Modelación de sistemas multiagentes con gráficas computacionales

(Gpo 402)
Evidencia 2

Saraí Santiago Lozano | A01735331

Miguel Angel Edelman Vindel | A01705713

Christian Flores Alberto | A01734997

David Martinez Molina | A01735425

15/ 10 / 2023

Evidencia 2

Link al repositorio: https://github.com/SaraiSLoz/Evidencia2_Multiagentes

Información de los integrantes del equipo:

Miguel Angel Edelman Vindel:

- Fortalezas:Conocimientos en Programación, Comprensión de Sistemas, Habilidad en Diseño de Software, Análisis de Algoritmos
- Áreas de oportunidad: Simulación y Modelado, Inteligencia Artificial, Interacción con Bases de Datos
- Expectativas del bloque: Desarrollo de Habilidades Prácticas, Aplicación de Conceptos Aprendidos, Enfoque Multidisciplinario, Solución Eficiente

Sarai Santiago Lozano:

- Fortalezas: Liderazgo, Habilidad de entender el código, conocimiento de OpenGl, ayuda de correcciones.
- Áreas de oportunidad: Desarrollo de las clases que se ocuparan
- Expectativas del bloque: Espero aprender mejor OpenGL para el desarrollo de proyectos con simulaciones 3D, así como, poder mejorar en mi programación orientada a objetos.

Christian Flores Alberto:

- Fortalezas: Responsable, rápido y eficiente en trabajo y codificación, buen colaborador en equipo, conocimientos básicos en programación con multiagentes.
- Áreas de oportunidad: Programación y Modelado de simulaciones.
- Expectativas del bloque: Mejora y avance en mis conocimientos sobre el desarrollo de simulaciones, esperando aplicar estos nuevos conocimientos en una simulación de una problemática real, en este caso relacionada con el tráfico vehicular.

David Martínez Molina:

- Fortalezas: Constante, diligente, investigador, trabajo en equipo.
- Áreas de oportunidad: Mayor proactividad.
- Expectativas del bloque: Aprender a usar la teoría multiagente en un sistema 3D que simula una abstracción de la vida real.

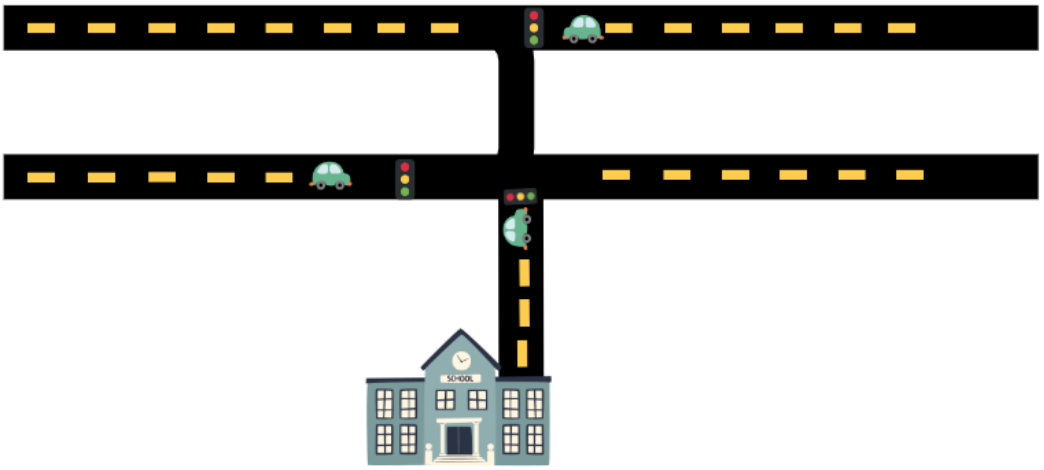
Descripción del reto:

El reto consiste en 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.

