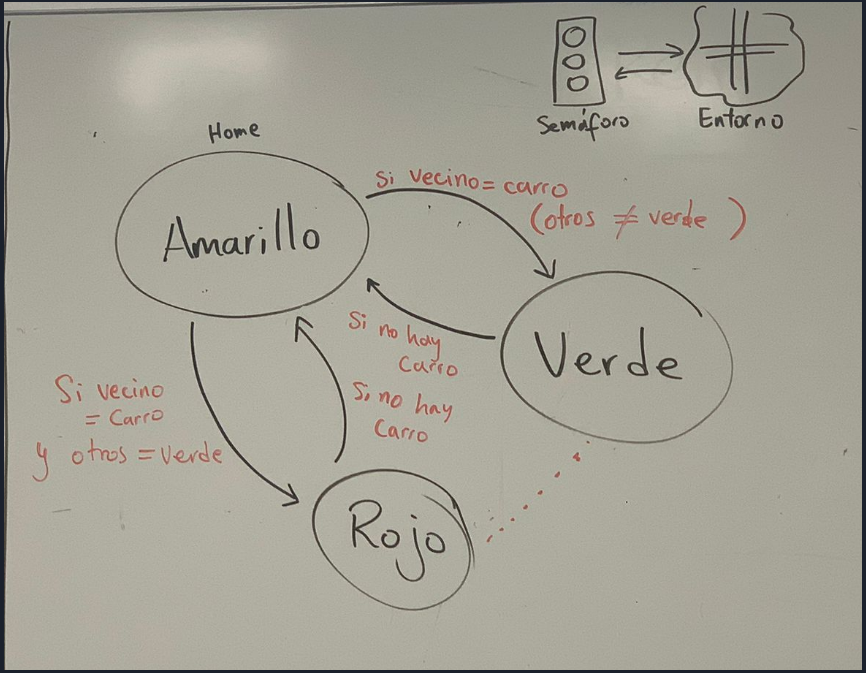


Diagrama de protocolos de interacción (Semáforo)



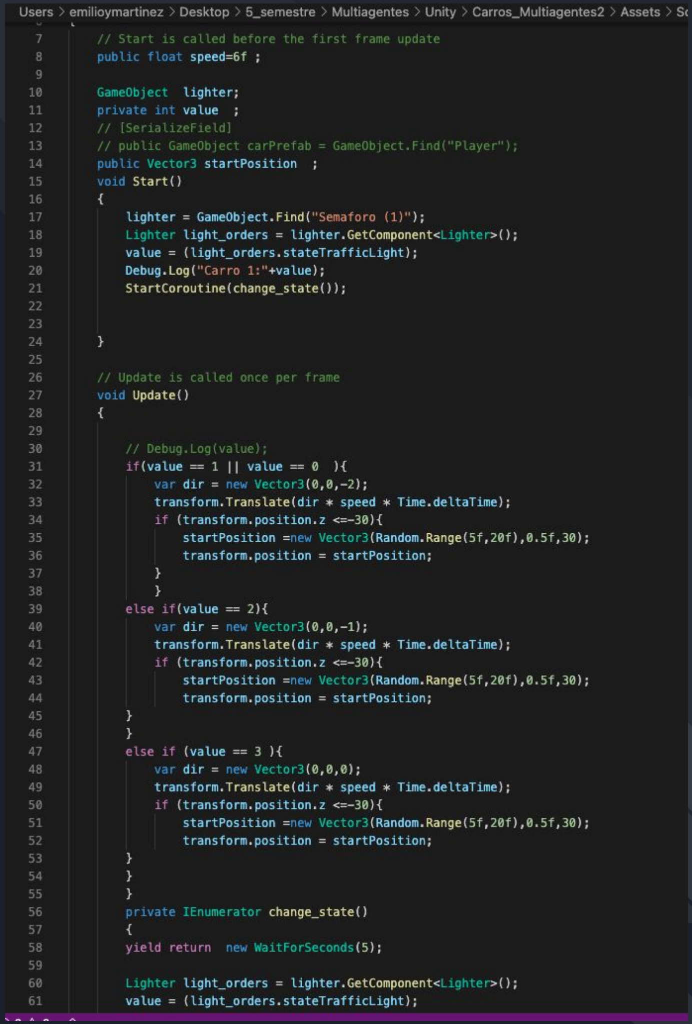
Plan de trabajo

Metodología SCRUM

- Sprints
  - Sprint 2
    - Modelación de Calles con atributos básicos
    - Modelación de Carros con atributos básicos
    - Modelación de Semáforos con atributos básicos
    - Creación de clase carro con sus instrucciones propias
    - Creación de clase semaforo con sus instrucciones propias
    - Creación de clase sensor con sus instrucciones propias
  - Sprint 3
    - Establecer el modelo en Unity con Spawn Points y colisiones
    - Codificar interacciones entre las clases en Python
  - Sprint 4
    - Interconexión entre Mesa y Unity
    - Bug Hunting y casos de prueba

