II Ciclo 2025
EIF-207 Estructuras de Datos
Primer Proyecto Programado

Courier Quest

1. Introducción

En este proyecto los estudiantes desarrollarán un videojuego en **Python** utilizando una librería de desarrollo de juegos (se sugiere **Arcade**, **Pygame** o **cocos2d**). El juego, llamado *Courier Quest*, simula a un repartidor que debe aceptar y completar pedidos en una ciudad, gestionando tiempos de entrega, clima, inventario y prioridades.

La información del mundo de juego (ciudad, pedidos, clima, etc.) será proporcionada a través de un **API alojado en**:

https://tigerds-api.kindflower-ccaf48b6.eastus.azurecontainerapps.io

2. Objetivos de aprendizaje

- Implementar y justificar el uso de **estructuras de datos lineales**
- Practicar el manejo de **archivos en múltiples formatos** (JSON, texto, binario)
- Aplicar **algoritmos de ordenamiento** en escenarios reales
- Desarrollar un videojuego con **Python y Arcade/Pygame/cocos2d**
- Integrar un **API real** y gestionar caché para trabajar en modo offline
- Diseñar un bucle de juego consistente con reglas cuantificables (clima, reputación, resistencia)

3. Jugabilidad (Gameplay)

El jugador controla a un **repartidor en bicicleta** que debe moverse por una ciudad representada como una **cuadrícula de calles, edificios y parques**.

Al **comienzo de la partida**, el jugador recibe una **meta de ingresos** que debe alcanzar antes de que termine la jornada laboral simulada (por ejemplo, 10–15 minutos de tiempo real).

Selección de pedidos

Los pedidos serán de dos clases: normales (0) o con prioridad (n) – a mayor el número, mayor prioridad. El jugador puede aceptar o rechazar pedidos conforme avanza el juego, pero sólo puede cargar una **cantidad máxima de peso**. Los pedidos aceptados se almacenan en el **inventario** que puede recorrerse hacia adelante o hacia atrás para decidir el orden de entrega.Sin embargo, el inventario podrá tener opciones de visualizar los encargos con mayor prioridad u ordenarlos por hora de entrega.

Movimiento por la ciudad

El repartidor se desplaza en bicicleta **casilla por casilla** por las calles de la ciudad. Para completar un pedido debe:

1. Ir al punto de recogida (pickup).
2. Transportar el paquete en el inventario.
3. Entregarlo en el punto de destino (dropoff).

Factores que afectan al jugador

El rendimiento del repartidor depende de varias variables interconectadas:

Resistencia: se representa con una barra de 0–100.

- Baja con el movimiento, sobre todo si lleva mucho peso o si el clima es adverso.
- Si llega a 0, el jugador queda **exhausto** y no puede moverse hasta recuperarse al 30%.

Clima: cambia automáticamente cada 45-60 segundos siguiendo una **cadena de Markov**.

- Lluvia, tormentas, viento o calor extremo reducen la velocidad y aumentan el consumo de resistencia.
- La transición entre climas es progresiva, para que los cambios se sientan naturales.

Peso del inventario: cada pedido tiene un peso; a mayor peso, menor velocidad y mayor gasto de energía.

Reputación: comienza en 70/100 y sube o baja según la puntualidad y las acciones del jugador.

- Entregas puntuales → aumentan la reputación.
- Retrasos o cancelaciones → la reducen.
- Con reputación alta (≥90) el jugador recibe un **5% extra en pagos**.

- Si baja de 20, el juego termina en derrota inmediata.

Condiciones de victoria y derrota

- **Victoria:** Alcanzar la meta de ingresos antes de que acabe el tiempo.
- **Derrota:** reputación < 20 o fin de la jornada sin haber cumplido la meta.

Puntuación y tabla de récords

Al final de la partida, el juego calcula un **puntaje final** en base a:

- Total de ingresos (afectados por la reputación).
- Bonos por terminar temprano.
- Penalizaciones por cancelaciones o retrasos severos.

