GO GO UIA-BOT

Documentacion Tecnica

Diagrama

El contenido generado por IA puede ser incorrecto.

Juego 2D side-scroll donde un robot repartidor enfrenta misiones en una Smart City

[**Funcionalidades extra implementadas** 3](#_Toc209149114)

[**Sistema de juego avanzado** 3](#_Toc209149115)

[**Sistema visual avanzado** 3](#_Toc209149116)

[**Arquitectura de software robusta** 4](#_Toc209149117)

[**Sistema de audio** 4](#_Toc209149118)

[**Sistemas de gameplay balanceado** 4](#_Toc209149119)

[**Valor educativo agregado** 5](#_Toc209149120)

[**Prompts para desarrollo del juego** 5](#_Toc209149121)

[**Prompts de implementacion** 5](#_Toc209149122)

[**Prompts para la resolución de bugs** 6](#_Toc209149123)

[**Optimizaciones** 7](#_Toc209149124)

[**Object Pooling** 7](#_Toc209149125)

[**Background Rendering Optimization (prerender / cache)** 8](#_Toc209149126)

[**Cache de textos (HUD) con dirty flags** 9](#_Toc209149127)

# Funcionalidades extra implementadas

# **Sistema de juego avanzado**

**1. Sistema de Dash con Energía**

* **Qué es:** Habilidad especial que permite al jugador moverse rápido hacia adelante
* **Cómo funciona:** Presionar Z consume energía y da velocidad extra
* **Por qué es especial:** Agrega estrategia - ¿usar dash ahora o guardarlo?
* **Tiempo de cooldown:** Sistema que evita spam de la habilidad

**2. Sistema de Pausa Inteligente**

* **Funcionalidad:** Presionar P pausa completamente el juego
* **Características:** Overlay transparente, texto informativo, navegación con ESC
* **Valor educativo:** Permite a estudiantes tomarse breaks sin perder progreso

**3. Cámara Dinámica con Seguimiento Suave**

* **Implementación:** Cámara que sigue al jugador con interpolación suave
* **Efecto:** Movimiento cinematográfico, no abrupto
* **Beneficio:** Experiencia visual profesional

## **Sistema visual avanzado**

**4. Efecto Parallax de 3 Capas**

* **Capas:** Fondo estatico, capa media, capa frontal
* **Velocidades diferentes:** Crea profundidad e ilusión de velocidad
* **Superando consigna:** El PDF solo pedía "animar fondo"

**5. Spritesheets Animados Completos**

* **Jugador:** 5 frames de animación de caminar
* **Autos:** 3 frames de animación por vehículo
* **Sistema robusto:** Maneja sprites faltantes con placeholders

**6. Efectos Visuales del Dash**

* **Estela visual:** Trail effect cuando se usa dash
* **Feedback inmediato:** El jugador ve claramente cuándo está usando dash
* **Polish profesional:** Detalles que hacen la diferencia

## **Arquitectura de software robusta**

**7. Sistema de Escenas Completo**

* **Arquitectura:** Scene Manager que maneja transiciones entre pantallas
* **Escalabilidad:** Fácil agregar nuevas pantallas sin tocar código existente
* **Profesional:** Patrón de diseño usado en juegos comerciales

**8. Resource Manager Completo**

* **Gestión automática:** Carga y libera recursos automáticamente
* **Spritesheets:** Sistema completo de manejo de hojas de sprites
* **Música/Sonidos:** Sistema completo con controles de volumen
* **Fallbacks:** Placeholders automáticos si faltan archivos

**9. Sistema de Configuración Avanzado**

* **Pantalla de Settings:** Control de música, acceso a controles
* **Pantalla de Controles:** Tutorial visual de todos los comandos
* **Navegación completa:** ESC funciona consistentemente en todo el juego

## **Sistema de audio**

**10. Música Dinámica por Pantalla**

* **Menú:** Música de ambiente relajada
* **Juego:** Música de acción energética
* **Transiciones:** Cambios suaves entre temas

**11. Efectos de Sonido Reactivos**

* **Hover en botones:** Feedback audio inmediato
* **Acciones del jugador:** Sonidos para salto, dash
* **Estados del juego:** Game over y victoria tienen audio especial