Código

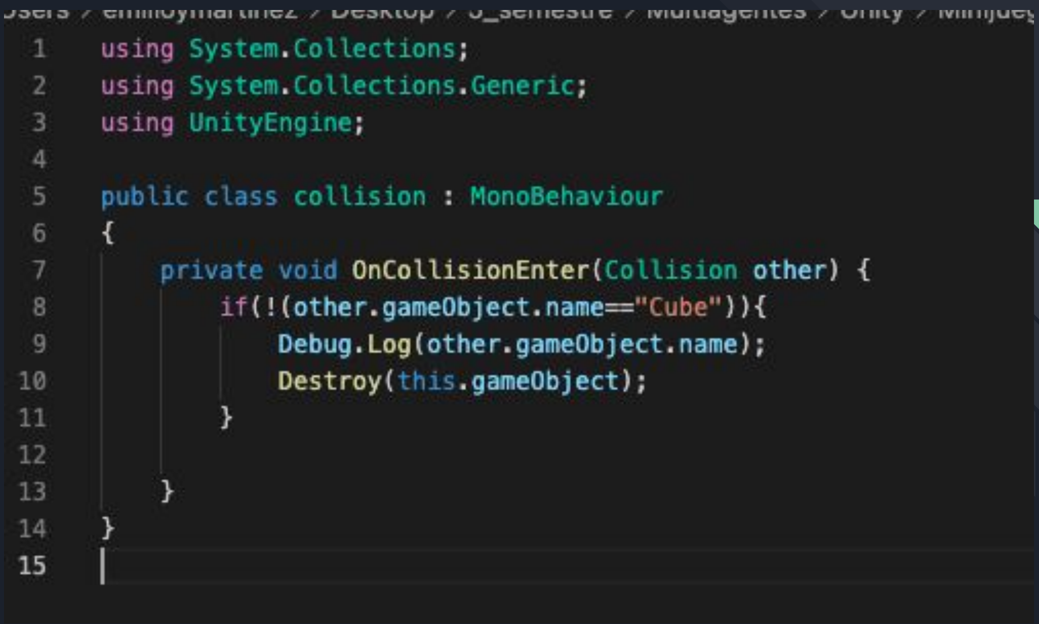
Unity



Users > emilioymartinez > Desktop > 5_semestre > Multiagentes > Unity > Carros_Multiagentes2 > Assets > Sc

```
    // Start is called before the first frame update
    public float speed=6f ;

    GameObject  lighter;
    private int value  ;
    // [SerializeField]
    // public GameObject carPrefab = GameObject.Find("Player");
    public Vector3 startPosition  ;
    void Start()
    {
        lighter = GameObject.Find("Semaforo (1)");
        Lighter light_orders = lighter.GetComponent<Lighter>();
        value = (light_orders.stateTrafficLight);
        Debug.Log("Carro 1:"+value);
        StartCoroutine(change_state());


    }

    // Update is called once per frame
    void Update()
    {

        // Debug.Log(value);
        if(value == 1 || value == 0  ){
            var dir = new Vector3(0,0,-2);
            transform.Translate(dir * speed * Time.deltaTime);
            if (transform.position.z <=-30){
                startPosition =new Vector3(Random.Range(5f,20f),0.5f,30);
                transform.position = startPosition;
            }
            }
        else if(value == 2){
            var dir = new Vector3(0,0,-1);
            transform.Translate(dir * speed * Time.deltaTime);
            if (transform.position.z <=-30){
                startPosition =new Vector3(Random.Range(5f,20f),0.5f,30);
                transform.position = startPosition;
        }
        }
        else if (value == 3 ){
            var dir = new Vector3(0,0,0);
            transform.Translate(dir * speed * Time.deltaTime);
            if (transform.position.z <=-30){
                startPosition =new Vector3(Random.Range(5f,20f),0.5f,30);
                transform.position = startPosition;
        }
        }
        }
        private IEnumerator change_state()
        {
        yield return  new WaitForSeconds(5);

        Lighter light_orders = lighter.GetComponent<Lighter>();
        value = (light_orders.stateTrafficLight);
```

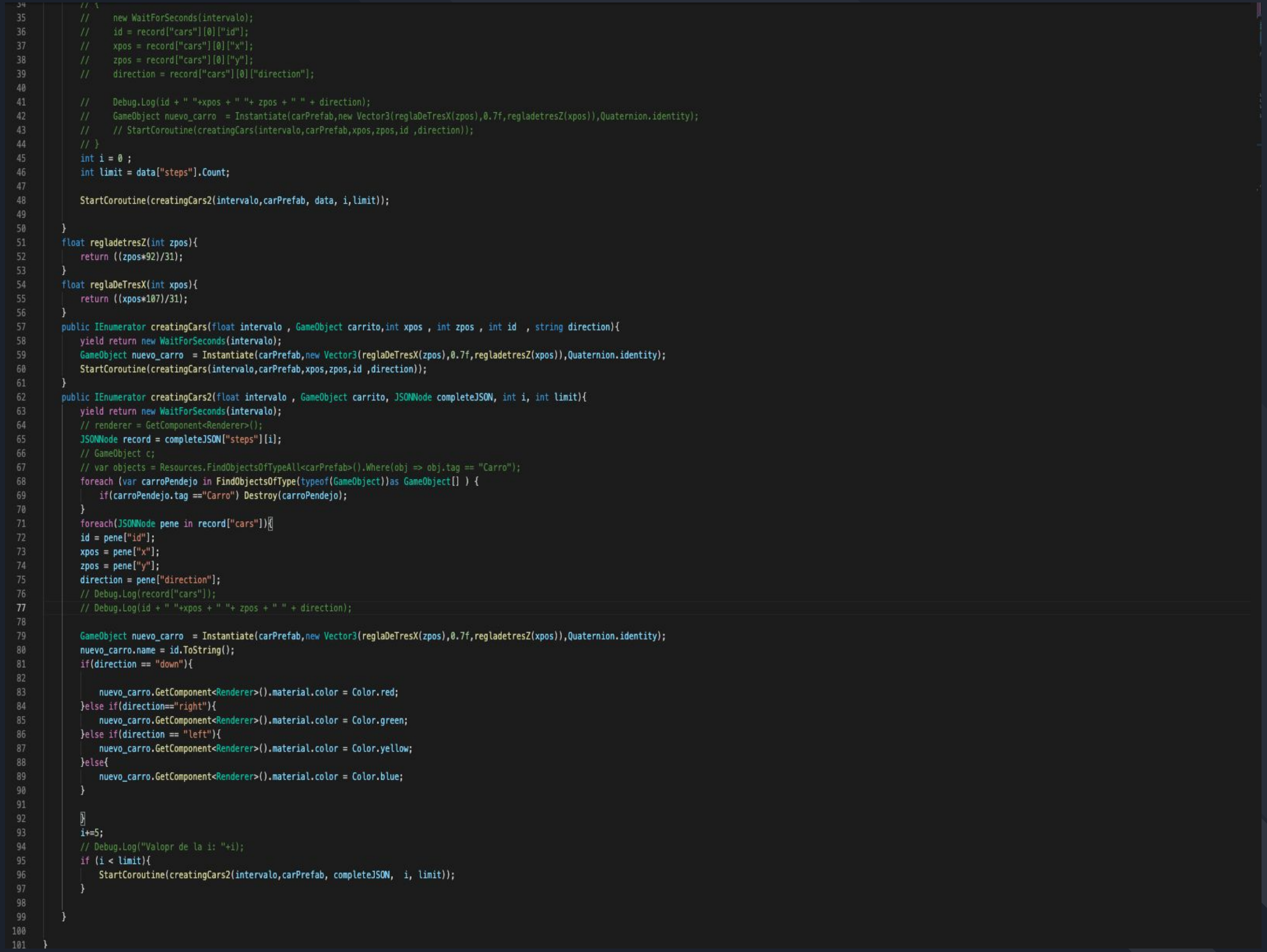
Collider



Users / emilioymartinez / Desktop / 5_semestre / Multiagentes / Unity / Minijueg

```csharp
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class collision : MonoBehaviour
{
    private void OnCollisionEnter(Collision other) {
        if(!(other.gameObject.name=="Cube")){
            Debug.Log(other.gameObject.name);
            Destroy(this.gameObject);
        }

    }
}
```