Imagina una solución que implemente una de las siguientes estrategias de ejemplo:

Controlar y asignar los espacios de estacionamiento disponible en una zona de la ciudad, evitando así que los autos estén dando vueltas para encontrar estacionamiento.
Compartir tu vehículo con otras personas. Aumentando la ocupación de los vehículos, reduciría el número de vehículos en las calles.
Tomar las rutas menos congestionadas. Quizás no más las cortas, pero las rutas con menos tráfico. Más movilidad, menos consumo, menos contaminación.
Que permita a los semáforos coordinar sus tiempos y, así, reducir la congestión de un cruce. O, quizás, indicar en qué momento un vehículo va a cruzar una intersección y que de esta forma, el semáforo puede determinar el momento y duración de la luz verde.

Propuesta:



Nuestra representación está basada en la salida del Tecnológico de Monterrey Campus Puebla, un problema enorme es la salida de los estudiantes hacia la incorporación de la Atlixcayotl, de igual manera el cruzar hacia el otro sentido es una situación que pone en riesgo la vida de muchas personas y se provoca un tráfico masivo, es por ello que

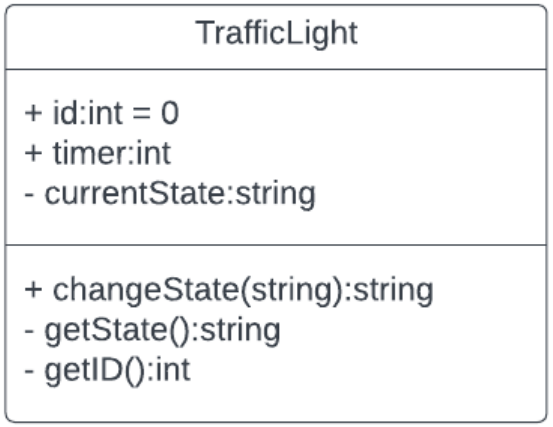
consideramos agregar 3 semáforos que permitan la salida controlada de los alumnos en su automóvil, todo esto lo coordinaremos e implementaremos en una simulación. Así mismo se evita el exceso de velocidad en ese tramo de la Atlixcayotl previniendo muchos accidentes.

Agentes involucrados:

Agente carro: Este agente es el grueso de nuestro programa, es el principal agente.

Agente semáforo: El agente semáforo indica el comportamiento del agente carro dependiendo de su atributo color en el momento dado.

Diagrama de clases:



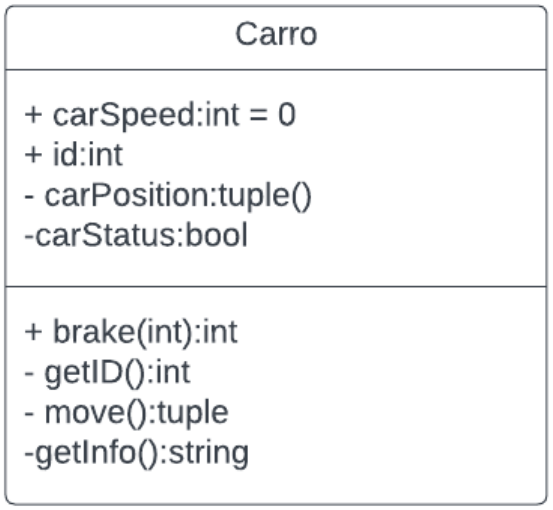
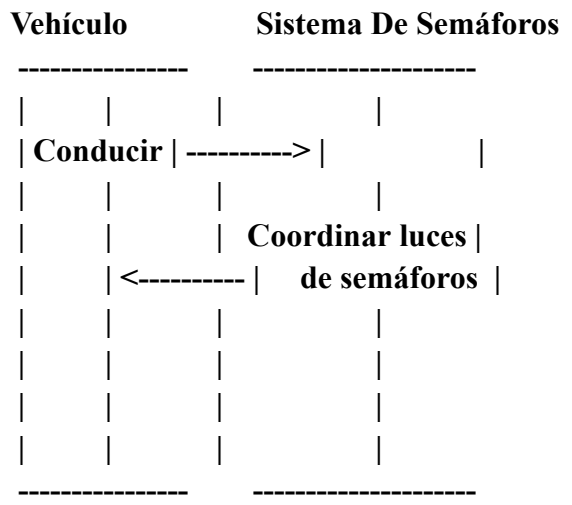


Diagrama de protocolos de interacción:



En este diagrama, el "Vehículo" lleva a cabo la acción de "Conducir", que incluye decisiones sobre cambiar de carril, respetar semáforos, y seguir la ruta. Mientras tanto, el "SistemaDeSemaforos" coordina las luces de los semáforos para regular el flujo vehicular en las intersecciones.

