Universidad Nacional de Moquegua
Escuela Profesional de Ingeniería de Sistemas e Informática

Paralelizacion de pixeles - Trabajo Final

ALGORITMOS PARALELOS

Estudiante: Carlos Eduardo Patina Rojas
Docente: Honorio Apaza Alanoca

26 de julio de 2025

1. INTRODUCCIÓN

Hoy en día, el procesamiento de imágenes digitales es una de las áreas más activas y con mayor crecimiento dentro del mundo de la computación. Lo vemos en todas partes: desde videojuegos hasta aplicaciones científicas y médicas. Uno de los principales desafíos en este campo es lograr que las animaciones se vean fluidas y en tiempo real, lo que no es nada fácil, ya que se necesita procesar muchísima información visual en muy poco tiempo. En el caso de las animaciones, eso significa generar rápidamente muchos cuadros o frames, lo cual puede volverse bastante pesado si se hace de forma secuencial. Y como cada vez se exige mayor calidad visual, también se requiere que las técnicas de procesamiento sean más rápidas y eficientes.

Es aquí donde entra la paralelización, una solución bastante útil que permite dividir las tareas y repartir el trabajo entre varios núcleos del procesador. En este informe se presenta un proyecto que justamente aplica esta técnica: se paraleliza el procesamiento de píxeles usando Python, apoyado en librerías como threading, NumPy, Pillow y Matplotlib. La animación creada muestra un dinosaurio moviéndose sobre un escenario con objetos fijos como nubes y piedras. El propósito principal de este trabajo es mostrar que, usando bien los hilos y recursos de ejecución simultánea, se puede mejorar el rendimiento del sistema sin comprometer la calidad visual del resultado final.

2. RESUMEN

Este proyecto aborda el problema del procesamiento secuencial de imágenes en animaciones, implementando una solución basada en paralelización mediante hilos en Python. Se genera una animación compuesta por 120 frames, donde un dinosaurio animado se desplaza sobre un escenario fijo compuesto por una imagen de fondo y objetos gráficos creados mediante matrices. Para lograr una ejecución eficiente, se divide la generación de los frames entre cuatro hilos utilizando la biblioteca threading, mientras que el control de acceso a los recursos compartidos se maneja mediante mecanismos de sincronización como Lock.

El uso de NumPy permite representar tanto el fondo como los objetos dentro de una única matriz de escena, lo que simplifica su manipulación. En lugar de superponer imágenes, se actualiza directamente la posición del dinosaurio dentro de esta matriz, fotograma a fotograma. La biblioteca Pillow ayuda a convertir la matriz en imágenes visibles, y Matplotlib permite mostrar la animación de forma clara. Gracias a la paralelización del proceso, se logran generar los fotogramas en menos tiempo sin afectar la calidad visual del resultado.

3. PLANTEAMIENTO DEL PROBLEMA

El principal problema abordado en este proyecto es la ineficiencia del procesamiento secuencial en la generación de animaciones, especialmente cuando se requiere producir una gran cantidad de frames en tiempo limitado. En sistemas donde cada cuadro es procesado de manera lineal, el tiempo de ejecución total puede crecer rápidamente, afectando negativamente la fluidez de la animación y la experiencia del usuario.

Ante este desafío, surge la necesidad de adoptar técnicas que permitan distribuir la carga computacional entre varios núcleos o hilos, de manera que las tareas puedan ejecutarse en paralelo. El presente trabajo propone como solución la implementación de procesamiento paralelo utilizando hilos en Python para dividir el trabajo de generación de frames, logrando así una reducción significativa del tiempo total de ejecución.

4. OBJETIVOS PLANTEADOS

4.1. Objetivo general

Implementar un sistema de paralelización de píxeles en Python utilizando la paralelización usando threading, con el fin de optimizar el tiempo de generación de frames en una animación.

