



Campus Santa Fe

Reto: Movilidad Urbana

Modelación de sistemas multiagentes con gráficas computacionales

Mauricio Emilio Monroy González - A01029647
Diego De la Vega Saishio - A01420632

Octavio Navarro
Gilberto Echeverría

4 de diciembre del 2025

Reto: Movilidad Urbana

Problema a resolver

Actualmente, navegar en una ciudad resulta complicado a raíz de distintos factores. Por ejemplo, el tráfico en la Ciudad de México es incluso similar al que se experimenta durante horas pico en Calcuta, Bangalore, Londres, Kyoto, entre otras, donde sus habitantes pasan más de cien horas al año en esos períodos. Un estudio realizado determinó que en la Ciudad de México, una persona que conduce un vehículo de motor pierde 152 horas de su vida en el tráfico (Zamarrón, 2025). Las congestiones viales pueden ser causadas por distintas razones, algunas como (Patel, 2025):

- Incremento de población y urbanización desmedida, sobrepasando la capacidad de flujo en calles, las cuales no habían sido planeadas para dicho ritmo. En la CDMX, los Kilómetros-Auto Recorridos (VKT por sus siglas en Inglés) se han triplicado, de 106 millones en 1990, a 339 millones en 2010.
- Mala infraestructura vial también pueden ser causadas por una mala infraestructura de las vialidades. El estado de una calle e intersecciones mal planeadas exacerbarán el problema. De igual manera, los semáforos desincronizados y en ubicaciones propensas a crear embotellamientos ralentizan el flujo vehicular, mientras que en otros puntos incluso hacen falta para regular el movimiento urbano.
- Finalmente, también queda considerar el comportamiento propio de los conductores, los cuales impiden que una planeación de las vías urbanas se refleje en la realidad, y al contrario, termine en accidentes.

Por lo anterior, para el propio desarrollo de actividades productivas de una ciudad y el aprovechamiento de tiempo personal, es esencial que el diseño y comportamiento del flujo vial esté cuidadosamente planificado.

Propuesta de solución

Dado lo anterior, el presente proyecto busca modelar un entorno de tráfico urbano con representaciones de semáforos y entidades móviles que buscan llegar a un destino particular mientras respetan señalizaciones y direcciones de flujo especificadas por un mapa.

a. Mapa

Las reglas de tráfico como ubicación de semáforos, direcciones de calles, destinos y edificios fue dada por un mapa predeterminado con caracteres mapeados a significados. Un semáforo “S” tenía una mayor duración que un semáforo “s”, la letra “D” es un posible destino para los agentes, y el “#” es para un obstáculo, el cuál representamos como un edificio de manera gráfica. Nos fue ofrecida la posibilidad de modificar caracteres como ^><v, los cuales representan direcciones, en puntos estratégicos para mejor orientar a los agentes en la simulación. Por ejemplo, en una intersección, un agente puede que no supiera que existe una continuación de la calle en la que transitaba al cruzar la perpendicular al mismo, por lo que un carácter en la perpendicular que lo acercara un poco más se introdujo de manera que se acercara al cruce sin dañar el flujo de la perpendicular, también. Otra modificación similar se aplicó en la sección de glorietas, para aclararle a los agentes una posible ruta de salida y que encontraran un camino válido. De igual manera, en pocas esquinas o en “avenidas” largas se introdujo un carácter que ayudaba a que los agentes identificaran con mayor facilidad una desviación, o incluso que evitaran el carril para dar curva si su intención era continuar sobre la misma avenida. Finalmente, algunos # se cambiaron por símbolos T o R, pero esto es meramente para lo visual de la simulación en el servidor Flask, sin afectar el funcionamiento del modelo, en donde T es

un árbol y R una calle sin dirección (aspectos decorativos que funcionan con la misma estructura del agente obstáculo-edificio). A continuación, se presenta el mapa original y el modificado.