Json Reader

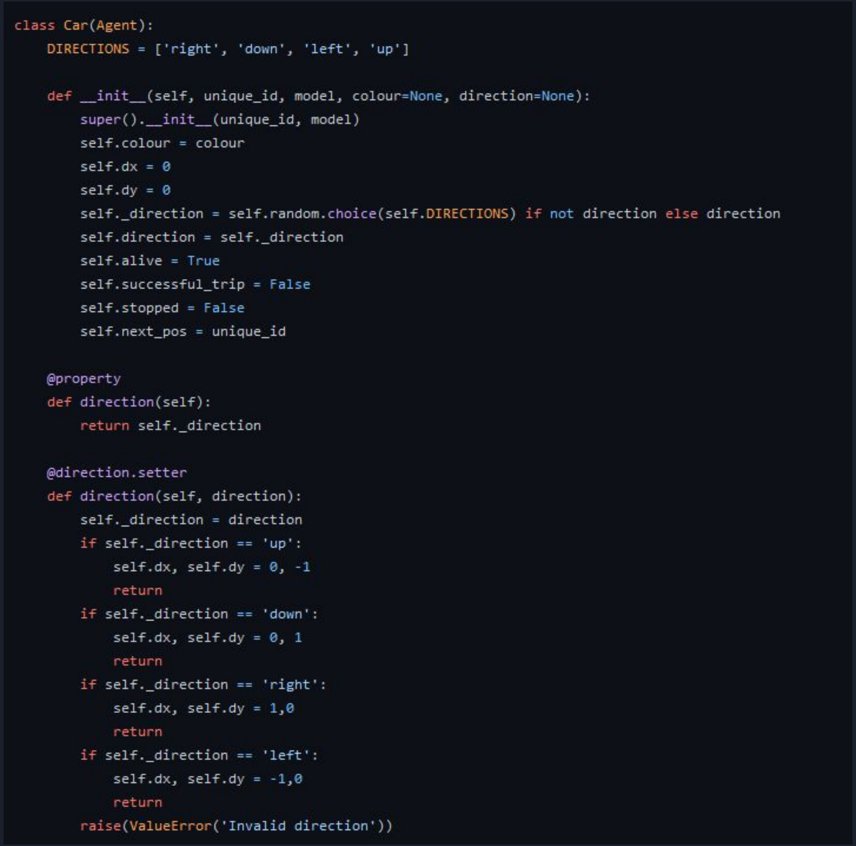


```
        // {
        //     new WaitForSeconds(intervalo);
        //     id = record["cars"][0]["id"];
        //     xpos = record["cars"][0]["x"];
        //     zpos = record["cars"][0]["y"];
        //     direction = record["cars"][0]["direction"];

        //     Debug.Log(id + " "+xpos + " "+ zpos + " " + direction);
        //     GameObject nuevo_carro  = Instantiate(carPrefab,new Vector3(reglaDeTresX(zpos),0.7f,regladetresZ(xpos)),Quaternion.identity);
        //     // StartCoroutine(creatingCars(intervalo,carPrefab,xpos,zpos,id ,direction));
        // }
        int i = 0 ;
        int limit = data["steps"].Count;

        StartCoroutine(creatingCars2(intervalo,carPrefab, data, i,limit));

    }
    float regladetresZ(int zpos){
        return ((zpos*92)/31);
    }
    float reglaDeTresX(int xpos){
        return ((xpos*107)/31);
    }
    public IEnumerator creatingCars(float intervalo , GameObject carrito,int xpos , int zpos , int id  , string direction){
        yield return new WaitForSeconds(intervalo);
        GameObject nuevo_carro  = Instantiate(carPrefab,new Vector3(reglaDeTresX(zpos),0.7f,regladetresZ(xpos)),Quaternion.identity);
        StartCoroutine(creatingCars(intervalo,carPrefab,xpos,zpos,id ,direction));
    }
    public IEnumerator creatingCars2(float intervalo , GameObject carrito, JSONNode completeJSON, int i, int limit){
        yield return new WaitForSeconds(intervalo);
        // renderer = GetComponent<Renderer>();
        JSONNode record = completeJSON["steps"][i];
        // GameObject c;
        // var objects = Resources.FindObjectsOfTypeAll<carPrefab>().Where(obj => obj.tag == "Carro");
        foreach (var carroPendejo in FindObjectsOfType(typeof(GameObject))as GameObject[] ) {
            if(carroPendejo.tag =="Carro") Destroy(carroPendejo);
        }
        foreach(JSONNode pene in record["cars"]){
        id = pene["id"];
        xpos = pene["x"];
        zpos = pene["y"];
        direction = pene["direction"];
        // Debug.Log(record["cars"]);
        // Debug.Log(id + " "+xpos + " "+ zpos + " " + direction);

        GameObject nuevo_carro  = Instantiate(carPrefab,new Vector3(reglaDeTresX(zpos),0.7f,regladetresZ(xpos)),Quaternion.identity);
        nuevo_carro.name = id.ToString();
        if(direction == "down"){

            nuevo_carro.GetComponent<Renderer>().material.color = Color.red;
        }else if(direction=="right"){
            nuevo_carro.GetComponent<Renderer>().material.color = Color.green;
        }else if(direction == "left"){
            nuevo_carro.GetComponent<Renderer>().material.color = Color.yellow;
        }else{
            nuevo_carro.GetComponent<Renderer>().material.color = Color.blue;
        }

        }
        i+=5;
        // Debug.Log("Valopr de la i: "+i);
        if (i < limit){
            StartCoroutine(creatingCars2(intervalo,carPrefab, completeJSON,  i, limit));
        }

    }

}
```

Python-Mesa

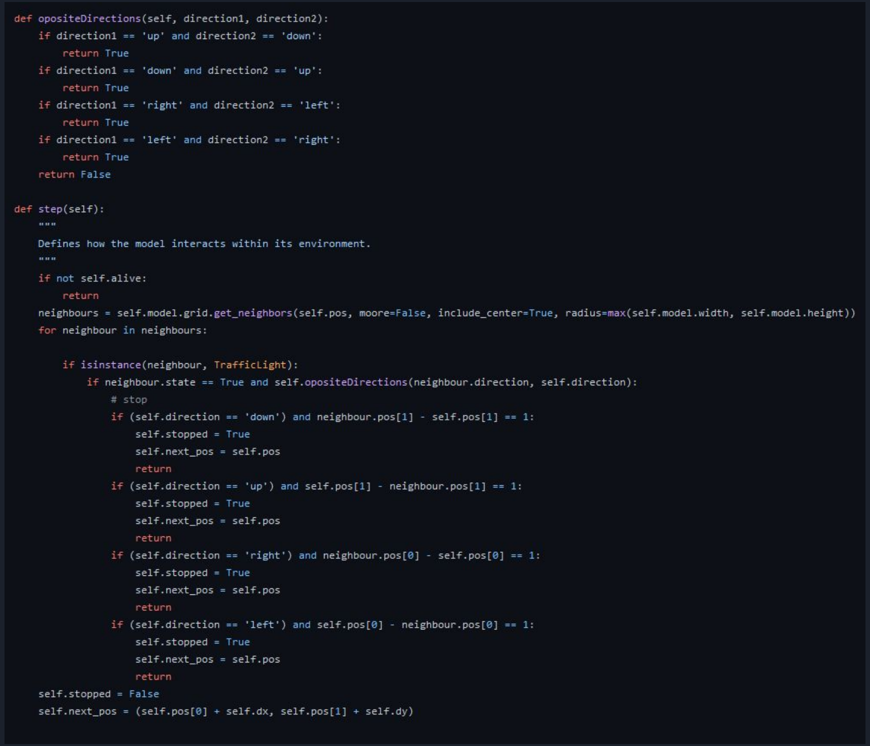
Agentes

Coche (declaracion)