La interacción entre estos dos agentes implica que el vehículo debe responder a las señales de los semáforos, deteniéndose o avanzando según corresponda. El sistema de semáforos debe tomar decisiones sobre cuándo cambiar las luces para permitir que los vehículos se muevan de manera coordinada y eficiente.

Cronograma

	Christian	Sarai	Miguel Angel	David
Revisión 1	Diseño y planteamiento de propuesta.	Diseño 3D del automóvil de la simulación	Diseño de diagrama de protocolos de interacción.	Explicación de agentes y planteamiento de la solución.
Revisión 2	Desarrollo de sistema multiagentes.	Desarrollo de sistema multiagentes.	Desarrollo de sistema multiagentes.	Desarrollo de sistema multiagentes.
Revisión 3	Desarrollo del código.	Desarrollo del código.	Desarrollo del código.	—

	Documentación del código de la implementación de los agentes.	Plan de trabajo y aprendizaje adquirido.	Diagramas de clase y protocolos de interacción finales.	
Revisión 4	Desarrollo de presentación. Presentación.	Desarrollo de presentación. Presentación.	Desarrollo de presentación. Presentación.	—-

Actividades	Semana 1	Semana 2	Semana 3	Semana 4	Semana 5
Diseño y planteamiento de prropuesta					
Diseño de los agentes					
Diseño de diagramas					
Documentación Semana 1					
Desarrollo de sistema multiagentes					
Documentación Semana 2					
Desarrollo del código.					
Diagramas finales.					
Documentación del código.					
Documentación Semana 4					
Desarrollo de presentación					
Presentación					
Documentación final					

Plan de trabajo y aprendizaje:

Investigación y Análisis:

- Realizar un análisis profundo de la situación de movilidad urbana en México, identificando las principales ciudades afectadas, causas de congestión vehicular y patrones de tráfico.
- Investigar y recopilar datos de tráfico y movilidad urbana para utilizar como base en la simulación.

Definición de Objetivos:

- Definir objetivos claros para la simulación, como reducir la congestión vehicular, mejorar los tiempos de viaje y optimizar el flujo de tráfico.

Diseño del Modelo Multiagente:

- Identificar los tipos de agentes en el sistema, como vehículos, peatones, semáforos, etc.
- Definir las reglas y comportamientos de los agentes, como las decisiones de ruta, cambios de carril y decisiones de frenado.

Diseño de la Cuadrícula:

- Establecer una cuadrícula de la ciudad que representará el entorno urbano.
- Definir la topografía de la ciudad, incluyendo carreteras, intersecciones, semáforos y otros elementos de tráfico.

Fase 2: Desarrollo del Modelo Multiagente y Simulación

Implementación de los Agentes:

- Desarrollar clases para los diferentes tipos de agentes involucrados en la simulación, como vehículos, semáforos, peatones, etc.
- Definir los comportamientos y decisiones de cada tipo de agente.

Implementación de las Reglas de Tráfico:

- Codificar las reglas de tráfico y comportamientos de los agentes, como el seguimiento de las señales de tráfico, el cambio de carril y la adaptación de la velocidad.

Desarrollo del Sistema de Simulación:

- Construir el entorno de simulación, incluyendo la cuadrícula de la ciudad y la interacción de los agentes en ella.
- Implementar la lógica de simulación que permita avanzar en el tiempo y ejecutar las acciones de los agentes en cada paso.