## **Sistemas de gameplay balanceado**

**12. Spawn Inteligente de Enemigos**

* **Distancias seguras:** Los autos nunca aparecen injustamente cerca
* **Dificultad progresiva:** Sistema que mantiene el desafío equilibrado
* **Cleanup automático:** Memoria optimizada eliminando autos lejanos

## **Valor educativo agregado**

**13. Feedback Visual Constante**

* **Barras de progreso:** Energía con colores semafóricos
* **Contadores claros:** Kilómetros restantes siempre visibles
* **Estados evidentes:** Jugador siempre sabe qué está pasando

**14. Arquitectura Educativa**

* **Código limpio:** Comentarios claros para que estudiantes entiendan
* **Modularidad:** Cada sistema es independiente (fácil de estudiar/modificar)
* **Constantes centralizadas:** Fácil modificar dificultad sin tocar lógica

## **Sistema de Misiones**

El juego cuenta con un **sistema de misiones** diseñado para darle progresión y contexto narrativo a la jugabilidad.

**Objetivo del sistema**

* Darle al jugador un **propósito concreto** en cada nivel
* Introducir **narrativa ligera** mediante diálogos entre el protagonista (UIABOT) y personajes secundarios.
* Permitir que cada misión tenga un **tono distinto** (humor, sarcasmo, aventura) sin modificar la jugabilidad central.

**Estructura técnica**

* Cada misión está definida como una **escena independiente** que contiene:
  + **Información del remitente y destinatario del paquete.**
  + **Diálogos predefinidos** en formato JSON adentro de un array o estructura de lista de diccionarios (con atributos: speaker, text, position, emotion).

El sistema permite **agregar nuevas misiones fácilmente**: es tan fácil como incluir un nuevo array de configuración con diálogos y parámetros, sin cambiar la lógica central.

## **Modo Infinito**

Pensado como un **desafío arcade** para quienes quieran jugar sin límites.

* El jugador controla a UIABOT en un entorno de tráfico infinito
  + Las plataformas y power-ups se distribuyen de forma coherente según las reglas de salto/dash.
* El objetivo es sobrevivir el mayor tiempo posible, acumulando **kilómetros recorridos** como puntaje.

# **Prompts para desarrollo del juego**

## **Prompts de implementacion**

**Creación del sistema de pantallas de controles:**

"Necesito crear una pantalla en mi juego de pygame que muestre los controles del juego.

Debe tener:

- Fondo animado igual que el menú

- Mostrar controles como [ESPACIO] → Saltar, [Z] → Dash, etc.

- Botón para volver a settings

- Diseño consistente con el resto del juego"

**Mejora del sistema de animaciones:**

"Tengo un juego en pygame con spritesheets. Necesito mejorar el sistema de animaciones

para que sea más fluido. El jugador debe tener animación de caminar y los autos

también deben estar animados."

**Optimización del sistema de colisiones:**

"Mi juego tiene colisiones entre jugador y autos. Necesito hacer el sistema más

justo, con hitboxes más pequeñas y que no sea tan difícil esquivar los autos."

**Mejora de efectos visuales:**

"Quiero agregar efectos visuales a mi juego:

Estela de dash cuando el jugador usa la habilidad especial

Efectos de hover en botones con sonido

Transiciones suaves entre pantallas"

**cProfile**

“Estoy haciendo un juego en Pygame y quiero medir bien el rendimiento. Me gustaría integrar cProfile para ver en qué funciones se va más tiempo (por ejemplo draw\_background, render de texto, colisiones, etc). La idea es:

* poder correr el juego con cProfile activado,
* guardar los resultados en un archivo .prof o .txt para revisarlos después,

**Arquitectura**

“Estoy organizando la arquitectura del juego y me surgió la duda de si conviene seguir con un enfoque que sea más procedural (con módulos sueltos y funciones globales para cada sistema) o dar el salto a una arquitectura más orientada a objetos, con clases específicas para cada responsabilidad.”

## **Prompts para la resolución de bugs**

**Corrección del sistema de spawn de autos:**

"Los autos en mi juego aparecen muy cerca del jugador o demasiado juntos.

