Tecnológico de Monterrey

Modelación de Sistemas Multiagentes con Gráficas Computacionales (Gpo 401)

Evidencia 1

“Actividad Integradora”

Jacqueline Villa Asencio | A01736339
Estefany Vanessa Castañeda Hernandez | A01730601
José Juan Irene Cervantes | A01736671
Augusto Gómez Maxil | A01736346
Hugo Muñoz Rodríguez | A01736149

29 de Septiembre de 2023

Interacción con API

Para la realización de ésta evidencia, utilizamos tanto "Mesa" como "OpenGL" al hacer una conexión y realizar la simulación de los robots limpiadores en un formato 3D. Para ello, mesa proporciona implementaciones API gráficas para dibujar los gráficos en pantalla, dónde Mesa actúa como el vínculo entre la aplicación y la API gráfica.

Archivo “backend.py”

Primeramente se cuenta con un código en python llamado "backend.py", el cual utiliza el framework Flask para crear una aplicación web que expone dos rutas (endpoint) relacionadas con la creación y consulta de juegos.

Dentro de dicho código se importa la librería "flask", utilizada para crear la aplicación web, junto con la librería "jsonify" que importa la función 'jsonify' de Flask que será utilizada para convertir objetos python en respuestas JSON.

Se crea un diccionario llamado "games" para almacenar instancias de la clase "sala" y se crea una instancia de la aplicación Flask llamada "app".

Se define una ruta con el método POST, en donde se crea un nuevo juego generando un identificador único para posteriormente crear una instancia de la clase "Sala" y se almacena en el diccionario game con el id como clave. Después se itera sobre los agentes en el juego recién creados y se construye una lista que contiene información sobre cada agente: id, coordenada en x, coordenada en z. Finalmente se devuelve una respuesta JSON con la lista de agentes.

Al definir una rusa con el método GET, nos permite consultar el estado del juego de acuerdo a su id y se ejecuta el método "step()" con el fin de avanzar un paso en la simulación. Se itera sobre los agentes en el juego y se construye una lista que contiene la información sobre cada agente. Finalmente, se devuelve una respuesta JSON con dicha lista.

Archivo “incinerador.py”

Se definen tres clases, las cuales dos de ellas representan a los agentes dentro de la simulación:

- **Bot.-** Representa a un robot limpiador en la simulación, el cual se mueve por la sala y recolectan basura para llevarla al incinerador. Tiene un estado "w_trash" que indica si lleva basura o no, y se mueve por el plano hasta que recogen la basura. Una vez que hayan recogido algo, se dirigen al incinerador para quemar dicha basura y repetir el procedimiento.

- **Incinerador.-** Representa un incinerador en la simulación y cuenta con dos estados: ON (encendido) y OFF (apagado), junto con un temporizador para cambiar entre dichos estados después de cierto tiempo.

- **Basura.-** Representa un objeto de basura en la sala. Puede estar en uno de los dos estados establecidos: COLLECT (recolectado) o UNCOLLECT (no recolectado).

La **clase "Sala"** es la clase principal del modelo de simulación, la cual se encarga de crear la cuadrícula, los agentes y la basura. La basura se coloca de manera aleatoria en la cuadrícula, excluyendo las posiciones del incinerador y los robots.

Archivo “Cubo.py”

Dicho archivo contiene una clase llamada "Cubo" que representa a un cubo 3D; inicializa los vértices del cubo y sus colores. Cuenta con un atributo 'Position' que almacena la posición del cubo en el espacio y cuenta con métodos para cargar el modelo 3D del auto que recoge la basura, por ejemplo el método "cargar" y "draw2".
La clase también cuenta con la función "update" que actualiza la posición del cubo en el espacio 3D, recibiendo nuevas coordenadas tanto en x como en y. De igual forma cuenta con la función "draw" que dibuja dicho cubo en 3D.