Original:

Adaptado:

b. Generación de vehículos, e inicio y fin de simulación

El entorno al activarse, avanza 1 *step* cada 1000 ms (configurable en *random_agents.js*), y cada 10 steps (configurable en *agents_server.py*) se intentará poner en el modelo 1 caballo por esquina para así tener 4 en el mejor caso, y así sucesivamente. En el caso de que no haya una sola esquina disponible porque está ocupada por un caballo al momento de intentar generar más, la simulación se detendrá y se imprimirán en consola las principales métricas de desempeño recolectadas: total de caballos generados, caballos que llegaron a su destino, y caballos aún en tránsito hasta ese momento. De esta forma, es posible identificar la eficiencia con la que se mueven los caballos hacia su ruta, si en verdad están llegando a su destino o si el mapa está demasiado saturado. De otra forma, si al menos 1 esquina está disponible, la simulación continuará iterando.

c. Comportamiento de caballos

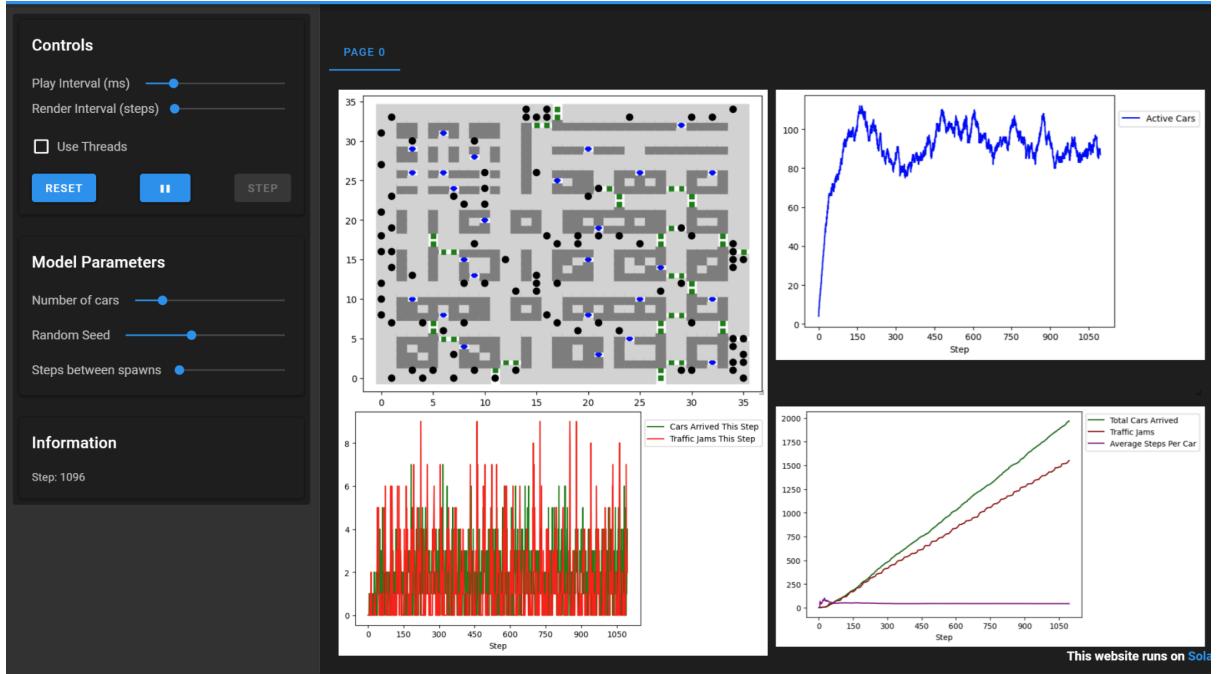
Naturalmente, los caballos respetarán señalizaciones viales y direcciones de tráfico (detenerse ante semáforos en rojo, no ir en sentido contrario, ...), pero de manera general, los agentes Car (caballo, gráficamente) implementan un sistema de navegación inteligente basado en A* para calcular rutas óptimas hacia destinos aleatorios, considerando penalizaciones por tráfico, semáforos y

