



Tecnológico de Monterrey

TC2008B

Modelación de sistemas multiagentes con gráficas computacionales

M5. Revisión de avance 3

Equipo 1

Julia Maria Stephanie Duenkelsbuehler Castillo - A01784399

Miguel Angel Cabrera Victoria - A01782982

Elit Shadday Acosta Pastrana - A00834393

7 de febrero de 2025

1. INTRODUCCIÓN

En el panorama actual, las urbes se encuentran con desafíos significativos en términos de movilidad y seguridad vial, particularmente en contextos donde vehículos y peatones se encuentran en contacto constante. El propósito de esta simulación es modelar estas interacciones de forma realista para comprender y examinar las dinámicas del tráfico urbano, incluyendo la administración de semáforos, el comportamiento de los peatones y la existencia de barreras que obstaculizan el tránsito vehicular y peatonal.

Este proyecto tiene como objetivo no solo perfeccionar nuestro entendimiento del diseño urbano, sino también funcionar como un instrumento educativo y de estudio para crear soluciones innovadoras en favor de la seguridad y eficacia de las ciudades.

2. CRÉDITOS

Julia Maria Stephanie Duenkelsbuehler Castillo - A01784399: Diseñador y Documentación

Miguel Angel Cabrera Victoria - A01782982: Programador

Elit Shadday Acosta Pastrana - A00834393: Especialista en gráficos

3. CONTEXTO Y PROBLEMA

La simulación busca representar y analizar el comportamiento de tráfico vehicular y peatonal en un entorno urbano controlado, incorporando semáforos, cruces peatonales y diferentes tipos de obstáculos. El objetivo es modelar interacciones realistas entre agentes (vehículos, peatones, semáforos) y el entorno para mejorar la gestión del tráfico y reducir posibles conflictos.

El problema central a abordar incluye:

- Control de flujos vehiculares y peatonales en intersecciones con semáforos.
- Gestión segura de cruces peatonales.
- Interacción de agentes con obstáculos como banquetas o peatones.

Se investigaron propuestas relacionadas con la optimización de tiempos de semáforos y simulaciones urbanas, como las realizadas en sistemas de transporte inteligente, donde se emplean enfoques multiagentes para modelar comportamiento realista.

4. OBJETIVOS GENERALES

1. Elaborar una simulación multiagente que ilustre el tráfico en una intersección urbana, teniendo en cuenta las normas básicas de tránsito y la interacción con semáforos.
2. Examinar el tráfico de vehículos para detectar potenciales atascos y mejorar los tiempos de espera en semáforos y movimientos en las vías principales.

3. Analizar la efectividad de la simulación al calcular los tiempos medios de espera y la optimización del flujo de vehículos.

5. RESTRICCIONES

La configuración del sistema presenta las siguientes limitaciones:

- Direcciones y calles: Las vías son de una sola o dos direcciones, con un sentido establecido previamente.
- Duraciones de los semáforos: Los ciclos de semáforos pueden extenderse entre 10 y 60 segundos, en función del flujo.
- Cruces para peatones: Solo activados cuando el semáforo para peatones muestra verde.
- Conexión: Los elementos deben responder ante sucesos externos, como interrupciones temporales.

6. HISTORIAS DE USUARIO

- Como conductor, deseo que mi coche respete los semáforos y se detenga frente a peatones para prevenir accidentes.
- Como peatón, deseo transitar de forma segura por las vías peatonales cuando el semáforo para peatones se encuentra en verde.
- Como gerente del sistema, deseo visualizar una ilustración del tráfico para examinar áreas de congestión.
- Como programador, debe garantizar que los agentes reaccionen correctamente a las normas del ambiente y a los impedimentos.

7. DESCRIPCIÓN DEL SISTEMA MULTIAGENTE

El sistema multiagente consta de tres tipos principales de agentes:

- Vehículos: Buscan sus destinos siguiendo las reglas de tráfico y respondiendo a semáforos y obstáculos.
- Peatones: Usan cruces peatonales para cruzar calles y evitan interacciones con vehículos estacionados.

- Semáforos: Controlan el flujo vehicular y peatonal, alternando entre estados rojo, amarillo y verde.

Interacciones Generales:

- Los vehículos interactúan con semáforos y peatones, deteniéndose cuando es necesario.
- Los peatones actúan con negligencia ya que no respetan las señales de tráfico pero sí evitan obstáculos.
- Los semáforos regulan el flujo de los carros.