Los resultados se guardan en un archivo **JSON de puntajes.**

Además, debe existir una opción para guardar el juego con el estado actual en cualquier momento, así como una opción para cargar una partida guardada. Finalmente, deben tener una opción de "deshacer" N cantidad de pasos por parte del jugador para volver a un estado anterior.

4. Datos del mundo (API y JSON)

El API expone los recursos mínimos: mapa, pedidos y clima. Cada recurso tiene un espejo local JSON para modo offline.

Esquema de ciudad (mapa)

```
{
 "version": "1.0",
 "width": 20,
 "height": 15,
 "tiles": [ ["C","C","C","B","B"], ["C","P","C","C","B"], ["B","C","C","C","C"] ],
 "legend": {
  "C": {"name":"calle","surface_weight":1.00},
  "B": {"name":"edificio","blocked":true},
  "P": {"name":"parque","surface_weight":0.95}
 },
 "goal": 3000
}
```

Esquema de pedidos

```
[
 {
  "id": "PED-001",
  "pickup": [3,7],
```

```
  "dropoff": [10,2],
  "payout": 120,
  "deadline": "2025-09-01T12:30:00",
  "weight": 2,
  "priority": 0,
  "release_time": 0
 },
 {
  "id": "PED-002",
  "pickup": [6,1],
  "dropoff": [9,9],
  "payout": 200,
  "deadline": "2025-09-01T12:10:00",
  "weight": 1,
  "priority": 1,
  "release_time": 45
 }
]
```


Esquema de clima por ráfagas (in-game)


```
{
 "city":"TigerCity",
 "date":"2025-09-01",
 "bursts":[
  {"duration_sec":90,"condition":"clouds","intensity":0.2},
  {"duration_sec":75,"condition":"rain","intensity":0.6},
  {"duration_sec":80,"condition":"storm","intensity":0.4}
 ],
 "meta":{"units":{"intensity":"0-1"}}
}
```


5. Clima dinámico

- Cada burst dura entre **45 y 60 segundos** de tiempo de juego.
- Al terminar ese tiempo, se debe **elegir la siguiente condición climática**.

Condiciones soportadas y multiplicadores base de velocidad (para la bicicleta)

- clear: ×1.00
- clouds: ×0.98
- rain_light: ×0.90
- rain: ×0.85
- storm: ×0.75
- fog: ×0.88
- wind: ×0.92
- heat: ×0.90
- cold: ×0.92

Consideraciones

Aquí entra en juego la **cadena de Markov**:

- Se define una **matriz de transición** donde cada fila representa la probabilidad de pasar de un estado climático al siguiente.
- Ejemplo simple (sólo con 3 climas: despejado, nublado, lluvia):

Condición actual → Despejado →	**Nublado →**	**Lluvia**
Despejado 0.6	0.3	0.1
Nublado 0.3	0.5	0.2
Lluvia 0.2	0.4	0.4

Si el clima actual es **nublado**, hay un 30% de que vuelva a despejado, 50% de que siga nublado y 20% de que cambie a lluvia.

Además de la condición, cada ráfaga tiene un **nivel de intensidad (0–1)**.

Para que el cambio entre climas no sea brusco, la transición entre la condición actual y la nueva debe hacerse **en 3–5 segundos** interpolando los efectos (ej. velocidad del jugador, visibilidad).

Ejemplo: si pasamos de `Mclima=1.0` (despejado) a `Mclima=0.85` (lluvia), el multiplicador se va ajustando gradualmente.

En la práctica del juego

1. Se inicia con la primera condición (ejemplo: `clear`).
2. Se fija un temporizador de duración (ej. 80 segundos).
3. Al expirar, se usa la matriz de Markov para **sortear** la siguiente condición.
4. Se determina la intensidad.
5. Se inicia transición suave y empieza un nuevo temporizador.
6. El ciclo se repite hasta que termine la partida.