dirección de las calles. La heurística utilizada es la distancia de Manhattan entre la posición actual y el destino, lo que permite estimar eficientemente el costo restante sin sobreestimarla (el cual fue un problema con el que nos encontramos). Los pesos del algoritmo están configurados con un costo base de 1.0 para movimientos rectos, mientras que los movimientos diagonales (utilizados para cambios de carril o giros) tienen un costo de 1.4 para representar la mayor distancia euclídea recorrida. Las celdas ocupadas por otros agentes reciben una penalización de 80 unidades cuando *avoidCars=True* para fomentar rutas alternas, los semáforos añaden un peso proporcional a su *timeToChange* multiplicado por 0.2 para reflejar el tiempo de espera esperado, y los movimientos contra el flujo de tráfico reciben una penalización de 150 unidades, desalentando fuertemente ir en sentido contrario sin bloquearlo por completo. Operan mediante una máquina de estados finitos con cinco estados: "*calculating*" cuando trazan una ruta nueva, "*moving*" mientras avanzan por su path, "*waiting*" al estar bloqueados temporalmente (por semáforo en rojo o tráfico), "*unjamming*" cuando intentan escapar de un embotellamiento tras superar un umbral de paciencia configurable, y "*arrived*" al alcanzar su destino. Los agentes recalculan su ruta A* de manera dinámica en tres situaciones: cuando entran al estado "*calculating*" por primera vez o al no tener un path válido, cuando están en estado "*waiting*" y han estado bloqueados por más de *recalculateThreshold* (5 steps, pero configurable) o superan su patience (2 steps), lo cual los lleva a "*unjamming*", y cuando están en "*unjamming*" y fallan más de *maxUnjammingAttempts* (2 intentos), momento en el que recalculan agresivamente sin evitar otros coches (*avoidCars=False*).

Adicionalmente, existe una probabilidad aleatoria (*escapeProb = 0.2*, es decir, 20%) de que recalculen inmediatamente al detectar que su path está bloqueado en estado "*moving*", lo que permite respuestas rápidas ante congestiones inesperadas. Todos estos parámetros son configurables en *agent.py* y en *model.py*, y no se optó por los mismos para evitar que el costo de procesamiento de A* y la detección de vecinos fuera demasiada para una computadora y resultara en problemas de rendimiento. Los agentes no pueden moverse lateralmente; solo avanzan en su dirección actual o realizan movimientos diagonales hacia adelante para cambiar de carril cuando el path está obstruido o dar vueltas, pero priorizando siempre avanzar en línea recta para no obtener movimientos erráticos en zig-zag. La toma de decisión para cambio de carril evalúa movimientos válidos hacia adelante que respeten la dirección de la calle y minimicen la distancia al destino, permitiendo maniobras evasivas sin retroceder. Además, implementamos mecanismos de paciencia (*waitCounter*) y desatascamiento (*unjammingAttempts*) para que los agentes recalculen rutas agresivamente o exploren alternativas después de estar bloqueados por varios steps, incrementando el contador de embotellamientos en el modelo cuando las estrategias de escape fallan. Este comportamiento adaptativo permite que los agentes naveguen de forma realista en situaciones de alta densidad vehicular, respetando reglas de tráfico mientras buscan eficiencia en su recorrido.