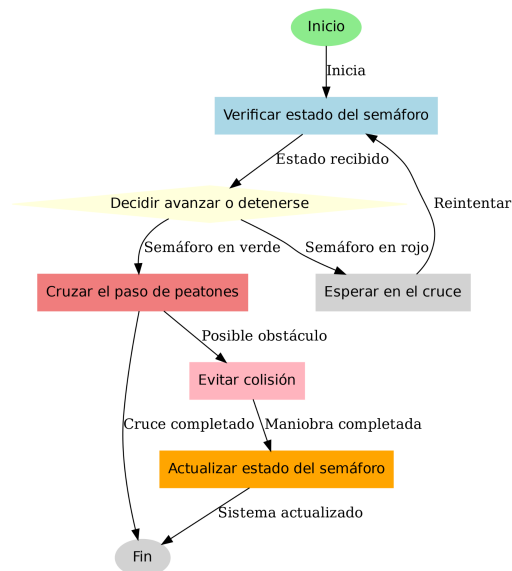


Diagrama de actividades

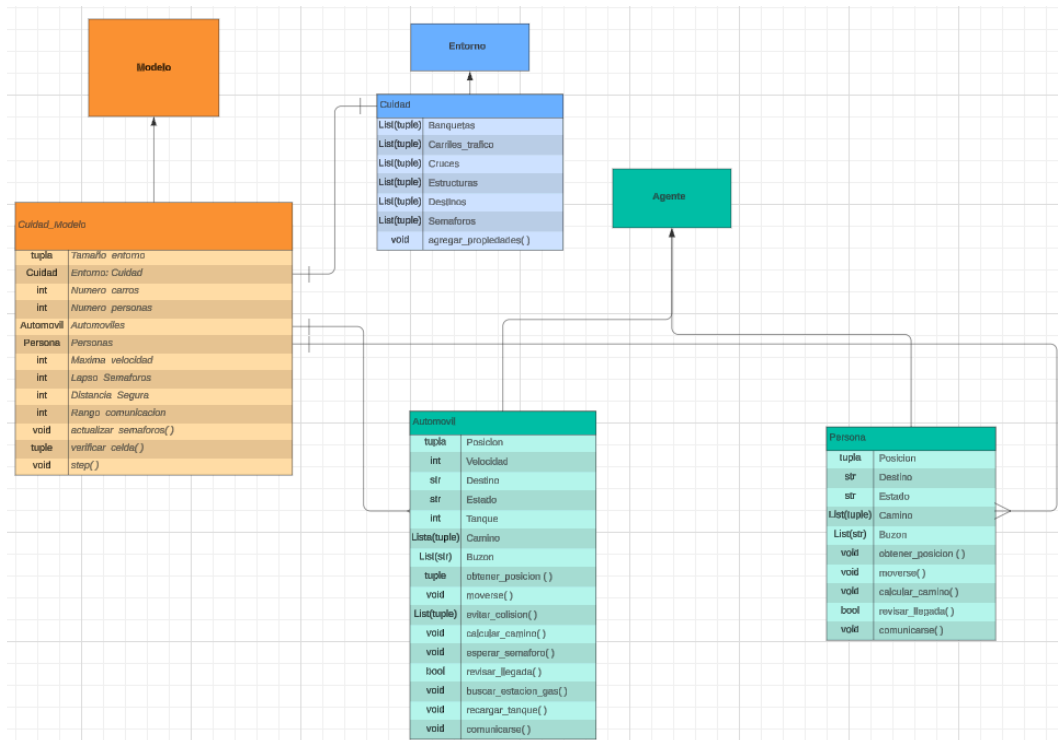


Diagrama de Clase

7.1 MODELO DE LOS AGENTES

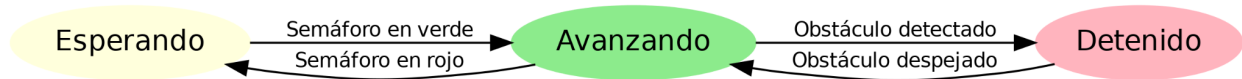
Cada agente tiene las siguientes características:

- **Vehículos:**
 - Creencias: Estado de los semáforos y posición de otros agentes.
 - Planes: Llegar al destino evitando colisiones.
 - Cooperación: Ceden el paso en cruces peatonales.
 - Aprendizaje: Opcional (mejorar rutas según congestionamiento).
- **Peatones:**
 - Creencias: Estado del semáforo peatonal.
 - Planes: Cruzar la calle cuando sea seguro.
 - Cooperación: Evitar obstruir vehículos.
 - Aprendizaje: N/A.
- **Semáforos:**
 - Creencias: Ciclos predefinidos de cambio de estado.

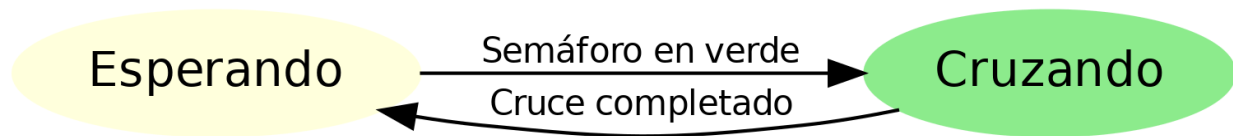
- Planes: Alternar entre estados para regular el tráfico.
- Cooperación: Sin interacción activa con otros agentes.