Fase 3: Implementación de la Visualización y Pruebas

Desarrollo de la Visualización:

- Introducir gráficos y representaciones visuales que muestren la ciudad, los vehículos y la congestión del tráfico en tiempo real.
- Utilizar bibliotecas de visualización para crear una interfaz gráfica atractiva y comprensible.

Integración del Modelo y la Visualización:

- Conectar el modelo multiagente con la visualización gráfica para que los agentes y el entorno se reflejen adecuadamente en la interfaz.

Pruebas y Ajustes:

- Realizar pruebas exhaustivas de la simulación para garantizar que los agentes se comporten según lo previsto y que la visualización sea precisa.
- Ajustar parámetros y reglas según sea necesario para lograr resultados realistas y coherentes.

Fase 4: Evaluación y Optimización

Evaluación de Resultados:

- Ejecutar varias simulaciones con diferentes configuraciones y escenarios de tráfico para evaluar los resultados en términos de congestión vehicular, tiempos de viaje, etc.

Optimización del Modelo:

- Utilizar los resultados de las simulaciones para identificar áreas de mejora en el modelo y ajustar las reglas, parámetros y comportamientos de los agentes.

Proceso de Instalación y Ejecución

- Instalar e importar la librería numpy, flask, uuid, mesa para el uso de la parte de mesa.
- Instalar e importar las librerias pygame, requests, openGL, PyOpenGL PyOpenGL_accelerate
- Importar random, math, time

- Para ejecutar la simulación es necesario correr el archivo backend.py y pruebacarro.py en este orden. Es indispensable actualizar la dirección IP del local host en la línea #17 de la variable de URL_BASE del archivo pruebacarros.py.

Link a Video Demostrativo:

https://youtu.be/6tEd-TjYD9c?si=owdsei88OUgqwfmn

Explicación Mesa (movimiento y coordinación de los agentes)

simple_continuous_canvas.js

- Este código nos da una estructura para la visualización de figuras en 2D, en este caso se dibujaron círculos para la representación de los semáforos y los rectángulos como representación de los carros, así mismo tenemos métodos para renderizar y limpiar el canvas.

SimpleContinuousModule.py

- Importamos la clase VisualizationElement de un módulo llamado ModularVisualization correspondiente a la biblioteca de Mesa.
- Tenemos una clase llamada SimpleCanvas, hereda de VisualizationElement, el propósito de esta clase es crear la interfaz gráfica para visualizar la simulación en un entorno o espacio continuo.
- Los atributos de la clase son local_includes que contiene el nombre de un archivo .js , nombrado como simple_continuous_canvas.js, tenemos portrayal_method en None, que se utiliza para representar de forma gráfica un objeto en la interfaz gráfica, también establecemos el canvas_height y canvas_width en 500 cada uno.
- Tenemos el método __init__ que incluye el portrayal method y las alturas del plano.
- Creamos una instancia en javascript, con las dimensiones del lienzo. Agregamos la variable js_code y se le asigna un valor.
- También tenemos el método render, que recibe el modelo de la simulación, aquí se genera la representación gráfica de los agentes y el estado del modelo, retornando una lista que contiene la representación de los carros y semáforos. Finalmente se está proporcionando una estructura para la visualización gráfica de los elementos en la visualización.