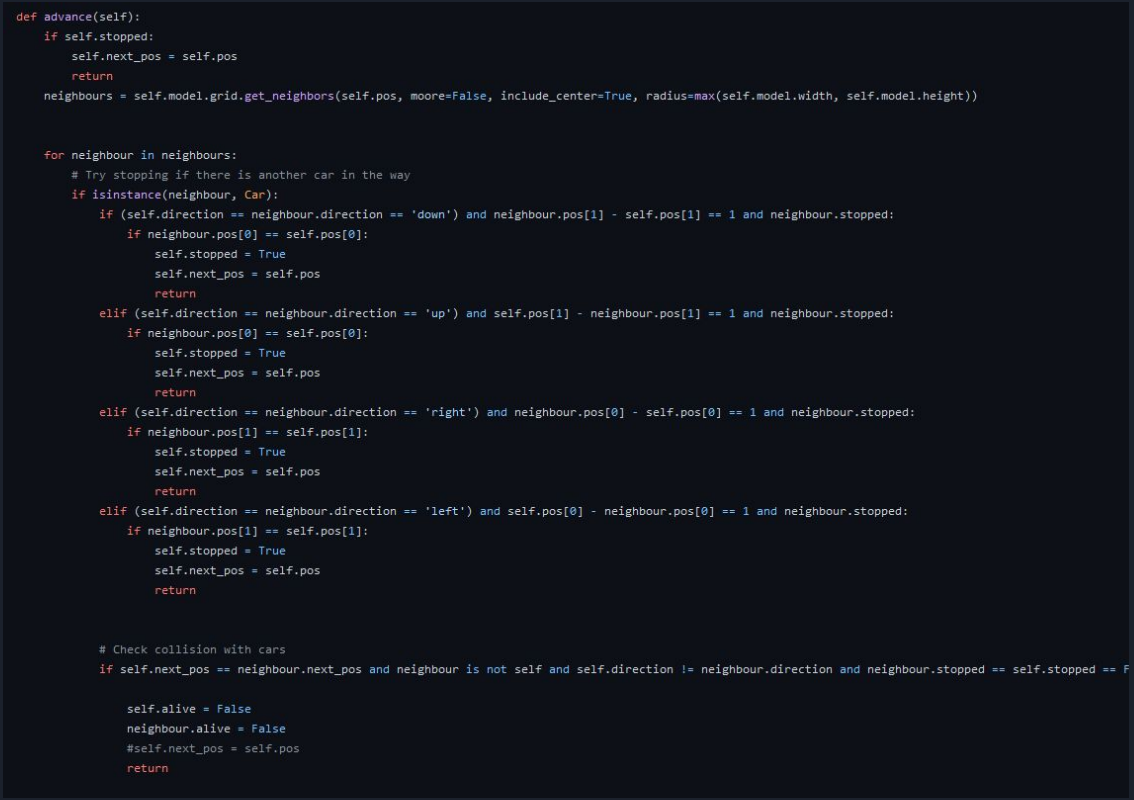
```python
class Car(Agent):
    DIRECTIONS = ['right', 'down', 'left', 'up']

    def __init__(self, unique_id, model, colour=None, direction=None):
        super().__init__(unique_id, model)
        self.colour = colour
        self.dx = 0
        self.dy = 0
        self._direction = self.random.choice(self.DIRECTIONS) if not direction else direction
        self.direction = self._direction
        self.alive = True
        self.successful_trip = False
        self.stopped = False
        self.next_pos = unique_id

    @property
    def direction(self):
        return self._direction

    @direction.setter
    def direction(self, direction):
        self._direction = direction
        if self._direction == 'up':
            self.dx, self.dy = 0, -1
            return
        if self._direction == 'down':
            self.dx, self.dy = 0, 1
            return
        if self._direction == 'right':
            self.dx, self.dy = 1,0
            return
        if self._direction == 'left':
            self.dx, self.dy = -1,0
            return
        raise(ValueError('Invalid direction'))
```

Coche (Movimiento Dirección)



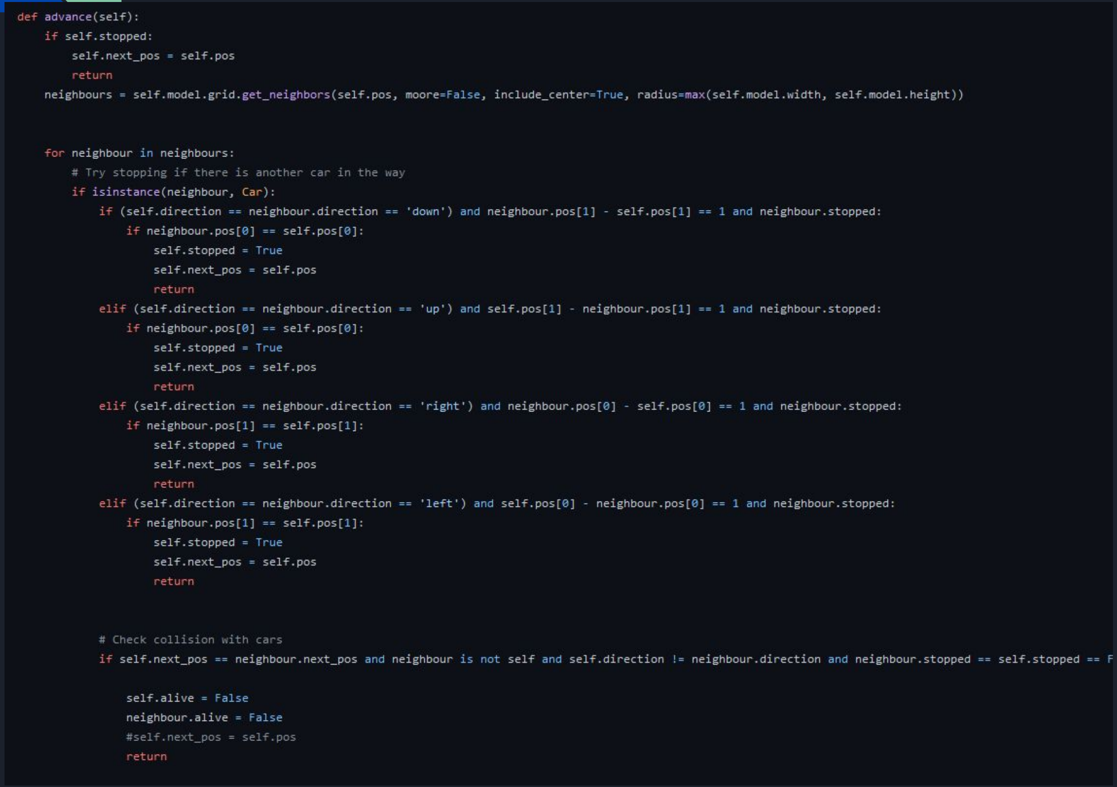
```python
def opositeDirections(self, direction1, direction2):
    if direction1 == 'up' and direction2 == 'down':
        return True
    if direction1 == 'down' and direction2 == 'up':
        return True
    if direction1 == 'right' and direction2 == 'left':
        return True
    if direction1 == 'left' and direction2 == 'right':
        return True
    return False

def step(self):
    """
    Defines how the model interacts within its environment.
    """
    if not self.alive:
        return
    neighbours = self.model.grid.get_neighbors(self.pos, moore=False, include_center=True, radius=max(self.model.width, self.model.height))
    for neighbour in neighbours:

        if isinstance(neighbour, TrafficLight):
            if neighbour.state == True and self.opositeDirections(neighbour.direction, self.direction):
                # stop
                if (self.direction == 'down') and neighbour.pos[1] - self.pos[1] == 1:
                    self.stopped = True
                    self.next_pos = self.pos
                    return
                if (self.direction == 'up') and self.pos[1] - neighbour.pos[1] == 1:
                    self.stopped = True
                    self.next_pos = self.pos
                    return
                if (self.direction == 'right') and neighbour.pos[0] - self.pos[0] == 1:
                    self.stopped = True
                    self.next_pos = self.pos
                    return
                if (self.direction == 'left') and self.pos[0] - neighbour.pos[0] == 1:
                    self.stopped = True
                    self.next_pos = self.pos
                    return
    self.stopped = False
    self.next_pos = (self.pos[0] + self.dx, self.pos[1] + self.dy)
```

Coche (Movimiento)

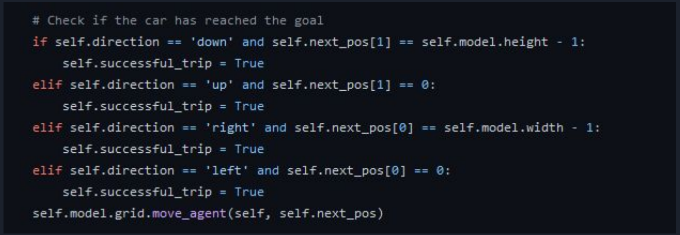


```python
def advance(self):
    if self.stopped:
        self.next_pos = self.pos
        return
    neighbours = self.model.grid.get_neighbors(self.pos, moore=False, include_center=True, radius=max(self.model.width, self.model.height))


    for neighbour in neighbours:
        # Try stopping if there is another car in the way
        if isinstance(neighbour, Car):
            if (self.direction == neighbour.direction == 'down') and neighbour.pos[1] - self.pos[1] == 1 and neighbour.stopped:
                if neighbour.pos[0] == self.pos[0]:
                    self.stopped = True
                    self.next_pos = self.pos
                    return
            elif (self.direction == neighbour.direction == 'up') and self.pos[1] - neighbour.pos[1] == 1 and neighbour.stopped:
                if neighbour.pos[0] == self.pos[0]:
                    self.stopped = True
                    self.next_pos = self.pos
                    return
            elif (self.direction == neighbour.direction == 'right') and neighbour.pos[0] - self.pos[0] == 1 and neighbour.stopped:
                if neighbour.pos[1] == self.pos[1]:
                    self.stopped = True
                    self.next_pos = self.pos
                    return
            elif (self.direction == neighbour.direction == 'left') and self.pos[0] - neighbour.pos[0] == 1 and neighbour.stopped:
                if neighbour.pos[1] == self.pos[1]:
                    self.stopped = True
                    self.next_pos = self.pos
                    return


            # Check collision with cars
            if self.next_pos == neighbour.next_pos and neighbour is not self and self.direction != neighbour.direction and neighbour.stopped == self.stopped == F

                self.alive = False
                neighbour.alive = False
                #self.next_pos = self.pos
                return
```