4.2. Objetivos específicos

- Diseñar una escena animada con un personaje (dinosaurio) desplazándose sobre un fondo fijo.
- Representar los objetos de la escena utilizando matrices binarias en NumPy.
- Dividir la generación de frames entre múltiples hilos usando threading.
- Controlar adecuadamente el acceso a recursos compartidos utilizando Lock.
- Visualizar la animación utilizando Matplotlib.
- Evaluar los resultados en términos de eficiencia y calidad visual.

5. PROPUESTA

La propuesta consiste en diseñar un sistema animado donde un dinosaurio se desplaza lateralmente a lo largo de un escenario compuesto por una imagen real como fondo, complementada con elementos como nubes y piedras representadas en matrices. El personaje animado también se representa mediante una matriz que se mezcla con el fondo utilizando técnicas de blending.

Para mejorar la eficiencia, se genera un total de 120 frames, divididos equitativamente entre 4 hilos mediante la biblioteca threading. Cada hilo se encarga de generar una porción de la animación, lo que reduce el tiempo de procesamiento total. Para evitar errores derivados del acceso concurrente a recursos compartidos, se utiliza threading.Lock, asegurando que los hilos escriban de manera controlada en la lista global de frames.

Finalmente, se utiliza la biblioteca Matplotlib para visualizar la animación, mientras que Pillow se encarga de cargar y redimensionar la imagen de fondo. Esta arquitectura modular y paralelizada permite una ejecución eficiente y una animación fluida.

6. MARCO TEÓRICO

6.1. Programación Paralela en Python

La programación paralela es una estrategia computacional que busca ejecutar tareas de manera simultánea para mejorar el rendimiento. Consiste en dividir un problema grande en partes más pequeñas que pueden resolverse al mismo tiempo en diferentes unidades de procesamiento. Python, aunque a menudo considerado más lento en el procesamiento de operaciones puramente computacionales debido al Global Interpreter Lock (GIL), ofrece herramientas como threading y multiprocessing para la concurrencia y el paralelismo.

El GIL en Python permite que solo un hilo de Python se ejecute a la vez. Sin embargo, cuando un hilo realiza operaciones que no implican código Python puro, como la manipulación de arreglos con NumPy o tareas de entrada/salida (I/O), el GIL se libera. Esto significa que threading es especialmente útil en situaciones donde las operaciones intensivas en cómputo están delegadas a bibliotecas optimizadas escritas en lenguajes como C (como NumPy), o cuando las tareas implican esperas por recursos externos. Para una comprensión más profunda sobre el GIL y su impacto, puedes consultar el artículo .[En]tendiendo el GIL de Python: Por qué es importante para la concurrencia"de Bitácora del Programador (2023).

6.2. Biblioteca Threading

La biblioteca threading es un módulo estándar de Python que permite crear y gestionar hilos de ejecución. Cada hilo se ejecuta de forma independiente, pero comparten el mismo espacio de memoria del proceso principal. Esto los hace adecuados para tareas que involucran esperas, como operaciones de lectura o escritura de archivos. En este proyecto, donde las operaciones pesadas de manipulación de imágenes están encapsuladas en funciones optimizadas de NumPy, threading es ideal para preparar los frames de la animación de forma concurrente.

El módulo threading facilita la creación de nuevos hilos con la clase threading.Thread, a la cual se le asigna la función que cada hilo debe ejecutar. Para evitar problemas de concurrencia como las condiciones de carrera (cuando varios hilos intentan modificar el mismo recurso al mismo tiempo), se utiliza threading.Lock. Un Lock asegura que solo un hilo tenga acceso a una sección crítica del código (la parte que modifica un recurso compartido) en un momento dado. Esto es fundamental en tu código para asegurar que la lista frames, donde se almacenan las imágenes generadas, se actualice correctamente sin sobrescrituras o errores. Para más información sobre la implementación de hilos y locks en Python, el equipo de Programación Fácil (2021) ofrece una guía en su artículo Çoncurrencia en Python: Hilos y Procesos".