Diagramas de estados

Automóvil:

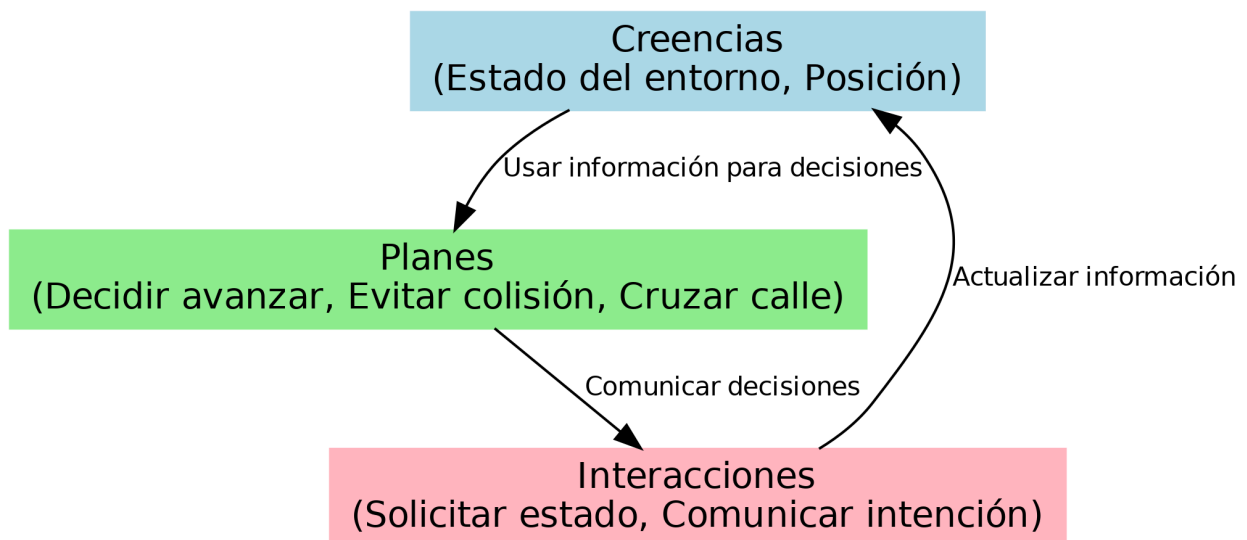


Peatón:



Representación gráfica del modelo del agente.

Creencias -> Información que el agente percibe del entorno, como el estado del semáforo o su posición. Planes -> Las acciones que el agente puede ejecutar como el empezar a avanzar, esquivar obstáculos o cruzar la calle. Interacciones -> Comunicación con otros agentes.



7.2 MODELO DEL ENTORNO

El ambiente a modelar es una ilustración simplificada, pero práctica, de un ambiente urbano en el que vehículos y peatones interactúan en un sistema de carreteras regulado por semáforos. Este panorama abarca banquetas, cruces, edificios y vehículos. Las siguientes son las propiedades del ambiente:

1. Observable o parcialmente observable

El medio ambiente es parcialmente perceptible. Los agentes (vehículos y peatones) solo pueden acceder a la información pertinente en su cercanía, como la ubicación de otros agentes próximos, la condición de los semáforos y las fronteras del entorno.

2. Determinista o estocástico

El sistema es estocástico, debido a que la conducta de los agentes varía en función de normas probabilísticas.

3. Episódico o secuencial

El ambiente es secuencial, dado que las elecciones de los actores se basan en su situación presente e influyen en sus estados futuros. Por ejemplo, un automóvil que espera en un semáforo de color rojo avanzará cuando este se transforme en verde, y su posición influirá en otros participantes.

4. Estático o dinámico

El ambiente es dinámico porque las condiciones varían de manera constante conforme los agentes se desplazan, interactúan y modifican la condición del sistema.

5. Discreto o continuo

El ambiente se configura de manera discreta en términos de gestión del tiempo y el espacio. El tiempo progresa en etapas establecidas, en tanto que el espacio se manifiesta como una malla o matriz que segmenta el espacio en celdas que se relacionan con calles, cruces y zonas concretas.

6. Agente simple o multiagente

El ambiente es multiagente, dado que diversas entidades autónomas (coches y peatones) se relacionan entre ellas de acuerdo a normas establecidas.

Se gestionará el tiempo en etapas discretas que representan intervalos de tiempo homogéneos. Cada transcurso del tiempo se refiere a una actualización del estado para todos los actores, lo que comprende modificaciones en la posición, decisiones de cruce o progreso y actualizaciones en los semáforos.