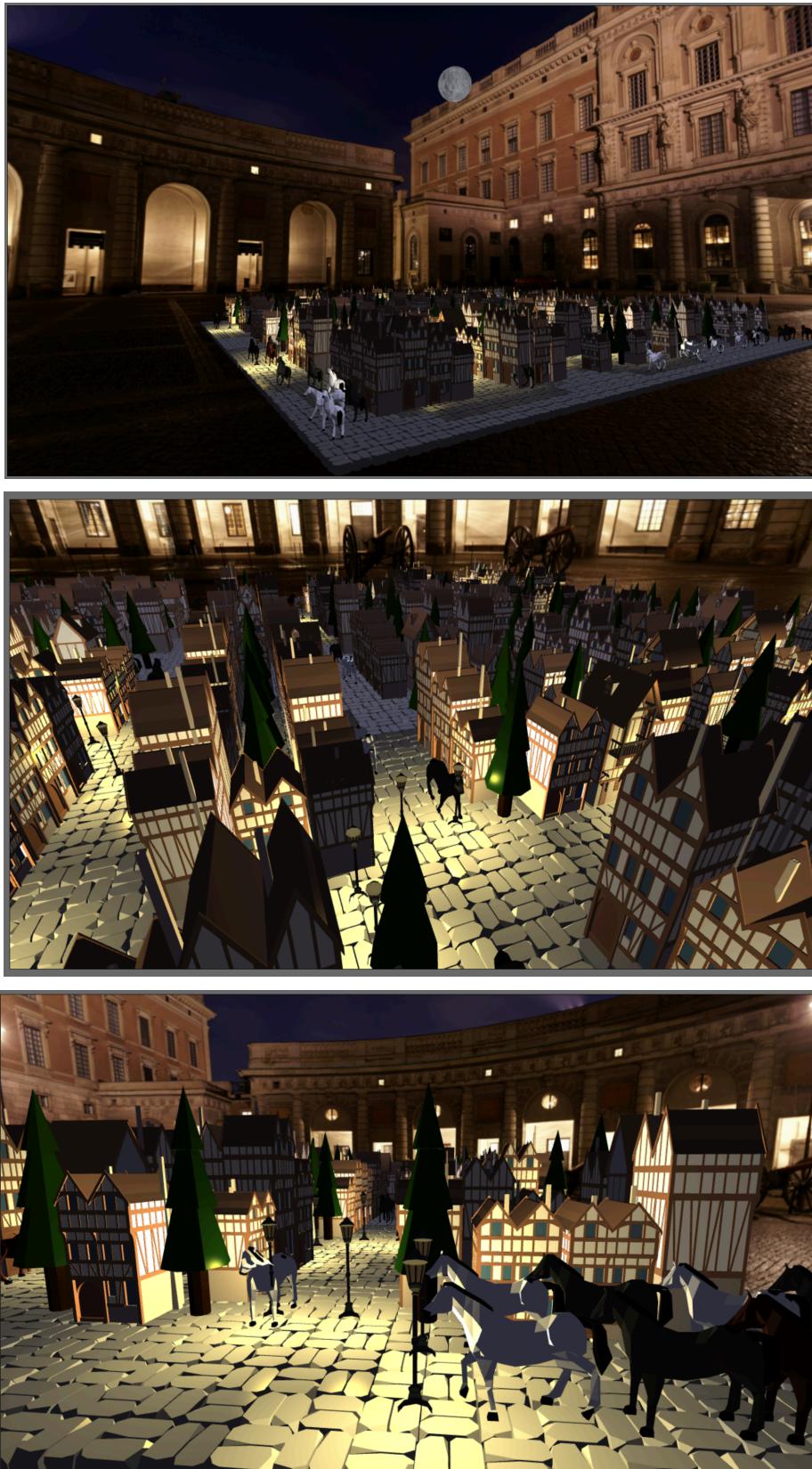
d. Ambientes

El modelo puede ejecutarse y visualizarse en 2 versiones de carga en servidores distintos. Una es en Solara utilizando *server.py*, donde es visible una cuadrícula con los agentes y es fácilmente distinguible entre ellos. A su vez, gráficas actualizadas a partir de información recolectada en cada step reflejan el estado actual de la simulación. Los deslizadores son los que previamente se usaban en clase, pero los únicos realmente importantes para la repetibilidad de una simulación son Random Seed y Steps between spawns, el cual determina qué tan seguido se buscará generar agentes nuevos. No se ha encontrado un máximo de Steps con el cual la simulación se detenga, ni siquiera configurando el spawnrate a 1 (en el siguiente snapshot fue con 2).



