Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Puebla

Modelación de sistemas multiagentes con gráficas computacionales (Gpo 502)

Evidencia 1. Actividad Integradora

Elvis Alain Calzada Trinidad - A01738650

Alberto Silva Oloarte - A01738457

Daniel Carballido Carreón - A01738467

Humberto Pérez Galindo - A01732526

Profesores:
Luciano Garcia Bañuelos
Iván Olmos Pineda

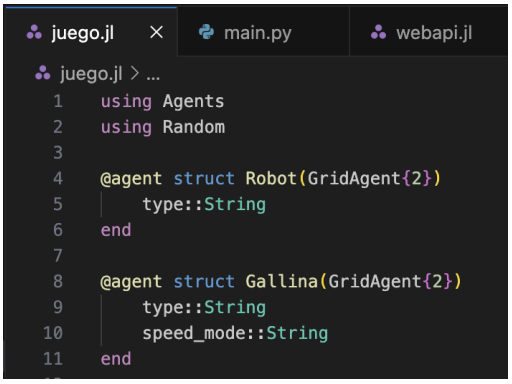
Noviembre 11, 2025

Funcionamiento general del juego (propuesta inicial): *M4. Actividad*

Backend - Sistema de Simulación Multiagente en Julia

El backend desarrollado en Julia utiliza Agents.jl v4.11.2 para la simulación multiagente y Genie.jl v5.0.0 para la API REST. El sistema opera en un espacio grid de 20x20 celdas donde cada celda representa una posición discreta en el mundo de simulación.

Definición de Agentes y Estructuras:

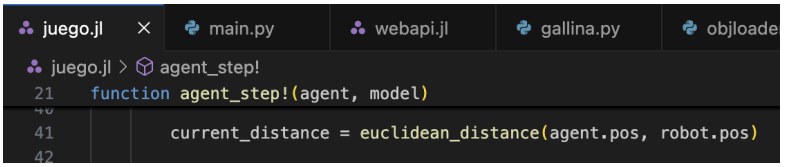


Parámetros clave:

- ➢ Grid 2D: 20×20 celdas (400 posiciones posibles)
- ➢ Robot: Agente controlado externamente(teclado), ID: 1
- ➢ Gallinas: n agentes autónomos, IDs: 2, 3, 4…
- ➢ Radio de Peligro: 5.0 celdas (constante FLEE_RADIUS)

Toma de decisiones (*agent_step*):

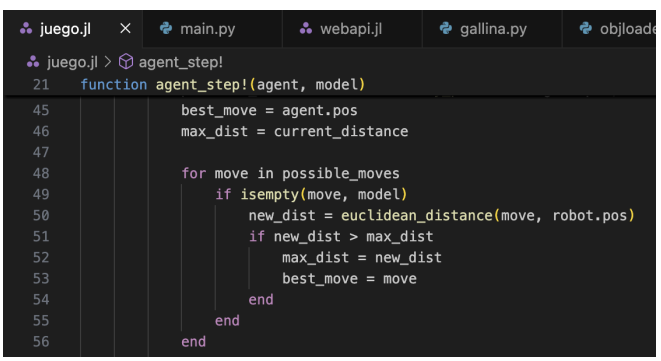
1. Detección del robot → Cada gallina escanea todos los agentes para localizar al robot en cada step.
2. Cálculo de distancia →



Fórmula: $\sqrt{[(x_1 - x_2)^2] + [(y_1 - y_2)^2]}$ *donde* $x, y \in [1, 20]$

3. Clasificación de movimientos:

Caso A: Distancia < 5 → MODO HUIDA

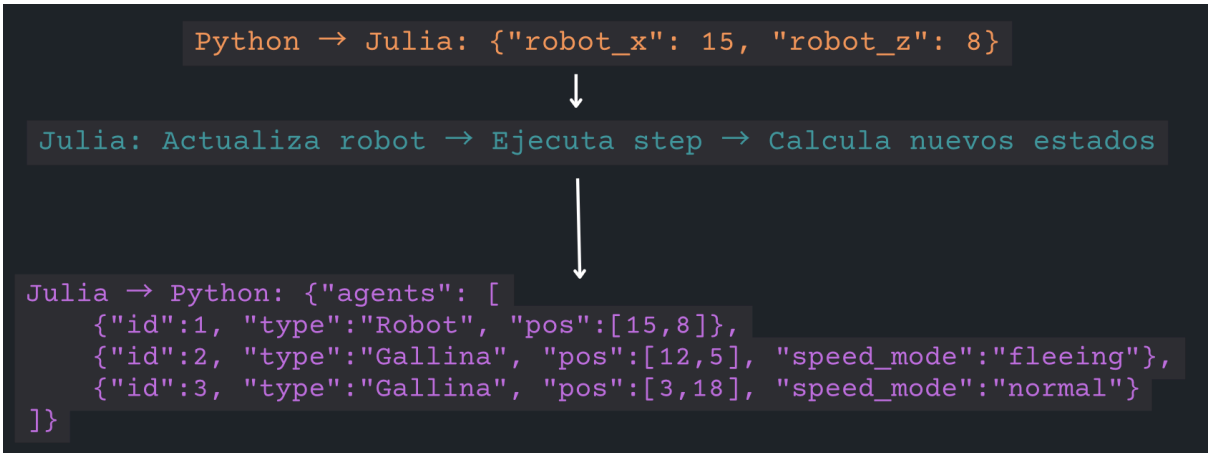


Se evalúan hasta 4 movimientos posibles (N,S,E,O) y se selecciona el que maximiza la distancia.