Se modela el espacio mediante una malla bidimensional (matriz de celdas), en la que cada celda simboliza una unidad del ambiente. Las celdas poseen características particulares que señalan si simbolizan una calle, un cruce para peatones, un impedimento o un espacio accesible. Este método posibilita una gestión granular de las ubicaciones de los elementos y promueve las interacciones entre estos y el ambiente.

7.3 MODELO DE LA NEGOCIACIÓN

Describe cómo van a interactuar los agentes, qué tipo de mensajes van a intercambiar y cómo. Se van a realizar subastas, votaciones, etc. Incluye un diagrama de comunicaciones.

En este sistema multiagente, las interacciones se basan en mensajes directos y reglas predefinidas sin necesidad de mecanismos complejos como subastas o votaciones. Esto se debe a que los roles de los agentes están claramente definidos:

- Semáforo:

Comunica su estado (verde, amarillo, rojo) a los automóviles y peatones.

- Automóvil:

Solicita el estado del semáforo y detecta obstáculos en su camino.

- Peatón:

Solicita permiso para cruzar y comunica su intención de avanzar.

Mensajes principales:

Automóvil → Semáforo: "Solicitar estado".

Semáforo → Automóvil/Peatón: "Estado del tráfico: Verde/Rojo".

Peatón → Automóvil: "Intención de cruzar".

El uso de reglas predefinidas asegura que los agentes puedan coordinarse eficientemente sin la complejidad adicional de negociaciones basadas en subastas o votaciones. Este enfoque es suficiente para garantizar un tránsito seguro y fluido.

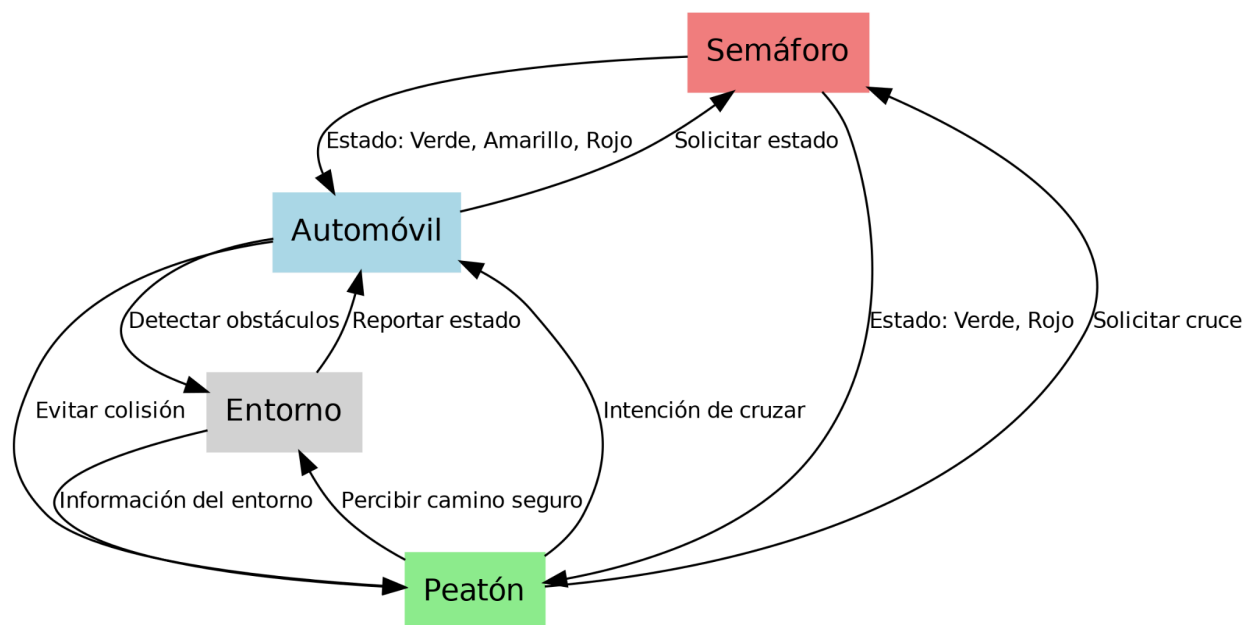


Diagrama de comunicación

7.4 MODELO DE LA INTERACCIÓN

El sistema multiagente interactúa con el sistema gráfico mediante la comunicación continua entre el modelo de simulación y la interfaz gráfica creada en Unity. Este procedimiento se lleva a cabo en dos fases fundamentales:

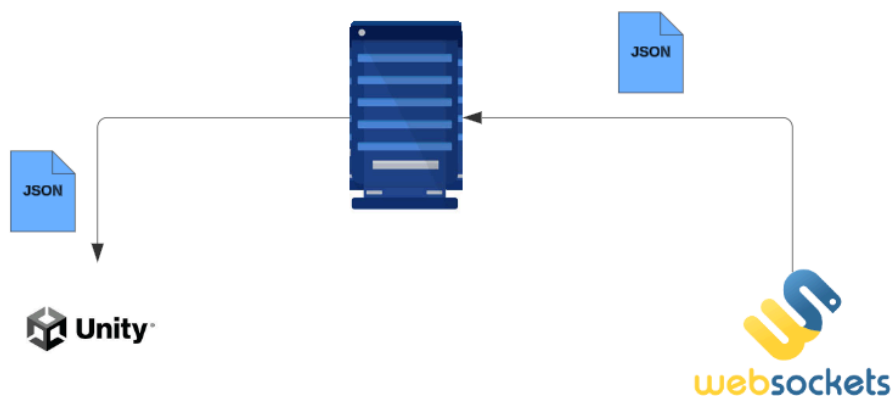
El sistema multiagente produce información actualizada acerca del estado de los agentes, tales como sus ubicaciones, velocidad, estados de los semáforos y rutas programadas. Esta información se transmite a Unity a través de un archivo JSON o de manera directa a través de una conexión en tiempo real, en función de la configuración establecida.

Unity procesa esta información y actualiza la ilustración visual del ambiente. Cada agente es presentado en su nueva ubicación, y las modificaciones en los semáforos o interacciones se muestran de forma gráfica en tiempo real.

Unity emplea los datos suministrados para:

- Ilustrar a los automóviles y peatones como objetos tridimensionales con animaciones fluidas.
- Actualizar los semáforos según su condición presente.
- Presentar rutas programadas o zonas ocupadas para el análisis visual.

La integración asegura que el sistema gráfico sea un fiel reflejo del modelo multiagente, posibilitando la observación en tiempo real del comportamiento de los agentes y la evaluación del efecto de las configuraciones.



8. DESCRIPCIÓN DEL MODELADO GRÁFICO



8.1 ESCENA A MODELAR





La escena modelada busca representar un sistema urbano simplificado, diseñado para simular un sistema multiagente donde peatones y semáforos interactúan de manera coordinada. La escena ya no está dividida en dos áreas por temas de simplicidad:

- Área Urbana: Contiene carreteras con líneas de tráfico, semáforos funcionales, cruces peatonales y edificios. Los agentes (vehículos y peatones) se moverán dentro de esta área, interactuando según las reglas definidas por los semáforos.



8.2 COMPONENTES GRÁFICOS

Nombre	Descripción	Imagen	Fuente
Edificio Urbano	Estructuras utilizadas para delimitar la zona urbana y dar contexto visual.		Diseñado por Julia
Carretera	Vías de circulación para vehículos, con carriles marcados y cruces peatonales.		Diseñado por Elit

Semáforo	Señal luminosa para controlar el flujo vehicular y peatonal.		Diseñado por Elit
Edificio Urbano	Estructuras utilizadas para delimitar la zona urbana y dar contexto visual.		Diseñado por Julia
Carretera	Vías de circulación para vehículos, con carriles marcados y cruces peatonales.		Diseñado por Elit
Semáforo	Señal luminosa para controlar el flujo vehicular y peatonal.		Diseñado por Elit

8.3 PREFABS

Semáforo

- Descripción Controla el flujo de los Vehículos y los peatones



- Imagen

- Script:

```
// TrafficLightController.cs
using System.Collections;
using UnityEngine;
using System.Linq;
```

```
public class TrafficLightController : MonoBehaviour
{
```

```
    public enum LightState { Green, Yellow, Red }
    public LightState currentState = LightState.Red;
```

```
    public float greenDuration = 5f;
```

```
    public float yellowDuration = 2f;
```

```
    public float redDuration = 5f;
```

```
    public float pedestrianCrossingDuration = 3f; // Tiempo para que los peatones crucen
```

```

[Header("Prioridad del Semaforo")]
[Tooltip("El semaforo con prioridad mas baja (0) sera el primero en ponerse en
verde")]
public int priority = 0; //Asignamos prioridad desde el inspector

public static TrafficLightController[] trafficLights;
private static bool isMaster = false;

public delegate void TrafficLightChange(LightState state);
public event TrafficLightChange OnTrafficLightChange;

void Awake()
{
    if (trafficLights == null)
    {
        trafficLights = FindObjectsOfType<TrafficLightController>();

        // Ordenar semaforos por la prioridad asignada en el inspector
        trafficLights = trafficLights.OrderBy(t => t.priority).ToArray();

        // Definir que el semaforo con la prioridad mas baja controle el ciclo
        if (trafficLights.Length > 0 && trafficLights[0] == this)
        {
            isMaster = true;
        }
    }
}

void Start()
{
    if (isMaster)
    {
        InitializeTrafficLights();
        StartCoroutine(TrafficLightCycle());
    }
}

void InitializeTrafficLights()
{
    // Al iniciar el semaforo con prioridad 0 esta en verde y los demás en rojo
    for (int i = 0; i < trafficLights.Length; i++)
    {
        if (trafficLights[i].priority == 0)
        {