Por otro lado, el modelo también puede ser visible en un servidor de Flask, `agents_server.py`, el cual cargará una página html simple con un canvas que contiene un entorno 3D cuya intención era ambientar al modelo en una ciudad estilo victoriano. Para ello, al entorno se le implementaron múltiples características visuales como shaders GLSL y Phong para iluminación realista, shaders especiales para la luna con textura y emisividad de la luz, y shaders para el skybox cúbico. Este envuelve completamente la escena con 6 caras texturizadas que simulan una plaza italiana, y existe una luna 3D posicionada arriba del modelo como una fuente de luz principal que emite un 60% de brillo propio y utiliza una textura importada, y un sistema dinámico de iluminación donde solo los semáforos en verde generan lámparas (luces puntuales) que iluminan su entorno con atenuación por distancia (radio de 5 unidades) y color amarillo cálido, mientras que los semáforos en rojo no emiten luz. Los caballos se renderizan con tres variaciones de color aleatorias (café, blanco y negro) asignadas al momento de su creación, cada una con su propio conjunto de materiales MTL cargados desde archivos separados, y utilizan animaciones de 4 frames que cicla cada 120 milisegundos cuando están en movimiento, regresando a un frame IDLE estático cuando se detienen. Las posiciones de los caballos se interpolan suavemente usando lerp (linear interpolation) con un ciclo de 1000 ms para transiciones fluidas entre celdas, incluyendo un pequeño efecto de salto vertical (0.1 unidades) mediante una función seno durante el movimiento para que suban y bajen mientras avancen, y sus rotaciones se interpolan en 500 ms hacia la dirección nextDir recibida del servidor, permitiendo giros anticipados suaves antes de cambiar de dirección y evitar rotar después de la curva. Los edificios tienen alturas aleatorias (escala Y entre 0.2 y 0.4) para crear variedad en el skyline urbano y se rotan automáticamente para mirar hacia la calle más cercana según un algoritmo que detecta carreteras adyacentes con prioridad cardinal (Sur > Norte > Este > Oeste), mientras que los árboles se distinguen visualmente con una escala menor (0.2x0.4x0.2) y color verde. Las calles empedradas (modelo `Road.obj`) se colocan automáticamente debajo de edificios, semáforos y destinos para garantizar cobertura completa del terreno, y los destinos también se rotan para mirar hacia la calle más cercana con un offset adicional de 180° para compensar la orientación del modelo porque el mapa se genera invertido. El sistema implementa una estrategia de doble buffer temporal donde, mientras se renderiza un frame con las posiciones actuales de los agentes, en paralelo se ejecuta una actualización asíncrona

en segundo plano (`updateInBackground()`) que consulta al servidor Flask para obtener las nuevas posiciones sin bloquear el ciclo de renderizado. Esta actualización se dispara cuando la animación alcanza el 50% de su duración, lo que permite que las nuevas posiciones estén listas justo cuando termina la interpolación actual, logrando transiciones fluidas y continuas sin pausas visuales.



Diseño de agentes

Objetivo

Los agentes representados como caballos buscan llegar a una celda “destino” de la manera más rápida, la cual es asignada al mismo de manera aleatoria al ser creados. En el trayecto deben respetar el sentido de las vías, los señalamientos viales cambiantes (modelados como linternas que indican “avanzar” al prender y “detener” al apagarse), así como evitar crear embotellamientos y chocar con otros agentes u obstáculos.