Coche (Movimiento)

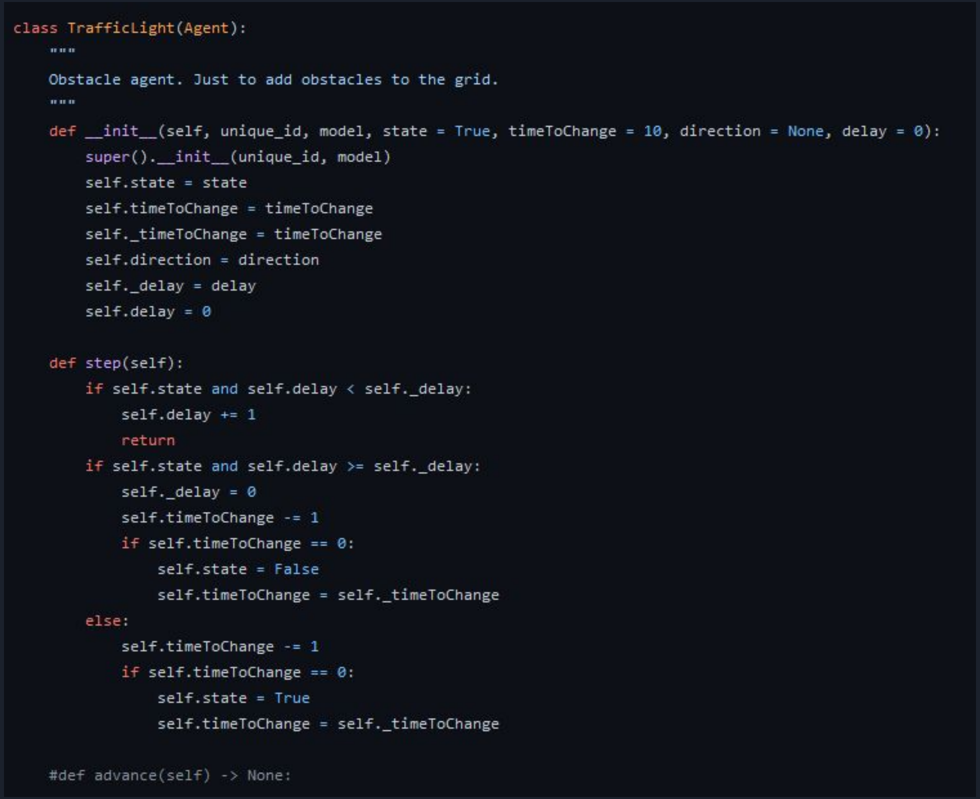


```python
def advance(self):
    if self.stopped:
        self.next_pos = self.pos
        return
    neighbours = self.model.grid.get_neighbors(self.pos, moore=False, include_center=True, radius=max(self.model.width, self.model.height))


    for neighbour in neighbours:
        # Try stopping if there is another car in the way
        if isinstance(neighbour, Car):
            if (self.direction == neighbour.direction == 'down') and neighbour.pos[1] - self.pos[1] == 1 and neighbour.stopped:
                if neighbour.pos[0] == self.pos[0]:
                    self.stopped = True
                    self.next_pos = self.pos
                    return
            elif (self.direction == neighbour.direction == 'up') and self.pos[1] - neighbour.pos[1] == 1 and neighbour.stopped:
                if neighbour.pos[0] == self.pos[0]:
                    self.stopped = True
                    self.next_pos = self.pos
                    return
            elif (self.direction == neighbour.direction == 'right') and neighbour.pos[0] - self.pos[0] == 1 and neighbour.stopped:
                if neighbour.pos[1] == self.pos[1]:
                    self.stopped = True
                    self.next_pos = self.pos
                    return
            elif (self.direction == neighbour.direction == 'left') and self.pos[0] - neighbour.pos[0] == 1 and neighbour.stopped:
                if neighbour.pos[1] == self.pos[1]:
                    self.stopped = True
                    self.next_pos = self.pos
                    return


            # Check collision with cars
            if self.next_pos == neighbour.next_pos and neighbour is not self and self.direction != neighbour.direction and neighbour.stopped == self.stopped == F

                self.alive = False
                neighbour.alive = False
                #self.next_pos = self.pos
                return
```



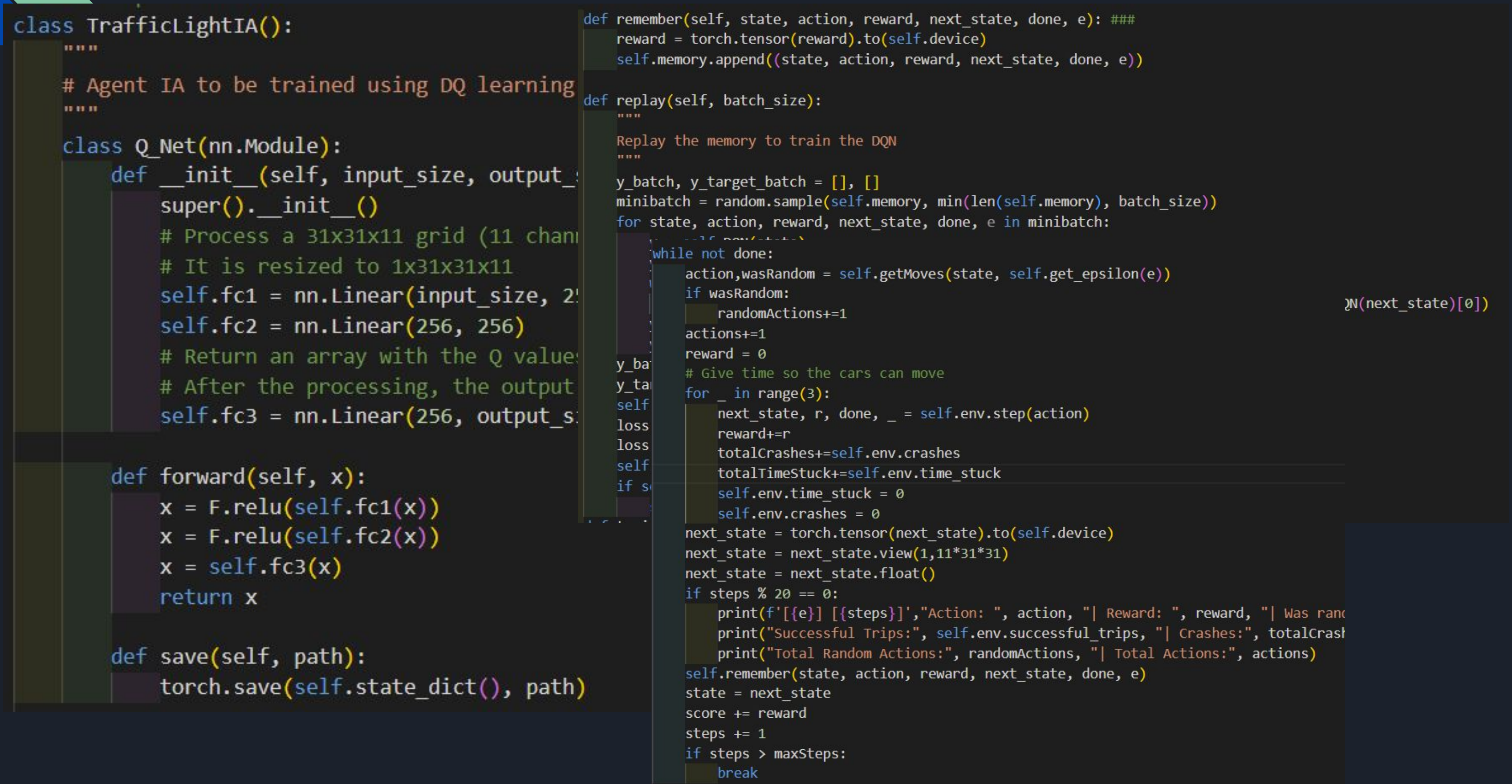
```python
# Check if the car has reached the goal
if self.direction == 'down' and self.next_pos[1] == self.model.height - 1:
    self.successful_trip = True
elif self.direction == 'up' and self.next_pos[1] == 0:
    self.successful_trip = True
elif self.direction == 'right' and self.next_pos[0] == self.model.width - 1:
    self.successful_trip = True
elif self.direction == 'left' and self.next_pos[0] == 0:
    self.successful_trip = True
self.model.grid.move_agent(self, self.next_pos)
```