Necesito un sistema que mantenga distancia segura entre autos y del jugador."

**Optimización de rendimiento**

"Estoy usando muchas llamadas a pygame.font.Font.render. ¿Como puedo implementar un sistema de cache de textos para el HUD y evitar que el renderizado se haga múltiples veces?"

"quiero mejorar la performance de mis entidades. ¿como implemento un sistema de object pooling para reusar autos y no crear/desechar objetos todo el tiempo?"

# **Optimizaciones**

Implementamos varias optimizaciones clave para mejorar la performance del juego y asegurar una experiencia estable Las mejoras principales son:

1. **Object pooling** para Car y para Pila (reutilización de objetos).
2. **Optimización del render de fondo** (pre-render y cache).
3. **Cache de textos (HUD)** con dirty flags para evitar font.render() por frame.
4. **Optimización del draw\_background()** para reducir blits innecesarios.

Cada punto explica la motivación técnica, la implementación, el impacto esperado/medible, riesgos y cómo probarlo.

## **Object Pooling**

***¿Qué es?***

**Object pooling:** Consiste en crear un **conjunto (pool)** de instancias reutilizables al inicio y, en vez de crear y destruir objetos todo el tiempo, tomar uno libre del pool, reinicializarlo y volverlo a usar. Cuando termina su uso, se devuelve al pool para reuse.

Es una técnica ampliamente usada en la industria de videojuegos.

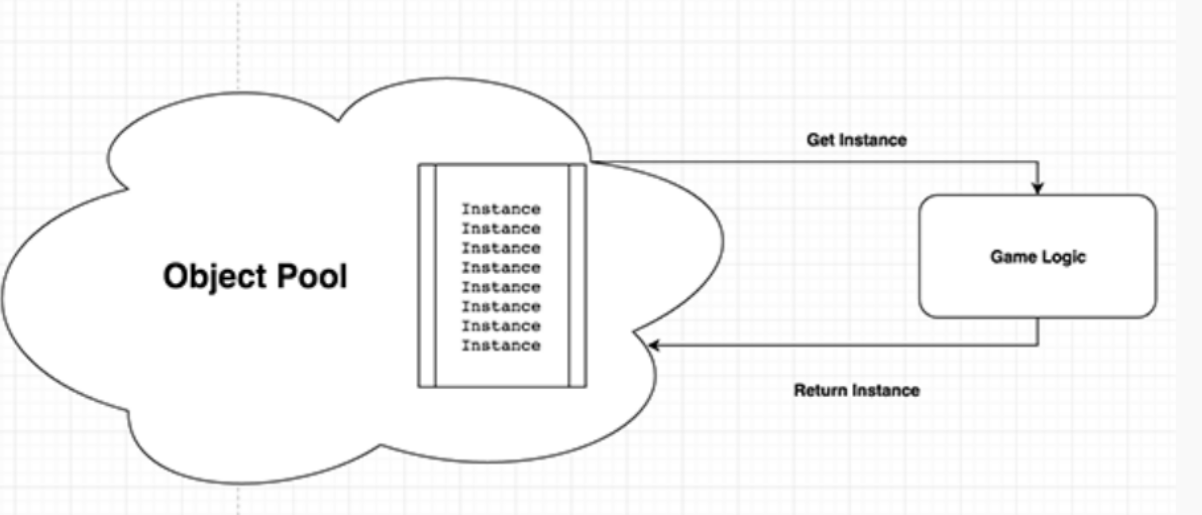


Ilustración 1

*Como podemos ver en la* Ilustración 1*: el* ***Object Pool*** *preinstancia un conjunto de objetos, la* ***Logica del juego*** *solicita uno libre con* ***Get Instance*** *y, al terminar su uso, lo devuelve con* ***Return Instance*** *para reinicializarlo y reutilizarlo sin nuevas asignaciones.*

***Por qué lo hicimos***

* Crear y destruir muchos objetos en tiempo de juego provoca overhead de memoria y CPU (construcción de objetos, inicializaciones, llamadas a ***\_\_init\_\_,*** y eventualmente colector de basura).
* En escenas con muchos spawns (autos, pilas), ese overhead genera micro-stutter y picos de latencia justo cuando aparecen objetos.