Los demás agentes programados dentro de la simulación, como Roads, Obstacles y TrafficLights, no son agentes inteligentes, dado que no tienen un objetivo particular ni las demás características asociadas a un agente racional. Carecen de un criterio de desempeño (no optimizan nada), no poseen un entorno propio sobre el cual actuar, no cuentan con actuadores que transformen su realidad más allá de su mera existencia estática, ni disponen de sensores que les permitan percibir cambios o responder a estímulos. Tampoco muestran proactividad, ya que no anticipan ni planifican, ni reactividad, porque no modifican su conducta ante alteraciones del entorno. En síntesis, no cumplen con ninguno de los componentes del modelo PEAS, pues su función es únicamente estructural dentro del entorno. Sin embargo, por definición de la librería Mesa se programaron como agentes para garantizar su coexistencia y correcta interacción espacial con el caballo dentro del grid.

Características de agentes

Analizando el PEAS de los caballos:

- Performance: las métricas de desempeño del caballo fueron
 - Cantidad de steps para llegar al destino, la cual se busca minimizar
 - Evitar colisiones
 - Evitar embotellamientos, loops o celdas congestionadas
 - Cumplir reglas del entorno
- Environment: el entorno al cual se enfrentó el caballo fue
 - Cuadrícula con calles de sentido definido
 - Semáforos que determinan si puede o no avanzar según su estado
 - Otros caballos que puedan impedir sus movimientos
 - Obstáculos, como árboles o edificios
 - Destino al cual llegar como objetivo
- Actuators: el caballo podría realizar las siguientes acciones
 - Moverse a una celda hacia enfrente o en diagonal-frente (considerando esquinas para girar también). Por ejemplo:

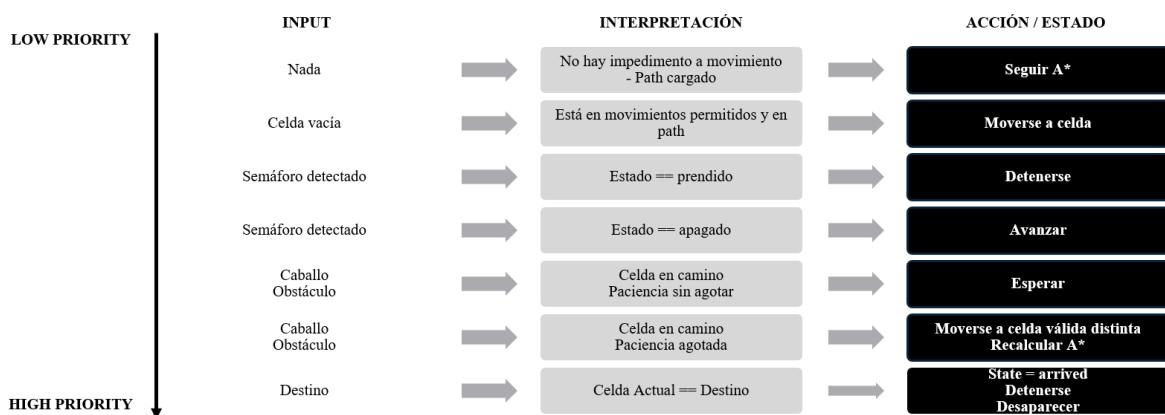
1	1	1	0	0	1	0	0	0	1	0	0
0	^	0	0	>	1	0	v	0	1	<	0
0	0	0	0	0	1	1	1	1	1	0	0
- Sensors: el caballo percibe
 - Estado de las celdas vecinas (ocupadas, libres, si son obstáculos, estado de semáforo)
 - Dirección de la calle en celda actual y celdas contiguas
 - Presencia de otros caballos
 - Ubicación del destino y celdas válidas para llegar a él