Caso B: $Distancia \geq 5$ celdas MODO PASEO →

Movimiento aleatorio restringido a celdas vacías.

Sistema de comunicación API REST: Flujo de Datos

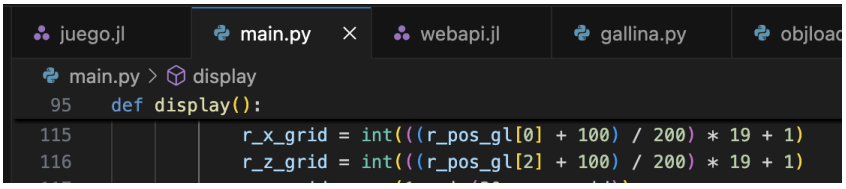


Procesamiento por Step:

- ➢ Recepción: Coordenadas del robot desde Python
- ➢ Validación: clamp(robot_x, 1, 20) asegura coordenadas válidas
- ➢ Actualización: move_agent!(robot, nueva_posición, model)
- ➢ Simulación: step!(model, agent_step!, 1) ejecuta lógica autónoma
- ➢ Respuesta: Estado JSON de todos los agentes

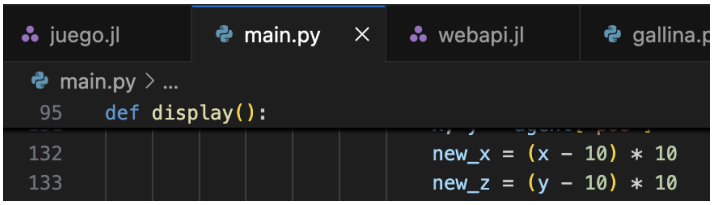
Mapeo de Coordenadas Grid ↔ 3D

Python → Julia (3D → Grid)



[-100,100] → [1,20]

Julia → Python (Grid → 3D)



[1,20] → [-95,95]

Frontend

El desarrollo del *frontend* se realizó en **Python**, utilizando las librerías **PyGame** y **OpenGL** para la renderización y control de los objetos 3D.

El objetivo principal de este módulo es construir el entorno visual del mini juego, integrando la lógica de movimiento, animaciones y detección de colisiones, en correspondencia con el modelo de simulación de agentes.

Componentes gráficos y animaciones

El entorno visual está compuesto por tres elementos principales:

1. **El mapa o escenario 3D**,
2. **Las gallinas (entidades secundarias manejadas por julia con multiagentes)**
3. **El robot (personaje principal controlado por el jugador)**.

El **mapa** corresponde a un modelo 3D estático exportado en formato `.obj`, que representa una granja. Este escenario no presenta animaciones propias, pero incorpora una **lógica de colisiones** que impide que los personajes atraviesen árboles, muros u otros obstáculos del entorno. Su función principal es actuar como base del espacio de simulación.

Las **gallinas** están modeladas como objetos compuestos. Cada gallina se encuentra dividida en partes independientes —cuerpo, alas y patas— lo que permite aplicar animaciones diferenciadas a cada componente. Las patas ejecutan un movimiento de marcha alterno, mientras que las alas realizan un aleteo sincronizado cuando el agente se desplaza. Esta estructura modular facilita futuras expansiones del comportamiento (por ejemplo, patrones de escape o búsqueda).

El **robot**, personaje principal controlado por el usuario, también está compuesto por partes separadas: cuerpo y brazos izquierdo y derecho. Durante el desplazamiento, los brazos realizan un movimiento alternado hacia adelante y atrás (simulando un paso natural), mientras el cuerpo presenta un leve **balanceo vertical** para imitar el efecto de gravedad y dar una sensación más orgánica al movimiento.
Cuando el robot se detiene, los brazos retornan gradualmente a su posición inicial vertical.

```python
def update(self, is_moving):
    """
    Actualiza el ángulo del brazo. Si el robot se mueve, se balancea.
    Si está quieto, regresa a su posición original.
    """
    if not is_moving:
        # Regresar suavemente a la posicion inicial
        if abs(self.swing_angle) > self.swing_speed:
            self.swing_angle -= math.copysign(self.swing_speed, self.swing_angle)
        else:
            self.swing_angle = 0
        return

    self.swing_angle += self.swing_speed * self.swing_direction

    # Invertir la direccion del balanceo al alcanzar el límite de 45 grados
    if abs(self.swing_angle) > 45.0:
        self.swing_direction *= -1
```

Codigo de movimientos de los brazos del robot

Lógica de interacción

El robot puede **desplazarse libremente dentro de los límites definidos del tablero**, respetando las restricciones de colisión con el entorno.
Cuando el jugador alcanza una gallina, esta **desaparece del mapa** y el **contador de captura** en pantalla aumenta. El objetivo del juego es recolectar todas las gallinas para completar la simulación.