Archivo “PlanosCubo.py”

Dicho código crea una ventana gráfica y renderiza los objetos tridimensionales en dicha ventana. De igual forma, interactúa con una API web al realizar una solicitud POSR para obtener información sobre la posición y el tipo de agentes dentro de la simulación.

Archivo “objloader.py”

Se utiliza para cargar y renderizar objetos en formato Wavefront OBJ en un contexto OpenGL. El código proporciona funcionalidades para cargar modelos OBJ, incluyendo sus vértices, normales, coordenadas de textura y materiales, y luego generar listas de visualización OpenGL para renderizar los modelos.

Liga al repositorio Git

https://github.com/jossjic/botCollector3D/tree/b07eaf0a3f7fe03a0d3cdda9a1cf01d1d51cab7e

Aportaciones

- **José Juan Irene Cervantes:** Funcionamiento del pathfinding, conexión mesa-OpenGL, refactorización del código, gestión de versiones y generación de estructuras de datos.
- **Augusto Gómez Maxil:** Pruebas de ejecución de versiones, pruebas de texturizado en agentes, apoyo en correcciones al algoritmo de pathfinding y la creación de matrices.
- **Hugo Muñoz Rodríguez:**Movimiento de los robots, aparición y modificación de la densidad de la basura, recolección de basura, funcionamiento del pathfinding, todo esto en mesa.
- **Jacqueline Villa Asencio:** Cargado de texturas y objetos en 3D dentro de la simulación de acuerdo a los tipos de agentes presentados.
- **Estefany Vanessa Castañeda Hernandez:** Pruebas de ejecución, cargado de texturas, apoyo con modelación de agentes.

Reflexiones personales

- **José Juan Irene Cervantes:** Me pareció interesante cómo se pueden utilizar protocolos como el HTTP para hacer la conexión entre el backend y frontend, había visto cómo se utilizaba en desarrollo web, por lo que me sorprendió verlo en un proyecto de esta índole. Se me hizo interesante poder dividir las tareas y recursos en computadoras distintas para aligerar la carga. Finalmente, quiero decir que lo que más se me dificultó fue la sincronización de ambos lados, ya que hay diversos factores que pueden interferir en el rendimiento del proyecto.
- **Augusto Gómez Maxil:** Hacer la conexión entre OpenGL y Mesa fue algo retador; el frontend junta aspectos de OpenGl y el uso de objetos en 3D; mientras que el backend nos permitió hacer uso de lo aprendido en mesa; el movimiento de los robots y la implementación del path finding, con el objetivo de realizar una simulación más realista conforme a los elementos que hubiera dentro del programa. Lo qué me llamó la atención fue las diferencias de ejecución que había entre las computadoras del equipo.
- **Hugo Muñoz Rodríguez:** La implementación de mesa de con OpenGL fue bastante interesante ya que podemos interactuar con el frontend con el back end, creando una simulación en 3D usando casos prueba en el backend ya que consume menos recursos y se pueden hacer pruebas mucho más retadoras.
- **Jacqueline Villa Asencio:** Fue algo nuevo e interesante, pudimos aprender acerca de la conexión que existe entre mesa y OpenGL, notando que se respetan muchas cosas del código original en OpenGL a pesar de tener comunicación con el API web del cual recibía respuestas, y el cómo con solo modificar el back, el front mostraba la simulación deseada. Así mismo, se conoció la importancia de json, el cuál sinceramente creí que no tenía mucha relevancia.

- **Estefany Vanessa Castañeda Hernandez:** Al realizar este proyecto, me pareció interesante el cómo se realizó la conexión entre mesa con OpenGL, así como la comunicación que había que realizar con el API web. De igual manera, fue retador tanto la parte del frontend que contiene aspectos de OpenGL para crear la simulación 3D como del backend que con ello pudimos darle movimiento a los robots, así como controlar la interacción entre los mismos y los elementos dentro de la simulación. Fue algo retador pero al final muy satisfactorio el ver cómo se integraban las dos partes.