6.3. NumPy y Representación Matricial

NumPy es una biblioteca esencial en Python para el cálculo numérico, especializada en el manejo eficiente de arreglos multidimensionales (matrices) y operaciones vectorizadas. Su gran velocidad se debe a que gran parte de su núcleo está escrito en C, lo que le permite realizar cálculos numéricos de forma muy eficiente.

En el contexto de este proyecto, NumPy es fundamental para representar tanto el escenario como los objetos (como la nube, la piedra y el dinosaurio) como matrices de píxeles. Cada píxel se representa por un valor o un conjunto de valores que indican su color. Por ejemplo, en tu código, los objetos se definen como matrices binarias (con 0s y 1s), donde 1 indica la presencia de un píxel del objeto. Esta representación matricial simplifica enormemente operaciones como la mezcla de colores (como se ve en la función colocar_objeto), la rotación y el posicionamiento de los elementos en el escenario. Al trabajar con matrices, se pueden aplicar transformaciones a todos los píxeles de un objeto de forma rápida y eficiente. Para ejemplos prácticos y una introducción en español, puedes consultar el artículo "NumPy: La Biblioteca Fundamental para la Computación Numérica en Python"de Datascientist (2020).

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	1	1	1	0
0	0	0	0	0	0	0	0	0	0	1	1	1	1	0
0	0	0	0	0	0	0	0	0	0	1	1	1	1	0
0	0	0	0	0	1	1	1	1	0	1	1	1	1	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0

Figura 1: Representación de la matriz y los objetos

6.4. Pillow (PIL) para el Manejo de Imágenes

Pillow es una biblioteca para la manipulación de imágenes en Python, derivada de la popular Python Imaging Library (PIL). Es una herramienta muy versátil que permite

cargar, redimensionar, rotar, recortar y convertir imágenes entre una variedad de formatos comunes como JPEG, PNG y BMP.

En tu implementación, Pillow se utiliza para cargar la imagen de fondo (imagen.jpg) y convertirla en un arreglo RGB compatible con NumPy. Esta conversión permite trabajar con la imagen como si fuera una matriz de datos, facilitando su incorporación directa en la escena. En lugar de aplicar técnicas como alpha blending, lo que haces es insertar o actualizar directamente los valores del dinosaurio dentro del arreglo del fondo, sin necesidad de fusionar píxeles ni manejar transparencia. Para aprender más sobre cómo usar Pillow, el artículo "Manipulación de imágenes con Pillow en Python"del blog de Adalid Martínez (2022)

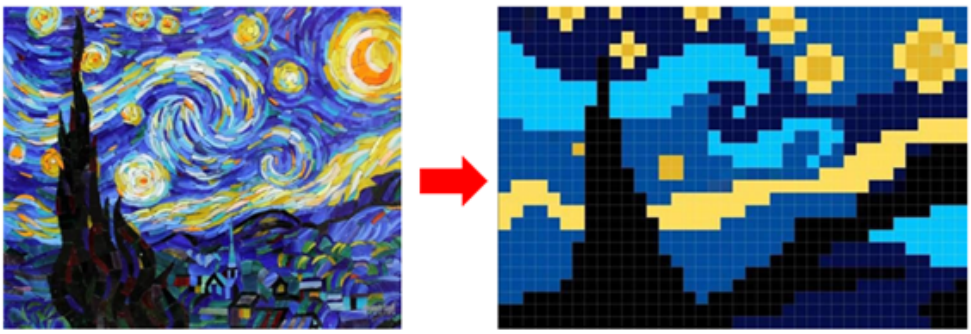

Figura 2: Funcionamiento de la librería PIL ajustando tamaños

6.5. Inserción directa de objetos en la matriz del escenario

En lugar de técnicas complejas como alpha blending, este proyecto utiliza una forma directa de integrar los objetos en la escena. Cada elemento (como el dinosaurio o las nubes) es representado como una matriz de valores, y se inserta dentro de una matriz principal que representa el escenario. No se realiza fusión de colores ni transparencia: los valores del objeto reemplazan los valores existentes en la posición correspondiente del fondo, esto es más sencillo computacionalmente y se adapta bien a proyectos donde se prioriza la eficiencia y el control total de los píxeles.