Detalles técnicos del renderizado

El renderizado de los modelos 3D se realiza mediante un **cargador de archivos `.obj` personalizado (objloader.py)**, que interpreta las coordenadas de vértices, normales y texturas, y genera listas de visualización en OpenGL para optimizar el desempeño.
Cada objeto puede incluir materiales definidos en archivos `.mtl`, los cuales se procesan para aplicar texturas y parámetros de iluminación.
El sistema cuenta con una **configuración de cámara dinámica**, que sigue al robot manteniendo una distancia y altura relativa, ofreciendo una perspectiva en tercera persona.

```python
import os
import pygame
from OpenGL.GL import *

class OBJ:
    generate_on_init = True
    @classmethod
    def loadTexture(cls, imagefile): ⋯

    @classmethod
    def loadMaterial(cls, filename): ⋯

    def __init__(self, filename, swapyz=False): ⋯

    def generate(self, no_textures=False): ⋯

    def render(self):
        glCallList(self.gl_list)

    def free(self):
        glDeleteLists([self.gl_list])
```

Codigo de carga de objetos en 3D (para mas información revisar el repositorio)

```python
def move(self, keys):
    """
    Procesa la entrada del teclado para actualizar el estado del robot.
    """
    is_moving = False
    is_moving_forward = False

    if keys[pygame.K_LEFT]:
        self.rotation_y += self.turn_speed
    if keys[pygame.K_RIGHT]:
        self.rotation_y -= self.turn_speed

    # --- Camara ---
    # Actualizar el vector de direccion DESPUES de cambiar la rotacion
    self.update_direction()
    dir_x = self.direction[0]
    dir_z = self.direction[2]

    if keys[pygame.K_UP]:
        self.position[0] += dir_x * self.speed
        self.position[2] += dir_z * self.speed
        is_moving = True
        is_moving_forward = True
    if keys[pygame.K_DOWN]:
        self.position[0] -= dir_x * self.speed
        self.position[2] -= dir_z * self.speed
        is_moving = True

    if is_moving_forward:
        self.bob_angle = (self.bob_angle + self.bob_speed) % 360
        self.vertical_bob = abs(math.sin(math.radians(self.bob_angle))) * self.bob_height
    else:
        if self.vertical_bob > 0.1:
            self.vertical_bob -= 0.1
        else:
            self.vertical_bob = 0
            self.bob_angle = 0
```

Funcion move que se encarga del calculo del movimiento del robot especialmente el salto horizontal al moverse

Asimismo, se incluye un **Skybox** con texturas que rodea la escena, aportando una sensación de profundidad y ambiente al entorno de juego.

Delimitación del Dimboard

Con el objetivo de que todos los agentes permanezcan dentro del dimboard delimitado por la granja y evitar que se desborden a los extremos, se implementó la función check_boundaries(), la cual verifica y ajusta las coordenadas de los agentes para que estén dentro de los límites definidos.

```python
def check_boundaries(x, y, z, object_radius=0):
    """Verifica y ajusta las coordenadas para que estén dentro de los límites"""
    x = max(X_MIN + object_radius, min(X_MAX - object_radius, x))
    y = max(Y_MIN, min(Y_MAX, y))
    z = max(Z_MIN + object_radius, min(Z_MAX - object_radius, z))
    return x, y, z
```

Compromiso de Desarrollo Futuro

Comportamiento de Parvada

Como compromiso de mejora continua, se implementará un sistema de comportamiento de parvada para las gallinas cuando se encuentren en modo de paseo. Esta funcionalidad incorporará los principios de inteligencia de grupo que incluirán:

- Cohesión grupal: Movimiento hacia el centro de masa del grupo cercano
- Alineación direccional: Sincronización en la dirección de movimiento entre vecinos
- Separación inteligente: Mantenimiento de distancia personal para evitar colisiones
- Comportamiento emergente: Patrones complejos surgiendo de reglas simples

Parámetros técnicos planificados:

- Radio de influencia grupal: 3.0 celdas
- Mínimo de gallinas para activar flocking: 4 individuos
- Pesos comportamentales: Cohesión 40%, Alineación 35%, Separación 25%
- Transición suave entre comportamiento individual y grupal

Este milestone representará una evolución significativa en el realismo del simulation y añadirá profundidad estratégica al gameplay, creando desafíos más complejos para el jugador al enfrentar grupos coordinados en lugar de individuos aislados.