Asimismo, dado que se determina que el caballo es un agente inteligente porque es un sistema en un ambiente que es capaz de tomar acciones automáticas para cumplir sus objetivos con cierto grado de racionalidad, se puede analizar más a fondo. Tiene proactividad dado que toma la iniciativa de planear su ruta con A* y selecciona la mejor, así como reactividad, ya que cuando surge una

congestión, aparece un semáforo en rojo o una celda en su camino se bloquea, el caballo puede ajustar su ruta y comportamiento con cierta omnisciencia momentánea del mapa completo mientras usa A*. Eso sí, no tiene habilidad social porque no se comunica con otros agentes, solamente los detecta, así como no tiene aprendizaje de rutas previas o de otros agentes. No obstante, sí es racional en el sentido de que selecciona la acción que maximiza su desempeño esperado según la información con la que cuenta, y toma decisiones de manera autónoma. Finalmente, sí recopila información del entorno para las métricas del modelo y sus propios cálculos de ruteo y vecinos.

Arquitectura de subsunción

La manera en la que el caballo toma decisiones, de manera general, es representable a través del siguiente diagrama:



Es decir, la capa con mayor prioridad es la que revisa si ya llegó a su destino para detener su movimiento y cumplió su objetivo. Se cuenta con que el pathfinding ya precargó la ruta del agente desde que se generó. Finalmente, se considera que el agente reactivo tiene una arquitectura de Agente con Estado, ya que basa su estado actual en el ambiente sensado y a partir de reglas internas determina qué acción llevará a cabo con los actuadores a su disposición, que en este caso es movimiento puro.

Características de ambiente

El agente opera en un ambiente accesible, ya que de inicio conoce el mapa completo y cada que recalcula su ruta con A* también. Esto, combinado con cambios constantes en ocupación y semáforos, lo convierte también en un entorno no determinista, ya que una misma acción puede producir resultados distintos. El ambiente es no episódico, pues el desempeño depende de una secuencia continua de decisiones que afectan la ruta futura, evitando congestiones y adaptándose a variaciones del entorno. Además, es dinámico, dado que evoluciona aunque el agente no actúe, debido al movimiento simultáneo de otros caballos y al cambio de semáforos. También, el ambiente es discreto, porque tanto el espacio (celdas del grid) como las acciones y percepciones (ocupado/libre, lámpara prendida/apagada) están representados de manera finita y claramente separada.

Conclusiones

Mauricio Monroy

La simulación permitió entender tanto las fortalezas como los retos de diseñar agentes en un entorno urbano dinámico. Integrar A* con modificaciones para ajustar preferencias de movimientos, semáforos cambiantes, direcciones de vía y una arquitectura de subsunción dio lugar a movimientos más coherentes y evitó choques y congestiones, además de acercarnos a un comportamiento “inteligente” más consistente. Sin embargo, la presencia de múltiples agentes, los bloqueos en zonas estrechas y las posiciones de semáforos en continuidad directa a las áreas de spawn generaron rutas con tapones frecuentes, y esto causaba que los agentes frecuentemente llamaran a A* e incrementando el costo de recálculo, lo que nos obligó a considerar la complejidad computacional y a decidir cuidadosamente cuándo ejecutar ciertos algoritmos para no comprometer el ritmo general de la simulación. Aun así, las prioridades de la subsunción ayudó a mantener un comportamiento más claro, modular y controlado en cuanto a funciones llamadas, facilitando ajustes puntuales al solo cambiar atributos dentro de la clase del agente. Persisten áreas de oportunidad, como ampliar la percepción de los agentes, anticipar el movimiento de otros, mejorar la coordinación en zonas críticas y refinar la gestión de bloqueos y semáforos para aumentar la fidelidad y eficiencia del modelo. Estas mejoras futuras permitirían seguir avanzando hacia agentes más adaptativos y capaces de desenvolverse de manera robusta en entornos urbanos complejos. El proyecto me empujó a pensar de una manera más estructurada pero al mismo tiempo en un escenario donde yo solamente era el coordinador y la simulación acataría mis órdenes según le pareciera, lo cual fue una manera diferente de pensar a los programas que usualmente hacemos. Fue uno de mis primeros acercamientos a la inteligencia artificial, y quisiera continuar aprendiendo sobre tomas de decisión automatizadas con el fin de resolver tareas.