Luz de tráfico (IA Algorítmica)



```python
class TrafficLight(Agent):
    """
    Obstacle agent. Just to add obstacles to the grid.
    """
    def __init__(self, unique_id, model, state = True, timeToChange = 10, direction = None, delay = 0):
        super().__init__(unique_id, model)
        self.state = state
        self.timeToChange = timeToChange
        self._timeToChange = timeToChange
        self.direction = direction
        self._delay = delay
        self.delay = 0

    def step(self):
        if self.state and self.delay < self._delay:
            self.delay += 1
            return
        if self.state and self.delay >= self._delay:
            self._delay = 0
            self.timeToChange -= 1
            if self.timeToChange == 0:
                self.state = False
                self.timeToChange = self._timeToChange
        else:
            self.timeToChange -= 1
            if self.timeToChange == 0:
                self.state = True
                self.timeToChange = self._timeToChange

    #def advance(self) -> None:
```

Luz de tráfico (DQ Learning)



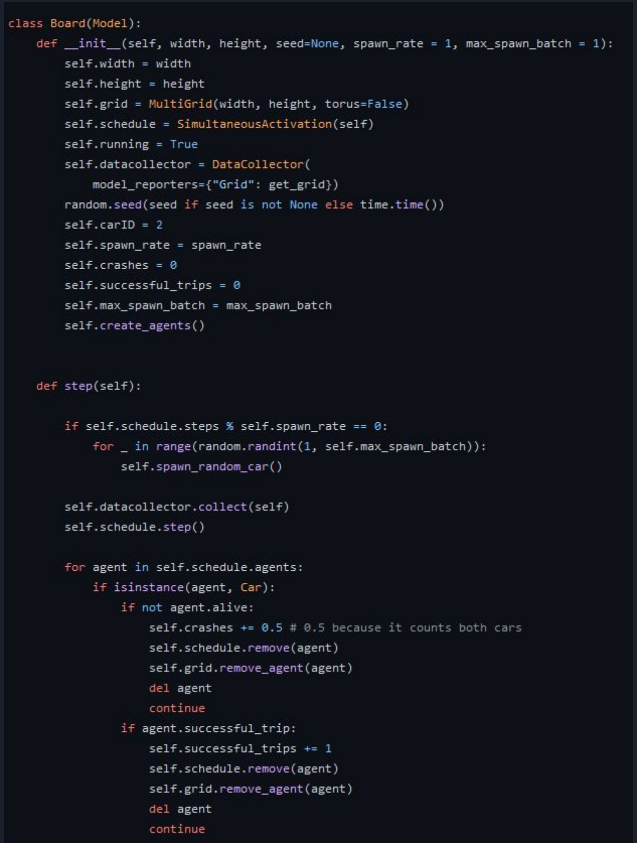
```
class TrafficLightIA():
    """
    # Agent IA to be trained using DQ learning
    """
    class Q_Net(nn.Module):
        def __init__(self, input_size, output_s
            super().__init__()
            # Process a 31x31x11 grid (11 chann
            # It is resized to 1x31x31x11
            self.fc1 = nn.Linear(input_size, 25
            self.fc2 = nn.Linear(256, 256)
            # Return an array with the Q values
            # After the processing, the output
            self.fc3 = nn.Linear(256, output_si

        def forward(self, x):
            x = F.relu(self.fc1(x))
            x = F.relu(self.fc2(x))
            x = self.fc3(x)
            return x

        def save(self, path):
            torch.save(self.state_dict(), path)
```

```
def remember(self, state, action, reward, next_state, done, e): ###
    reward = torch.tensor(reward).to(self.device)
    self.memory.append((state, action, reward, next_state, done, e))

def replay(self, batch_size):
    """
    Replay the memory to train the DQN
    """
    y_batch, y_target_batch = [], []
    minibatch = random.sample(self.memory, min(len(self.memory), batch_size))
    for state, action, reward, next_state, done, e in minibatch:
                                                                QN(next_state)[0])
    y_ba
    y_ta
    self
    loss
    loss
    self
    if s
```

```
while not done:
    action,wasRandom = self.getMoves(state, self.get_epsilon(e))
    if wasRandom:
        randomActions+=1
    actions+=1
    reward = 0
    # Give time so the cars can move
    for _ in range(3):
        next_state, r, done, _ = self.env.step(action)
        reward+=r
        totalCrashes+=self.env.crashes
        totalTimeStuck+=self.env.time_stuck
        self.env.time_stuck = 0
        self.env.crashes = 0
    next_state = torch.tensor(next_state).to(self.device)
    next_state = next_state.view(1,11*31*31)
    next_state = next_state.float()
    if steps % 20 == 0:
        print(f'[{e}] [{steps}]',"Action: ", action, "| Reward: ", reward, "| Was rand
        print("Successful Trips:", self.env.successful_trips, "| Crashes:", totalCrash
        print("Total Random Actions:", randomActions, "| Total Actions:", actions)
    self.remember(state, action, reward, next_state, done, e)
    state = next_state
    score += reward
    steps += 1
    if steps > maxSteps:
        break
```