6.6. Visualización con Matplotlib

Matplotlib es una biblioteca fundamental en Python para la creación de gráficos y visualizaciones en 2D. Es muy utilizada en el ámbito científico y de análisis de datos para generar una amplia gama de representaciones visuales, incluyendo gráficos, imágenes y animaciones.

En este proyecto, Matplotlib es la herramienta clave para mostrar la animación. Se utiliza la función matplotlib.animation.FuncAnimation para generar la secuencia de frames a partir de la lista frames que se va construyendo. Esta función invoca repetidamente

una función de actualización (actualizar en tu código) que toma el frame correspondiente y lo muestra. El método imshow() es el encargado de representar cada imagen (matriz de píxeles) en la figura. La propiedad blit=True en FuncAnimation es una optimización importante: asegura que solo se redibujen las partes de la imagen que han cambiado entre un frame y el siguiente, lo que mejora significativamente la fluidez y el rendimiento de la animación, especialmente en animaciones complejas o con muchos frames. Para aprender más sobre animaciones con Matplotlib, puedes consultar el tutorial Çómo crear animaciones con Matplotlib en Python"de Data Science para Todos (2021).

6.7. Sincronización y Locks

Cuando se trabaja con programación concurrente, como es el caso de tu animación donde varios hilos generan frames simultáneamente y escriben en una misma lista (frames), es crucial implementar mecanismos de sincronización. Sin ellos, pueden ocurrir condiciones de carrera: situaciones donde el resultado del programa depende del orden impredecible en que los hilos acceden y modifican los datos compartidos. Esto podría llevar a frames incompletos, datos corruptos o incluso errores en la aplicación.

Para evitar estas condiciones de carrera y garantizar la integridad de los datos, se utiliza threading.Lock. Un Lock funciona como una cerradura: un hilo debe .ªdquirir.el Lock antes de poder acceder a una sección de código que manipula recursos compartidos. Si otro hilo intenta adquirir el Lock mientras está ocupado, ese hilo se detiene (se bloquea) hasta que el Lock sea liberado. Una vez que el hilo ha terminado su operación en la sección crítica, "libera.el Lock, permitiendo que otro hilo lo adquiera. En tu código, la sentencia with lock: asegura que solo un hilo a la vez pueda escribir en la lista frames, lo cual es esencial para que cada frame se almacene correctamente y en el orden esperado. Para una explicación más detallada sobre los locks en Python, el artículo "Mecanismos de Sincronización en Python: Locks, Semáforos y Eventos"de Pyladies Madrid (2019) es una excelente referencia.

7. IMPLEMENTACIÓN

El sistema se implementó en Python utilizando las bibliotecas mencionadas en el marco teórico. A continuación, se describe la estructura del código y su funcionamiento:

7.1. Configuración inicial

Se definieron los parámetros básicos de la animación, como el número de frames, las dimensiones del escenario y la cantidad de hilos a utilizar:

```
NUM_FRAMES = 120
ANCHO_ESCENARIO = 100
ALTO_ESCENARIO = 40
NUM_HILOS = 4
```