Traffic.py

- Lo primero a realizar fue importar bibliotecas e importamos la clase SimpleCanvas del archivo SimpleContinuousModule, esta clase se encarga de la visualización de los agentes.
- Definimos la Clase Car, para el agente carro, esta clase hereda de Agent, que cuenta con atributos de color, posición, velocidad, estado y decisión, el método step actualiza la posición del carro en función de la velocidad, el método update_choice actualiza la elección del carro.
- Definimos la Clase Circle que hace referencia al agente semáforo, hereda de Agent, tiene atributos x, y para la posición, radio y color, el método change_color cambia el color del círculo, get_circle_portrayal devuelve la visualización del círculo y el método de change_color cambia el color del círculo esto se hace con la finalidad de que el semáforo vaya cambiando su color.
- Definimos la función car_draw que nos da la representación visual de nuestro carro, nos da la información para su dibujo dependiendo el color del carro y su tamaño.
- Tenemos las funciones para mantener distancia, respecto a los otros automóviles dentro de la simulación, esto para evitar las colisiones.
- Definimos la clase Street, que nos modela la calle, hereda de Model e incluye la lógica esencial de nuestro modelo, incluye el cambio de semáforos y el comportamiento de los carros.
- Contamos con las funciones reutilizar y reutilizar1, que nos ayudan a reubicar los automóviles, es decir como tenemos dos sentidos y dos carriles por cada sentido, seleccionamos el color de la dupla de un sentido la añadimos en reutilizar y el color de la otra dupla de otro sentido está en reutilizar1, esta función hace que si los agentes cumplen con ciertas condiciones el agente se reubica en la posición de los agentes Naranjas y se convierte en uno de ellos, la lógica de esta sección es que los agentes que van en horizontal se conviertan en agentes que salen en la parte vertical, de esta forma se evita que los carriles horizontales se saturen y de esta forma vamos a tener agentes naranjas incorporándose continuamente, gracias a esto pudimos evitar eliminar y agregar agentes, lo que hacemos ahora es reciclarlos o reutilizarlos.
- Posteriormente tenemos la función step, que ejecuta un paso de la simulación, actualizando el estado de los agentes y sus acciones correspondientes, aquí es donde se gestionan los cambios de color de los semáforos, también se controlan acciones y comportamientos de los carros en cada paso.
- Finalmente, configuramos la interfaz gráfica para la visualización, estableciendo como iniciar y configurar el modelo, así mismo configuramos el servidor de la simulación, pasando parametros y agregamos una función llamada setup_model, que configura un modelo tipo Street, se establece un contador de pasos de tiempo del modelo en 0 y retornamos el modelo creado.

Conexión entre Mesa y OpenGL

Para la conexión entre OpenGl y mesa, se ocupó un documento JSON para pasar los datos entre el documento traffic.py y PruebaCarros por el medio de backend.py. Define una ruta /games que maneja solicitudes POST. Cuando se realiza una solicitud POST a esta ruta, se crea un nuevo juego, en este caso es una clase de Street, se genera un ID único para él y se almacena en el diccionario games. Luego, la información de los agentes en el juego se recopila en una lista y se devuelve como una respuesta JSON.Define una ruta /games/<id> que maneja solicitudes GET. Cuando se realiza una solicitud GET a esta ruta con un ID de juego específico, se recupera ese juego del diccionario games. Luego, se llama al método step en el juego, que realiza una iteración del modelo de simulación. Después, la información actualizada de los agentes en el juego se recopila en una lista y se devuelve como una respuesta JSON.Finalmente, la aplicación Flask se ejecuta en el host "0.0.0.0".Este código crea un servidor web que permite interactuar con un simulador de tráfico y proporciona información sobre el estado actual del juego cuando se realiza una solicitud GET. El servidor se ejecutará en la dirección "0.0.0.0" y escuchará en el puerto predeterminado de Flask (5000).
Define la variable URL_BASE con la URL base del servidor web que aloja el simulador de tráfico. Realiza una solicitud POST a la URL base concatenada con "/games" utilizando la biblioteca requests. La opción allow_redirects=False se utiliza para evitar seguir redirecciones HTTP y, en su lugar, recibir la respuesta inicial. La respuesta de la solicitud POST se almacena en la variable r. Al mismo tiempo, la variable ocupa la función .json() para guardar toda la información en una lista, de la cual estaremos sacando la información para crear los objetos de carros y semáforos.