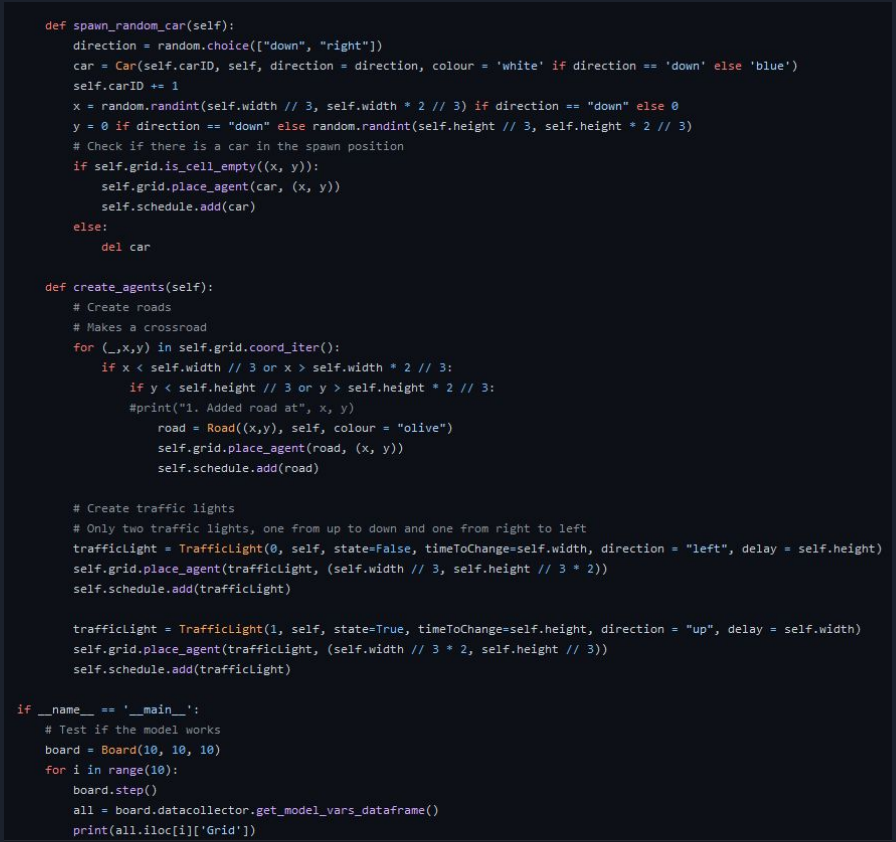
Models

Board (Inicialización y step)



```
class Board(Model):
    def __init__(self, width, height, seed=None, spawn_rate = 1, max_spawn_batch = 1):
        self.width = width
        self.height = height
        self.grid = MultiGrid(width, height, torus=False)
        self.schedule = SimultaneousActivation(self)
        self.running = True
        self.datacollector = DataCollector(
            model_reporters={"Grid": get_grid})
        random.seed(seed if seed is not None else time.time())
        self.carID = 2
        self.spawn_rate = spawn_rate
        self.crashes = 0
        self.successful_trips = 0
        self.max_spawn_batch = max_spawn_batch
        self.create_agents()


    def step(self):

        if self.schedule.steps % self.spawn_rate == 0:
            for _ in range(random.randint(1, self.max_spawn_batch)):
                self.spawn_random_car()

        self.datacollector.collect(self)
        self.schedule.step()

        for agent in self.schedule.agents:
            if isinstance(agent, Car):
                if not agent.alive:
                    self.crashes += 0.5 # 0.5 because it counts both cars
                    self.schedule.remove(agent)
                    self.grid.remove_agent(agent)
                    del agent
                    continue
                if agent.successful_trip:
                    self.successful_trips += 1
                    self.schedule.remove(agent)
                    self.grid.remove_agent(agent)
                    del agent
                    continue
```

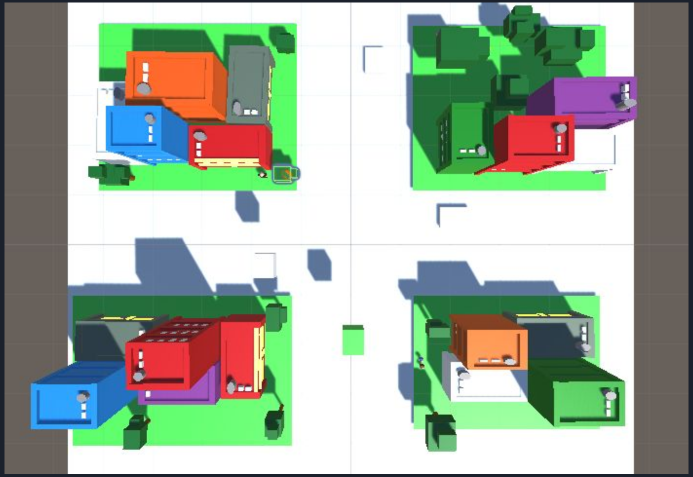
Board (spawner)



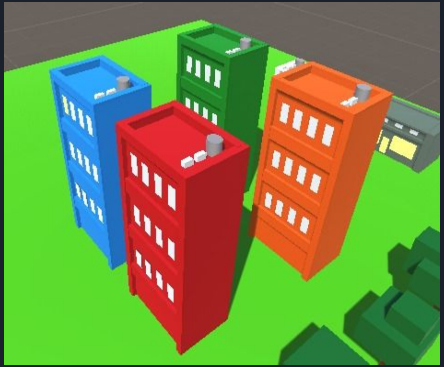
```
    def spawn_random_car(self):
        direction = random.choice(["down", "right"])
        car = Car(self.carID, self, direction = direction, colour = 'white' if direction == 'down' else 'blue')
        self.carID += 1
        x = random.randint(self.width // 3, self.width * 2 // 3) if direction == "down" else 0
        y = 0 if direction == "down" else random.randint(self.height // 3, self.height * 2 // 3)
        # Check if there is a car in the spawn position
        if self.grid.is_cell_empty((x, y)):
            self.grid.place_agent(car, (x, y))
            self.schedule.add(car)
        else:
            del car

    def create_agents(self):
        # Create roads
        # Makes a crossroad
        for (_,x,y) in self.grid.coord_iter():
            if x < self.width // 3 or x > self.width * 2 // 3:
                if y < self.height // 3 or y > self.height * 2 // 3:
                #print("1. Added road at", x, y)
                    road = Road((x,y), self, colour = "olive")
                    self.grid.place_agent(road, (x, y))
                    self.schedule.add(road)

        # Create traffic lights
        # Only two traffic lights, one from up to down and one from right to left
        trafficLight = TrafficLight(0, self, state=False, timeToChange=self.width, direction = "left", delay = self.height)
        self.grid.place_agent(trafficLight, (self.width // 3, self.height // 3 * 2))
        self.schedule.add(trafficLight)

        trafficLight = TrafficLight(1, self, state=True, timeToChange=self.height, direction = "up", delay = self.width)
        self.grid.place_agent(trafficLight, (self.width // 3 * 2, self.height // 3))
        self.schedule.add(trafficLight)

if __name__ == '__main__':
    # Test if the model works
    board = Board(10, 10, 10)
    for i in range(10):
        board.step()
        all = board.datacollector.get_model_vars_dataframe()
        print(all.iloc[i]['Grid'])
```