```

```

        trafficLights[i].SetLightState(LightState.Green);
    }
    else
    {
        trafficLights[i].SetLightState(LightState.Red);
    }
}
}

IEnumerator TrafficLightCycle()
{
    int currentIndex = 0;
    while (true)
    {
        yield return new WaitForSeconds(greenDuration);

        // Cambiar el semaforo actual a amarillo
        trafficLights[currentIndex].SetLightState(LightState.Yellow);
        yield return new WaitForSeconds(yellowDuration);

        // Cambiar el semaforo actual a rojo
        trafficLights[currentIndex].SetLightState(LightState.Red);

        // Cruce de peatones
        yield return new WaitForSeconds(pedestrianCrossingDuration);

        // Pasar al siguiente semaforo en la lista en base a prioridad
        currentIndex = (currentIndex + 1) % trafficLights.Length;

        // Encender en verde el siguiente semaforo segun la prioridad
        trafficLights[currentIndex].SetLightState(LightState.Green);
    }
}

public void SetLightState(LightState state)
{
    currentState = state;
    OnTrafficLightChange?.Invoke(currentState);
}
}

```

Peatón

- Descripción Representa agentes que interactúan con semáforos



- Imagen

- Script:

```
//PedestrianController.cs  
using System.Collections;  
using UnityEngine;
```

```
public class PedestrianController : MonoBehaviour  
{  
    public float speed = 2f;  
    private bool canCross = false;  
    private bool atTrafficLight = false;  
    private bool carDetected = false;  
  
    void OnEnable()  
    {  
        if (TrafficLightController.trafficLights != null)  
        {  
            foreach (var trafficLight in TrafficLightController.trafficLights)  
            {  
                trafficLight.OnTrafficLightChange += HandleTrafficLightChange;  
            }  
        }  
    }  
  
    void OnDisable()  
    {  
        if (TrafficLightController.trafficLights != null)  
        {  
            foreach (var trafficLight in TrafficLightController.trafficLights)  
            {  
                trafficLight.OnTrafficLightChange -= HandleTrafficLightChange;  
            }  
        }  
    }  
  
    void Update()  
    {  
        if ((!atTrafficLight || (atTrafficLight && canCross)) && !IsCarInFront())  
        {  
            transform.Translate(Vector3.forward * speed * Time.deltaTime);  
        }  
    }  
}
```

```

private bool IsCarInFront()
{
    RaycastHit hit;
    float detectionDistance = 1.5f; // Distancia para detectar coches al frente

    if (Physics.Raycast(transform.position, transform.forward, out hit,
detectionDistance))
    {
        if (hit.collider.CompareTag("Car"))
        {
            return true;
        }
    }
    return false;
}

void HandleTrafficLightChange(TrafficLightController.LightState state)
{
    if (atTrafficLight)
    {
        canCross = (state == TrafficLightController.LightState.Red);
    }
}

private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("TrafficLight"))
    {
        atTrafficLight = true;
    }
    else if (other.CompareTag("Obstacle"))
    {
        StartCoroutine(AvoidObstacle());
    }
    else if (other.CompareTag("Car"))
    {
        carDetected = true;
    }
}

private void OnTriggerExit(Collider other)
{
    if (other.CompareTag("TrafficLight"))

```

```

    {
        atTrafficLight = false;
    }
    else if (other.CompareTag("Car"))
    {
        carDetected = false;
    }
}

private IEnumerator AvoidObstacle()
{
    float evadeDistance = 1.5f;
    Vector3 evadePosition = transform.position + new Vector3(1f * evadeDistance, 0,
0); // Movimiento a la derecha en caso de obstaculo

    if (!Physics.Raycast(transform.position, new Vector3(1, 0, 0), evadeDistance))
    {
        float elapsedTime = 0f;
        float duration = 0.5f;
        Vector3 startingPosition = transform.position;

        while (elapsedTime < duration)
        {
            transform.position = Vector3.Lerp(startingPosition, evadePosition,
elapsedTime / duration);
            elapsedTime += Time.deltaTime;
            yield return null;
        }

        transform.position = evadePosition;
    }
}
}