6. Resistencia del repartidor

- Barra 0–100 (inicia en 100). Umbral de recuperación para moverse: 30.
- Estados: Normal (>30), Cansado (10–30, velocidad ×0.8), Exhausto (≤0, no se mueve).
- Consumo por celda: -0.5 (base) + extras:

- Peso total > 3: -0.2 por celda por cada unidad sobre 3.
- Clima adverso (rain/wind): -0.1 por celda; storm: -0.3 por celda; heat: -0.2 por celda.
- Recuperación parada: +5 por segundo (puntos de descanso opcionales: +10/seg).

7. Reputación (0–100)

- Inicial: 70. Derrota inmediata si <20. Excelencia (≥90): +5% pago.
- Cambios comunes:
  - Entrega a tiempo: +3
  - Entrega temprana (≥20% antes): +5
  - Tarde ≤30s: -2; 31–120s: -5; >120s: -10
  - Cancelar pedido aceptado: -4
  - Perder/expirar paquete: -6
  - Racha de 3 entregas sin penalización: +2 (una vez por racha)

Efectos: multiplicador de pago (+5% si ≥90), primera tardanza del día a mitad de penalización si reputación ≥85.

8. Velocidad y movimiento

- v0 (bicicleta) = 3 celdas/seg (ajustable)
- Fórmula final:
  v = v0 * Mclima * Mpeso * Mrep * Mresistencia * surface_weight(tile)

  Donde:

  Mpeso = max(0.8, 1 - 0.03 * peso_total)
  Mrep = 1.03 si reputación ≥ 90, si no 1.0
  Mresistencia = 1 (normal), 0.8 (cansado), 0 (exhausto)
  surface_weight proviene de la leyenda del mapa (ej.: parque 0.95)

9. Condiciones de victoria/derrota y puntaje

- Puntaje final (sugerido):
  score_base = suma de pagos * pay_mult (por reputación alta)
  bonus_tiempo = +X si terminas antes del 20% del tiempo restante
  penalizaciones = -Y por cancelaciones/caídas (opcional)
  score = score_base + bonus_tiempo - penalizaciones

La tabla de puntajes se almacenará en un archivo JSON de manera ordenada (de mayor a menor).

10. API del proyecto

- Base: https://tigerds-api.kindflower-ccaf48b6.eastus.azurecontainerapps.io

- Documentación: https://tigerds-api.kindflower-ccaf48b6.eastus.azurecontainerapps.io/docs

  GET /city/map → JSON de mapa
  GET /city/jobs → JSON de pedidos
  GET /city/weather → JSON de ráfagas de clima

11. Modo offline y caché

- Ante fallo del API, cargar archivos locales equivalentes:

  /data/ciudad.json,
  /data/pedidos.json,
  /data/weather.json

Guardar copias cacheadas de respuestas en /api_cache con fecha/hora.Si no hay conexión al API server, se utiliza la última versión.

12. Persistencia de archivos

• Guardado binario: /saves/slot1.sav
• Puntajes: /data/puntajes.json

13. Consideraciones

- El proyecto debe realizarse en grupos de máximo 3 personas. No se permite la entrega individual. Además, pueden trabajar con personas de otro horario

- El proyecto debe estar alojado en un repositorio de Github (a la hora de la entrega, solamente deben proporcionar el link donde está el código fuente del programa)

- Está permitido consultar documentación y pedir ayuda a un asistente de IA, pero deben registrar en un archivo de bitácora los **prompts utilizados** y las **modificaciones realizadas**

- Se penalizará la copia directa sin explicación del código

- El código debe cumplir las normas PEP8, y debe estar debidamente documentado

- Fecha de entrega: Domingo 28 de septiembre a las 11:59pm

- Deben incluir un archivo README.md con información general del proyecto, incluyendo las estructuras de datos que utilizaron y para qué partes del programa. Así como detalles de la complejidad algorítmica

15. Criterios de evaluación (100%)

- Estructuras utilizadas (20%)
- Algoritmos y rendimiento (20%)
- API+caché (15%)
- Archivos (15%)
- Jugabilidad (20%)
- Código y Documentación (10%)