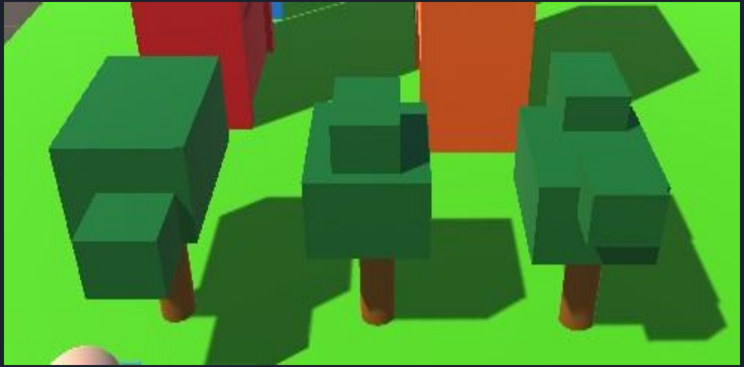
Resultados

Unity



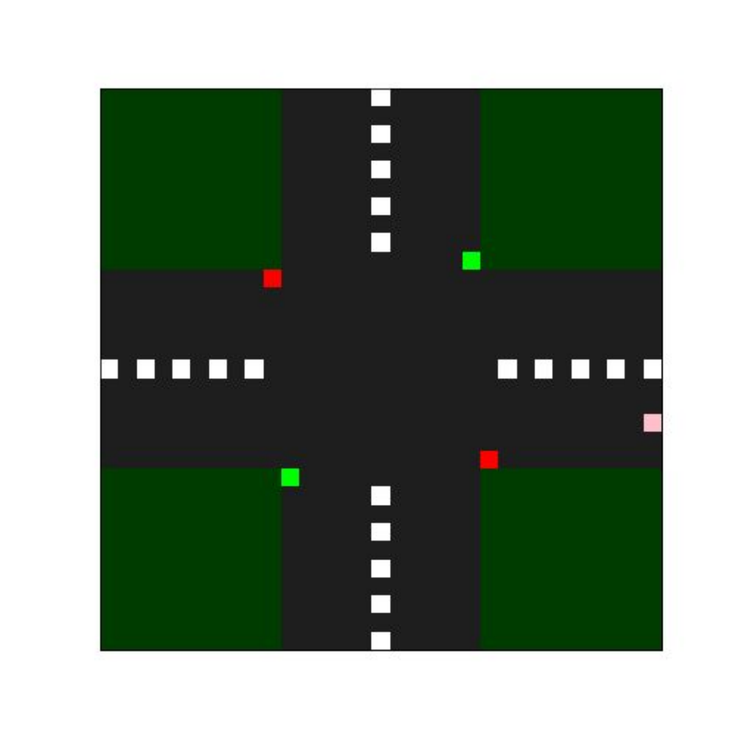
Entorno

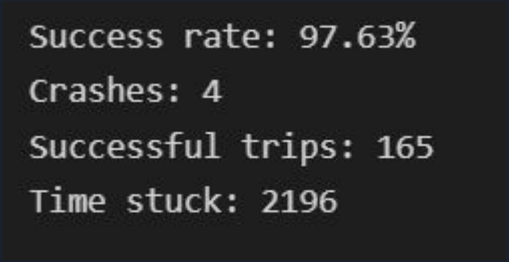




Análisis de resultados

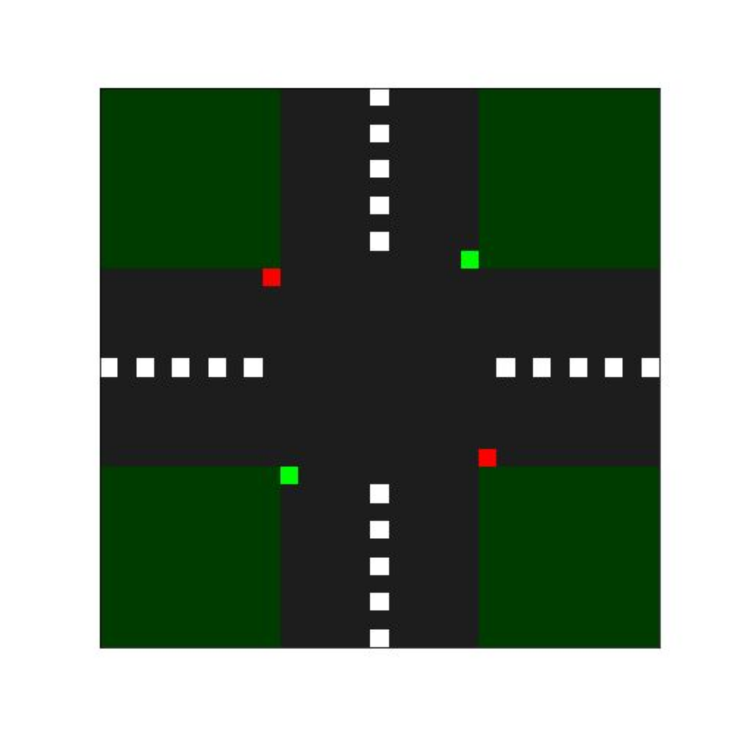
IA Algorítmica

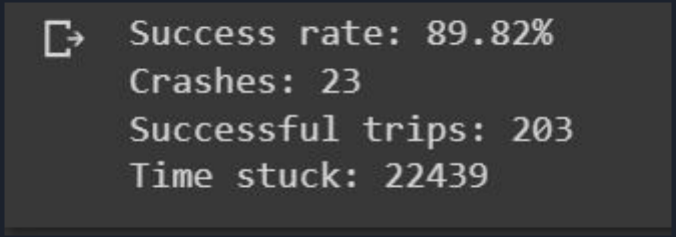




```
Success rate: 97.63%
Crashes: 4
Successful trips: 165
Time stuck: 2196
```

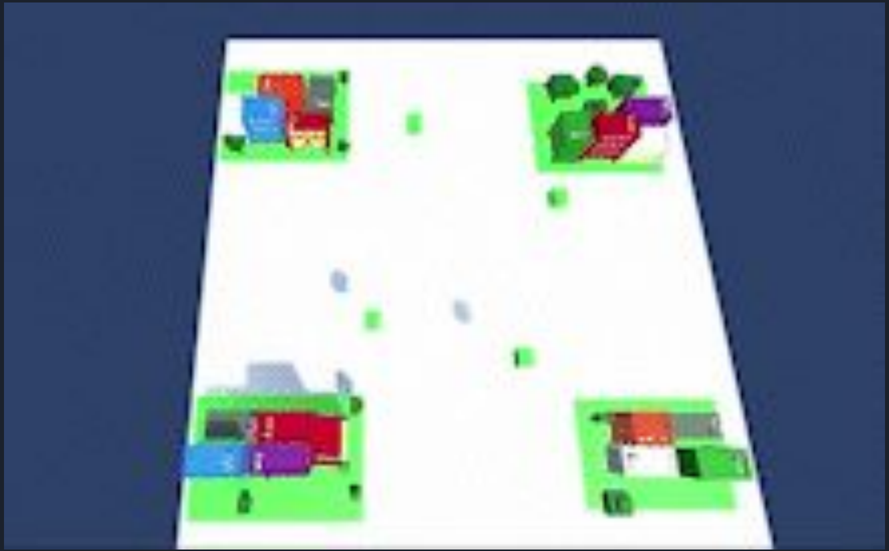
DQ Learning





```
Success rate: 89.82%
Crashes: 23
Successful trips: 203
Time stuck: 22439
```

Resultados



Discusión

Discusión

En la simulación se puede visualizar un intento de entrenamiento mediante IA para poder agilizar el tráfico al tratar de disminuir el tráfico mediante lecturas del entorno y la interconexión de los semáforos.

El manejo de multiagentes ha sido un proceso fundamental ya no solo en el estudio de inteligencia artificial, sino también en el avance de simulaciones avanzadas y cómo tratan de llegar a la meta propuestas, así como ya lo hacen los sistemas de multiagentes en el trading online, espionaje de objetivos y en la modelación de estructuras. Con esta simulación creemos que se demuestra este entendimiento por nuestra parte.

Conclusión

Tenemos la expectativa de que la aplicación que hemos desarrollado durante este bloque pueda ayudar a un mejor entendimiento del proceso en la simulación de multiagentes en torno a mejorar la circulación vial en calles y avenidas muy transitadas. Además consideramos que, de ser posible que se continúe este proyecto, este programa puede volverse cada vez más avanzado y permitirle la entrada de datos mediante aplicaciones de terceros y de esta manera hacer análisis con datos más sofisticados, esto con la finalidad de poder hacer simulaciones aplicables a la vida real con calles reales.