Diego de la Vega

Trabajar en esta visualización 3D nos hizo ver, por fin, todo el trasfondo que existe detrás de “renderizar un videojuego”. Descubrimos que nada es automático: cada modelo, textura o shader trae sus propios detalles y exige entender cómo funciona realmente el cálculo de la iluminación y materiales, entre la interpretación de los .obj’s y mtl’s. Aprendimos que cualquier pequeño error importa, desde una normal invertida hasta el orden exacto de las transformaciones o de actualización de datos para las interpolaciones. Uno de los retos que más se nos presentaron fue ajustar los modelos importados. Llegaban con escalas distintas o materiales que no coincidían con nuestra iluminación, y corregirlos significó reorientar objetos, reinterpretar MTLs y decidir cómo queríamos que el mundo se viera. También creo que la actividad nos ayudó a entender más sobre lo delicado que es el movimiento en tiempo real: una interpolación mal hecha o un ángulo mal calculado rompe por completo la naturalidad. Además, otro gran reto fue jugar con la iluminación para que se viera realista, pues la luz emitida por la luna “opacaba” la de las linternas, y eso daba como resultado, sombras que no deseábamos y que no coincidían con lo que se vería en la realidad. Resolverlo nos llevó a combinar geometría, y animación para suavizar trayectorias y darle vida a los agentes para que los caballos pareciese que caminaran y reflejaran correctamente la luz, combinando cambios de modelos con una transformación sinusoidal en cada step. Como conclusión, considero que esta práctica nos dejó claro que construir una escena interactiva no es solo “poner modelos en el canvas”, sino entender cómo cada pieza se integra, se transforma y se comporta dentro de un sistema coherente.

Recursos

Repository del proyecto

- https://github.com/A01420632/Equipo3_Multiagentes

Demostración

Servidor Flask:

- https://drive.google.com/file/d/1tm9KNwr0BwqBk_sHBhCz-P3H_x5oqb9D/view?usp=sharing

Servidor Solara:

- <https://drive.google.com/file/d/1EODIFp9ERmBUbKcncjHpEmqzKZQC-QC3/view?usp=sharing>

Referencias

- Zamarrón, H. (2025). *CDMX tiene el peor tráfico del mundo; ¿cuánto tiempo hace una persona manejando?* Milenio. <https://www.milenio.com/politica/comunidad/cdmx-entre-las-ciudades-del-mundo-con-el-peor-trafico>
- Patel, R. (2025). *What is Traffic Congestion?* Upper. <https://www.upperinc.com/glossary/route-optimization/traffic-congestion-constraints/>

Modelos utilizados

- Eldin. (s.f.). Night skyboxes [Skybox]. OpenGameArt.org. Recuperado de <https://opengameart.org/content/night-skyboxes>
- More 3D Studio (s.f.). Moon (natural satellite of Earth), Fab.com. Recuperado de <https://www.fab.com/listings/b6a35ec1-4d4c-47cf-a6e8-5400df96ab53>
- pablopri. (s.f.). Low poly cartoon pine christmas tree [Modelo 3D]. Free3D. Recuperado de https://free3d.com/3d-model/low-poly-cartoon-pine-christmas-tree-47325.html?dd_referrer=
- CGTrader. (s.f.). Download Page [Página web]. CGTrader. Recuperado de <https://www.cgtrader.com/items/2100356/download-page>
- Creazilla. (s.f.). Animales de granja [Modelo 3D]. Creazilla. Recuperado de <https://creazilla.com/es/media/3d-model/5228/animales-de-granja>
- Free3D. (s.f.). Old house [Modelo 3D]. Free3D. Recuperado de <https://free3d.com/es/modelo-3d/old-house-70840.html>