```

Vehículo

- Descripción El agente del móvil que sigue reglas de tráfico



- Imagen
- Script:
//CarController.cs
using UnityEngine;


```

public class CarController : MonoBehaviour
{
    public float speed = 5f;
    private bool canMove = true;
    private bool pedestrianDetected = false;
    private bool carDetectedAhead = false;
    private TrafficLightController assignedTrafficLight;

    void Start()
    {
        // Asignar el semaforo mas cercano al coche al iniciar
        assignedTrafficLight = FindClosestTrafficLight();
        if (assignedTrafficLight != null)
        {
            assignedTrafficLight.OnTrafficLightChange += HandleTrafficLightChange;
        }
    }

    void OnDisable()
    {
        if (assignedTrafficLight != null)
        {
            assignedTrafficLight.OnTrafficLightChange -= HandleTrafficLightChange;
        }
    }

    void Update()
    {
        if (canMove && !pedestrianDetected && !carDetectedAhead)
        {
            transform.Translate(Vector3.forward * speed * Time.deltaTime);
        }
    }

    void HandleTrafficLightChange(TrafficLightController.LightState state)
    {
        canMove = (state == TrafficLightController.LightState.Green);
    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Pedestrian"))
        {
            pedestrianDetected = true;
        }
    }
}

```

```

        canMove = false;
    }
    else if (other.CompareTag("Car"))
    {
        carDetectedAhead = true;
        canMove = false;
    }
}

private void OnTriggerExit(Collider other)
{
    if (other.CompareTag("Pedestrian"))
    {
        pedestrianDetected = false;
        canMove = true;
    }
    else if (other.CompareTag("Car"))
    {
        carDetectedAhead = false;
        canMove = true;
    }
}

// Metodo para encontrar el semaforo más cercano al coche
private TrafficLightController FindClosestTrafficLight()
{
    TrafficLightController[] allLights = FindObjectsOfType<TrafficLightController>();
    TrafficLightController closest = null;
    float minDistance = Mathf.Infinity;

    foreach (var light in allLights)
    {
        float distance = Vector3.Distance(transform.position, light.transform.position);
        if (distance < minDistance)
        {
            minDistance = distance;
            closest = light;
        }
    }
    return closest;
}
}

```

8.4 SCRIPTS

param_01.py

Descripción: Este script define un objeto de configuración (Parameter) que contiene un Value Object para la creación de los parámetros que el modelo necesitará para la simulación.

Interacciones:

- Utilizado por model.py para configurar el entorno de la ciudad y las propiedades de los agentes.

model.py

Descripción: Este script define la clase CityModel, que representa el modelo de simulación general. Hereda de ap.Model de la biblioteca agentpy.

- CityModel.setup(): Inicializa el tamaño del entorno del modelo, crea un objeto de entorno City, configura el rango de comunicación de los agentes, el número de automóviles y personas, y crea listas de agentes para automóviles y personas.
- CityModel.update_semaphores(): Esta función está actualmente vacía, pero está destinada a actualizar el estado de los semáforos en función de un lapso de tiempo.
- CityModel.update(): Esta función está actualmente vacía, pero está destinada a realizar cualquier actualización antes de cada paso de simulación.
- CityModel.step(): Ejecuta un paso de la simulación llamando al método execute de cada agente de automóvil y persona.

Interacciones:

- Crea un entorno City utilizando la configuración de param_01.py.
- Crea y gestiona agentes de automóviles y personas utilizando las clases Car y Person (probablemente de archivos separados).

city.py

Descripción: Este script define la clase City, que representa el entorno de la ciudad dentro del modelo de simulación. Hereda de ap.Grid de la biblioteca agentpy.

- `City.setup()`: Almacena los parámetros de configuración relacionados con el entorno de la ciudad (aceras, carriles, intersecciones, edificios, destinos y semáforos) de `param_01.py`.
- `City.add_sidewalks()`, `City.add_lanes()`, `City.add_intersections()`, `City.add_buildings()`, `City.add_destinities()`: Estas funciones agregan los elementos correspondientes (aceras, carriles, etc.) a la cuadrícula del entorno de la ciudad en función de la configuración.
- `City.add_semaphores()`: Esta función está actualmente vacía, pero está destinada a agregar semáforos al entorno.
- `City.add_properties()`: Llama a todas las funciones auxiliares para agregar aceras, carriles, intersecciones, edificios y destinos al entorno.

Interacciones:

- Utilizado por `model.py` para crear el entorno de la ciudad.

persons.py

Descripción: La clase `Person` se refiere a agentes peatonales que habrá en el entorno

- `Person.setup()`: Inicializa el agente persona con referencias al entorno, posición, destino, estado, ruta y buzón.
- `Person.get_position()`: Obtiene la posición actual del agente persona en el entorno.
- `Person.calculate_path()`: Esta función está actualmente vacía, pero está destinada a calcular la ruta óptima para que el agente persona llegue a su destino.
- `Person.in_destiny()`: Verifica si el agente persona ha llegado a su destino.
- `Person.execute()`: Esta función es actualmente solo un marcador de posición para las acciones de la persona durante cada paso de la simulación.