OpenGL (Gráficos, Texturas, Visualización y Sonido)

En el archivo pruebacarros.py, se importaron las librerías de OpenGL, pygame, requests, random, math y, de los agentes y elementos 3D de importó semaforo, edificio, faroles y Textures. Se inicializa una solicitud POST a un servidor web en la dirección base del localhost para crear un nueva simulación. Luego, se extrae la URL de la ubicación donde se aloja el juego recién creado. La respuesta del servidor en formato JSON, se convierte en un objeto almacenado en la variable "lista". Posteriormente, se configuran las dimensiones y propiedades del entorno de visualización de OpenGL.

En la función Init(), se inicializa el entorno gráfico y 3D utilizando la biblioteca Pygame y OpenGL. Se establece la pantalla del juego con las dimensiones especificadas y define la proyección de la cámara utilizando la función gluPerspective(). Además, configura la cámara utilizando la función gluLookAt(). La función también configura varios parámetros importantes para el entorno gráfico, como el color de fondo, la prueba de profundidad, el modo de polígono, y define una serie de coordenadas para los árboles, edificios y faroles en el entorno. Además, carga varias texturas que representan el suelo y los edificios. De las listas que se crearon en el backend.py de los agentes de traffic.py se recorre la lista en [0], que contiene información sobre los coches en el juego. Para cada elemento en esta lista, se crea un objeto de coche Car utilizando las coordenadas proporcionadas por el agente, junto con

otros parámetros como el color y la escala. Estos objetos de coche se agregan al diccionario Cars con la clave agent["id"]. De igual forma, se recorre la lista en [1], que contiene información sobre los semáforos en la simulación. Para cada elemento en esta lista, se crea un objeto de Semaforo utilizando las coordenadas proporcionadas por el agente, junto con otros parámetros como el color y las dimensiones. Estos objetos de semáforo se agregan al diccionario Semaforos con la clave agent["id"].

En la función display(), se renderizan los objetos. Se utiliza glClear() para borrar el búfer de color y el búfer de profundidad. Luego, se habilita la textura y se renderiza el plano del suelo con textura de césped y textura de carretera utilizando la función glBindTexture() y glBegin(GL_QUADS) para dibujar cuadriláteros. Las texturas se cargan de una lista de texturas denominada textures. Posteriormente, la función renderiza los objetos de la simulación, incluyendo semáforos, edificios, árboles, faroles y coches. Para cada tipo de objeto, se ejecuta un bucle que recorre la lista correspondiente (por ejemplo, Semaforos.values() o Arboles) y llama al método draw() de cada objeto.

Finalmente, en un ciclo for se recorre la lista de agentes de coches y semáforos, que se obtiene de las respuestas del backend. Para cada agente de coche, se obtiene su identificador y su color. Luego, se comprueba si el identificador del coche ya existe en el diccionario de Cars. Si el coche ya existe, se llama al método update() del coche correspondiente para actualizar su posición y su color en función de los valores obtenidos del backendI. Para los agentes de semáforos, se realiza un proceso similar. Se obtiene el identificador del semáforo y se comprueba si ya existe en el diccionario de Semaforos. Si el semáforo ya existe, se llama al método update() del semáforo correspondiente para actualizar su color en función de los valores obtenidos del backend. Si el semáforo no existe en el diccionario Semaforos, se ignora y se continúa con el siguiente agente.

La función play_sound(), se utiliza para reproducir un archivo de sonido utilizando la biblioteca pygame. La función toma un parámetro sound_file, que representa la ruta del archivo de sonido que se desea reproducir. En la simulación se escucha un sonido de tráfico de una ciudad. En la función main(), se llaman a las funciones principales y se crea un control del movimiento de la pantalla con el teclado.

En el archivo supercar.py, se utilizó el archivo objloader para cargar al objeto de un carro así como sus materiales (archivo .mtl). Adicionalmente, se definieron sus funciones de actualización, dibujo, normalización y escalamiento; se definió una orientación según el carril en el que se encuentra dependiendo de su id/color.

En el archivo semaforo.py, se crea la clase semaforo y se definen sus atributos. Se dibuja un cilindro que representa el objeto y otro medio cilindro que representa un protector trasero del semáforo. Los objetos se rotaron según la dirección del carril y se define el color según el parámetro correspondiente.