7.2. Representación de objetos

Los objetos (nube, piedra y dinosaurio) se modelaron como matrices binarias en NumPy, donde `1` indica la presencia del objeto y `0` el fondo. Por ejemplo, el dinosaurio se define como:

```
dino = np.array([
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,1,1,1,0,0,0,0,0,0],
        [0,0,0,0,1,1,1,1,1,1,0,0,0,0,0],
        [0,0,0,1,1,1,1,1,0,1,1,1,0,0,0],
        [0,0,0,1,1,1,1,1,1,1,1,1,0,0,0],
        [0,0,0,0,1,1,1,1,1,1,1,0,0,0,0],
        [0,0,0,0,0,1,1,1,1,0,0,0,0,0,0],
        [0,1,0,0,0,1,1,1,1,0,0,0,0,0,0],
        [0,1,1,0,1,1,1,1,1,1,0,0,0,0,0],
        [0,1,1,0,1,1,1,1,1,1,1,1,1,0,0],
        [0,0,1,1,1,1,1,1,1,1,1,0,1,0,0],
        [0,0,1,1,1,1,1,1,1,1,1,0,0,0,0],
        [0,0,0,1,1,1,1,1,1,1,0,0,0,0,0],
        [0,0,0,0,1,1,1,1,1,0,0,0,0,0,0],
        [0,0,0,0,1,0,0,1,0,0,0,0,0,0,0],
        [0,0,0,0,1,1,0,1,1,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
    ])
```

7.3. Paralelización con threading

La generación de los 120 frames se dividió en 4 hilos, cada uno procesando un lote de frames. Para evitar condiciones de carrera, se utilizó un `Lock` al escribir en la lista compartida `frames`:

```
def procesar_lote(inicio, fin):
    for i in range(inicio, fin):
        # Genera el frame y actualiza la posición del dinosaurio
        with lock:
            frames[i] = escenario  # Guardar el frame procesado
```

7.4. Integración del fondo y objetos

El fondo se cargó desde una imagen (`imagen.jpg`) usando Pillow y se redimensionó. Los objetos se insertaron directamente en la matriz del escenario mediante la función `colocar_objeto`, que reemplaza píxeles del fondo con los del objeto:

```
def colocar_objeto(imagen, matriz, pos_y, pos_x, color):
    for y in range(matriz.shape[0]):
        for x in range(matriz.shape[1]):
            if matriz[y, x] > 0:
                imagen[pos_y + y, pos_x + x] = color
    return imagen
```

7.5. Visualización con Matplotlib

La animación se mostró utilizando `FuncAnimation`, que actualiza el gráfico en cada frame:

```
ani = animation.FuncAnimation(
    fig, actualizar, frames=NUM_FRAMES, interval=100, blit=True
)
plt.show()
```

7.6. Notas técnicas

- **Optimización**: La paralelización redujo el tiempo de generación de frames al distribuir la carga entre hilos.
- **Sincronización**: El uso de `Lock` garantizó que no hubiera conflictos al escribir en la lista `frames`.
- **Eficiencia**: NumPy permitió operaciones rápidas sobre matrices, mientras que Pillow y Matplotlib facilitaron la manipulación y visualización.

8. RESULTADOS

Durante la ejecución del programa se lograron generar los 120 cuadros de la animación sin errores, aprovechando el uso de múltiples hilos para repartir el trabajo. Cada hilo se encargó de una parte de los frames, lo que aceleró bastante el proceso y permitió una animación fluida y continua, sin retrasos visibles.

Para evitar problemas mientras los hilos trabajaban al mismo tiempo, se utilizó un mecanismo de bloqueo que aseguró que cada hilo escriba su parte sin interferencias. Esto ayudó a mantener el orden en la creación de los cuadros y evitó que se pierda información o se duplique por accidente.

En cuanto al resultado visual, el dinosaurio se desplaza cuadro por cuadro dentro del escenario, que se mantiene constante con elementos como nubes y piedras. Este cambio de posición se logra modificando la matriz general en cada frame, lo que crea el efecto de movimiento sin necesidad de complicar la estructura del programa.