***Cómo lo implementamos***

Creamos **CarPool** y **PilaPool** con:

* **available:** Lista de objetos libres (esperando para usarse)
* **active:** Lista de objetos en uso
* **max\_pool\_size:** Límite total de instancias
* métodos clave:
* **get\_car(x, y, speed**): Devuelve un auto listo para usar
* **return\_car(car):** Lo devuelve al pool cuando sale de pantalla

En las clases Car y Pila agregamos un método **reset\_for\_reuse(...)** que reinicia posición, velocidad y flags (active=True, visible=True).

***Impacto medible***

* Reduce llamadas a **Car.\_\_init\_\_** y a pygame.image.load (si se cargan bien en recursos globales).
* Menos garbage / menores pausas de GC.
* Mejora del **framerate** en escenas con muchos spawns (medible con profiler).

***Riesgos y mitigaciones***

* Si el pool es demasiado pequeño, podes “perder” spawns. Mitigar: ajustar max o estrategia para retrasar spawns.
* Los objetos deben reinicializar todo el estado visible; si te olvidás algo (ej. timers internos), aparecen bugs. Mitigar: reset\_for\_reuse() debe reinicializar explícitamente todo.

## **Background Rendering Optimization (prerender / cache)**

***Problema***

Estabamos redibujando y recalculando capas de fondo cada frame, con operaciones repetidas (escalados, composiciones, efectos), lo que genera muchos blits/cálculos.

**Solución aplicada**

* Prerender cada capa pesada en Surface al cargar el recurso (o cuando cambia la resolución/bioma).
* Mantener versiones escaladas/recortadas si hace falta (ej. para múltiples resoluciones).
* En el loop sólo se hace: **screen.blit(pre\_rendered\_layer, (offset\_x, 0))** el cálculo de offset se hace pero la superficie ya está compuesta y optimizada (convert()/convert\_alpha() aplicado al cargar).
* Si hay animación en la capa (p. ej. carteles parpadeantes), se renderiza sólo la porción animada en un Surface pequeño y se blitea encima de la capa pre-render.

**Impacto**

* Baja drástica del tiempo de dibujo por capa.
* Menos operaciones **pygame.transform.scale** en runtime.

**Riesgos**

* Uso de memoria extra (almacenar varias superficies). Compensar calculando la mejor resolución intermedia y liberando versiones no usadas.

## **Cache de textos (HUD) con dirty flags**

**Problema**

**font.render()** es caro y estaba ocurriendo decenas de miles de veces (28k). Eso sale en el profiler y en el impacto de la CPU.

**Solución aplicada**

Mantener un diccionario text\_cache donde la key sea (label\_name, value) o un id lógico; el value es la Surface ya renderizada.

Cada variable que aparece en pantalla (ej: energía, km, score) tiene un **flag dirty** que se pone True cuando el valor cambia. En el draw():

* **Si** dirty: recalcula **font.render()** → guarda en cache y marca **dirty=False**.
* **Si** not dirty: usa **text\_cache[key]** y blitea la surface.

Para textos que cambian continuamente (ej: contador de km por cada frame), se actualiza la caché sólo cuando cambia el valor redondeado (por ejemplo centésimas).

**Impacto**

* Dejamos de ver las miles de llamadas a font.render en el profiler.
* CPU liberada

**Riesgos**

* Si no marcáramos dirty correctamente, el HUD mostrará valores desactualizados. Mitigar: centralizar la actualización de variables y marcar dirty ahí.

**Renderizado del Background**

**Problema**

Función hacía blits innecesarios (ej. dibujaba capas completas aún si no salían en pantalla o si no cambiaron).

**Qué hicimos**

* Lógica para sólo blitear las regiones visibles (dos copias por capa para loop).
* Evitar redibujar capas si la cámara no se movió (delta\_x cercano a 0).
* Separar capas estáticas (prerender) de capas dinámicas pequeñas.

**Impacto**

* Menos blits por frame.
* El perfil mostrará menor tiempo en game\_renderer.draw\_background y menor número total de blit calls.