Interacciones:

- Destinado a ser utilizado por `model.py` para crear y gestionar agentes de personas dentro del entorno de la ciudad.

car.py

Descripción: La clase es `Car` para los agentes de automóviles. Probablemente, tendría una estructura similar a `person.py`, pero manejaría atributos y comportamientos específicos de los automóviles.

Interacciones:

- Interactuar con el entorno de la ciudad (moverse por los carriles, detenerse en las intersecciones).
- Interactuar con los semáforos (seguir las señales de tráfico).

9. ENTREGABLES DE ADMINISTRACIÓN DE PROYECTO

Para asegurar el triunfo del proyecto, el equipo ha segmentado las labores en tareas concretas que tratan cada uno de los objetivos establecidos. Este plan especifica las obligaciones personales, el plazo previsto para finalizar cada labor y las fechas en las que se llevarán a cabo. Además, se determinan las tareas por realizar y los esfuerzos requeridos para lograr los objetivos de la simulación.

Semana	Actividad	Descripción	Responsables	Fecha	Esfuerzo Estimado
Semana 1	Investigación Inicial	Revisar conceptos clave sobre movilidad urbana y simulación multiagente.	Todos los integrantes	Lunes a miércoles	6 horas
Semana 1	Herramientas	Explorar frameworks de Python (AgentPy, Mesa) y funcionalidades de Unity.	Todos los integrantes	Jueves	4 horas

Tabla 1

Semana	Actividad	Descripción	Responsables	Fecha	Esfuerzo Estimado
Semana 2	Definición del Modelo	Diseñar los agentes (vehículos, peatones, semáforos) y establecer sus reglas de interacción.	Julia y Elit	Lunes a Miércoles	8 horas
Semana 2	Diagramas de Clase	Crear diagramas de clase y protocolos de interacción para documentar las relaciones.	Julia	Jueves a Viernes	6 horas
Semana 2	Implementación del Modelo Base	Desarrollar el modelo inicial en Python y preparar Unity para la simulación visual.	Miguel	Toda la semana	10 horas
Semana 3	Simulación en Python	Ejecutar simulaciones iniciales en Python para validar los parámetros definidos.	Todos los integrantes	Lunes a Miércoles	6 horas
Semana 3	Visualización en Unity	Integrar los resultados de Python en Unity para representar el tráfico visualmente.	Miguel y Julia	Jueves a Viernes	8 horas
Semana 3	Documentación de Pruebas	Registrar avances, ajustes y resultados de las pruebas iniciales.	Julia	Toda la semana	4 horas
Semana 4	Optimización	Ajustar los parámetros en Python y Unity, basados en los resultados iniciales.	Todos los integrantes	Lunes a Martes	6 horas
Semana 4	Resultados Finales	Ejecutar simulaciones optimizadas y preparar las visualizaciones finales en Unity.	Miguel y Elit	Miércoles	6 horas
Semana 4	Preparación de la Presentación	Crear diapositivas y video del proyecto.	Todos los integrantes	Jueves a Viernes	8 horas
Semana 5	Revisión Final	Revisar el repositorio de GitHub y preparar la entrega formal del proyecto.	Miguel y Julia	Viernes	4 horas

Tabla 2

10. Repositorio con el código implementado

El código de la implementación final de los agentes como el código de la parte gráfica se encuentra dentro de un repositorio de GitHub

Código agentes : MODELO_CUIDAD/Modelo

Código (Unity) parte gráfica: MODELO_CUIDAD/UnityModel

Link Repositorio : https://github.com/MiguelCabreraVictoria/Equipo1_TC2008B/tree/main

11. Correcciones

Las correcciones se pidieron en el último avance, fue la implementación de algunos obstáculos en las calles como en las banquetas que pudieran afectar el trayecto de los agentes para eso se integraron algunos componentes como conos para que los carros como los peatones pudieran evitarlos, por otra parte se mejoró el comportamiento de los carros para que se respetan los semáforos con el fin de que haya un orden en el tránsito automovilístico y evitar pueda presentar un accidente, dentro del repositorio se encuentra un video, el cual se demuestra las correcciones descritas para un mayor entendimiento en el comportamiento de los agentes como del modelo.



12. REFERENCIAS

- Macal, C. M., & North, M. J. (2010). Tutorial on agent-based modeling and simulation. *Journal of Simulation*
- Helbing, D., & Balmelli, S. (2011). How to do agent-based simulations in the future: From modeling social mechanisms to emergent phenomena and interactive systems design. *Advances in Complex Systems*
- Nagel, K., & Schreckenberg, M. (1992). A cellular automaton model for freeway traffic. *Journal de Physique*