Figura 3: Ejecucion del programa

9. CONCLUSIONES

- Se logró implementar un sistema de paralelización de píxeles en Python utilizando threading, cumpliendo con el objetivo general del proyecto.
- La división del trabajo en múltiples hilos permitió mejorar la eficiencia en la generación de frames, cumpliendo con el objetivo específico de optimización temporal.
- Se representaron objetos utilizando matrices de NumPy, lo cual facilitó su manipulación y mezcla visual, cumpliendo con otro de los objetivos específicos.
- El uso de threading.Lock evitó conflictos de concurrencia, asegurando la integridad de los datos, cumpliendo así el objetivo de sincronización.
- La visualización fue exitosa gracias a Matplotlib, permitiendo mostrar los resultados de manera clara y profesional.

10. REFERENCIAS

- Bitácora del Programador. (2023). Entendiendo el GIL de Python: Por qué es importante para la concurrencia. Recuperado de https://bitacoradelprogramador.com/entendiendo-el-gil-de-python-por-que-es-importante-para-la-concurrencia/
- Data Science para Todos. (2021). Cómo crear animaciones con Matplotlib en Python. Recuperado de https://datascienceparatodos.com/animaciones-matplotlib-python/
- Datascientist. (2020). NumPy: La Biblioteca Fundamental para la Computación Numérica en Python. Recuperado de https://www.datascientist.es/numpy-la-biblioteca-fundamental-para-la-computacion-numerica-en-python/

- Martínez, A. (2022). Manipulación de imágenes con Pillow en Python. Recuperado de https://adalidmartinez.com/manipulacion-de-imagenes-con-pillow-en-python/
- Programación Fácil. (2021). Concurrencia en Python: Hilos y Procesos. Recuperado de https://www.programacionfacil.io/blog/concurrencia-python-hilos-procesos/
- Programación Gráfica. (2018). Alpha Blending: Combinando Imágenes con Transparencia. Recuperado de https://programaciongrafica.com/alpha-blending-combinando-imagenes-con-transparencia/
- Pyladies Madrid. (2019). Mecanismos de Sincronización en Python: Locks, Semáforos y Eventos. Recuperado de https://pyladies.com/blog/mecanismos-sincronizacion-python/
- Real Python Team. (2024). An Intro to Threading in Python. Recuperado de https://realpython.com/intro-to-python-threading/
- NumPy Developers. (2024). NumPy Documentation. Recuperado de https://numpy.org/doc
- PIL Developers. (2024). Python Imaging Library (Pillow). Recuperado de https://pillow.readthedocs.io

ANEXOS

1. Código Completo

```
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import numpy as np
import threading
import random
from PIL import Image

NUM_FRAMES = 120
ANCHO_ESCENARIO = 100
ALTO_ESCENARIO = 40
NUM_HILOS = 4

frames = [np.zeros((ALTO_ESCENARIO, ANCHO_ESCENARIO, 3), dtype=np)
    for _ in range(NUM_FRAMES)]
lock = threading.Lock()

OBJETOS = {
    'nube': {
        'matriz': np.array([
            [0,0,0,0,0,0,0,0,0,0,0,0],
            [0,0,0,0,1,1,1,1,0,0,0,0],
            [0,0,1,1,1,1,1,1,1,1,0,0],
            [0,1,1,1,1,1,1,1,1,1,1,0],
            [1,1,1,1,1,1,1,1,1,1,1,1]
        ]),
        'color': [222, 255, 251]
    },
    'piedra': {
        'matriz': np.array([
            [0,0,0,0,0,0],
            [0,0,1,1,1,0],
            [1,1,1,1,1,1],
            [1,1,1,1,1,1]
        ]),
        'color': [100, 100, 100]
    },
}

fondo_imagen = Image.open("imagen.jpg").resize((ANCHO_ESCENARIO,
    ALTO_ESCENARIO))
FONDO_RGB = np.array(fondo_imagen)

def colocar_objeto(imagen, matriz, pos_y, pos_x, color):
    alto, ancho = matriz.shape
    for y in range(alto):
        for x in range(ancho):
```

```
            if matriz[y, x] > 0:
                if 0 <= pos_y + y < imagen.shape[0] and 0 <= pos_x + x < imagen.shape[1]:
                    imagen[pos_y + y, pos_x + x] = color
    return imagen

def crear_escenario_fondo():
    fondo = np.copy(FONDO_RGB)
    for x in range(0, ANCHO_ESCENARIO, 15):
        tipo = random.choice(list(OBJETOS.keys()))
        obj = OBJETOS[tipo]
        pos_x = min(x, ANCHO_ESCENARIO - obj['matriz'].shape[1])
        pos_y = 0 if tipo == 'nube' else ALTO_ESCENARIO - obj['matriz'].shape[0] - 2
        fondo = colocar_objeto(fondo, obj['matriz'], pos_y, pos_x, obj['color'])
    return fondo

ESCENARIO_FIJO = crear_escenario_fondo()

def crear_dinosaurio(mover_izq, mover_der):
    dino = np.array([
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,1,1,1,0,0,0,0,0],
        [0,0,0,0,1,1,1,1,1,1,0,0,0,0],
        [0,0,0,1,1,1,1,1,0,1,1,1,0,0],
        [0,0,0,1,1,1,1,1,1,1,1,1,0,0],
        [0,0,0,0,1,1,1,1,1,1,0,0,0,0],
        [0,0,0,0,0,1,1,1,1,0,0,0,0,0],
        [0,1,0,0,0,1,1,1,1,0,0,0,0,0],
        [0,1,1,0,1,1,1,1,1,1,0,0,0,0],
        [0,1,1,0,1,1,1,1,1,1,1,1,0,0],
        [0,0,1,1,1,1,1,1,1,1,0,1,0,0],
        [0,0,1,1,1,1,1,1,1,1,0,0,0,0],
        [0,0,0,1,1,1,1,1,1,0,0,0,0,0],
        [0,0,0,0,1,1,1,1,0,0,0,0,0,0],
        [0,0,0,0,1,0,0,1,0,0,0,0,0,0],
        [0,0,0,0,1,1,0,1,1,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0]
    ])

    if mover_izq:
        dino[14:16, 4:6] = 0
    if mover_der:
        dino[14:16, 7:9] = 0
    return dino

def procesar_lote(inicio, fin):
    for i in range(inicio, fin):
        mover_izq = (i % 2 == 0)
```

```
        escenario = np.copy(ESCENARIO_FIJO)
        dino = crear_dinosaurio(mover_izq, not mover_izq)
        color = [85, 124, 0]
        pos_x = (i * 2) % (ANCHO_ESCENARIO - dino.shape[1])
        pos_y = ALTO_ESCENARIO - dino.shape[0] - 1
        escenario = colocar_objeto(escenario, dino, pos_y, pos_x,
    color)
        with lock:
            frames[i] = escenario


hilos = []
frame_por_hilo = NUM_FRAMES // NUM_HILOS

for i in range(NUM_HILOS):
    inicio = i * frame_por_hilo
    fin = (i + 1) * frame_por_hilo if i != NUM_HILOS - 1 else
    NUM_FRAMES
    hilo = threading.Thread(target=procesar_lote, args=(inicio, fin)
    )
    hilos.append(hilo)
    hilo.start()

for hilo in hilos:
    hilo.join()

fig = plt.figure(figsize=(10, 5))
img = plt.imshow(frames[0], animated=True)
plt.axis('off')

def actualizar(i):
    img.set_array(frames[i % NUM_FRAMES])
    return [img]

ani = animation.FuncAnimation(
    fig, actualizar, frames=NUM_FRAMES, interval=100, blit=True
)

plt.tight_layout()
plt.show()
```

Listing 1: Implementación principal (EscenarioFinal.py)

2. Repositorio del Proyecto

github.com/eduardo0121/Proyecto-Final_Paralelizacion-de-pixeles

- `/Sources`: Archivos fuente Python
- `/Img`: Representaciones visuales
- `/Docs`: Documentación técnica