



UNIVERSIDAD DE TALCA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA EN DESARROLLO DE
VIDEOJUEGOS Y REALIDAD VIRTUAL

Generación de Exteriores y Poblaciones en Tiempo de Ejecución para Level Building Sidekick

DARKO MUÑOZ REBOLLEDO

Profesor Guía: NICOLÁS BARRIGA

Profesor Co-guía: NICOLÁS ROMERO

Memoria para optar al título de
Ingeniero en Desarrollo de Videojuegos y Realidad Virtual

Talca – Chile
Julio, 2026



UNIVERSIDAD DE TALCA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA EN DESARROLLO DE
VIDEOJUEGOS Y REALIDAD VIRTUAL

Generación de Exteriores y Poblaciones en
Tiempo de Ejecución para Level Building Sidekick

DARKO MUÑOZ REBOLLEDO

Profesor Guía: NICOLÁS BARRIGA

Profesor Co-guía: NICOLÁS ROMERO

Profesor Informante: FELIPE BESOAIN

Memoria para optar al título de
Ingeniero en Desarrollo de Videojuegos y Realidad Virtual

El presente documento fue calificado con nota: _____

Talca – Chile

Julio, 2026

TABLA DE CONTENIDOS

página

Tabla de Contenidos	I
Resumen	III
1. Introducción	1
1.1. Estado Actual de LBS	2
1.2. Problemática	2
1.3. Propuesta	3
1.4. Objetivos y Pregunta de Investigación	4
2. Marco Teórico	5
2.1. Generación Procedural de Contenido	5
2.2. PCG de Iniciativa Mixta	6
2.3. Algoritmos de PCG en LBS	7
2.4. Motores de Videojuegos	9
3. Estado de Arte	10
3.1. Aplicaciones de MI-PCG	10
3.2. Generación de Terreno	11
3.3. Generación de Población	12
3.4. Aplicaciones PCG Online	14
4. Métodos	16
4.1. Arquitectura propuesta	16
4.2. Adaptación del módulo de generación de exteriores (Paso 1B)	18
4.2.1. Arquitectura interna	18
4.2.2. Flujo de Ejecución	19
4.3. Adaptación del módulo de generación de población (Paso 2)	20
4.3.1. Arquitectura Interna	20
4.3.2. Flujo de Ejecución	21
4.4. Consideraciones sobre la ejecución asíncronica	22
4.5. Flujo de generación de un chunk	23

5. Experimentos	25
5.1. Diseño Experimental	25
5.2. Configuración del entorno	26
5.3. Escenario de Pruebas	27
5.4. Experimento 1: Pruebas de tiempo por tamaño de Chunk	28
5.4.1. Procedimiento	28
5.4.2. Variables Medidas	28
5.5. Experimento 2: Comparación entre la implementación original de LBS y Runtime LBS	29
5.5.1. Procedimiento	29
5.5.2. Variables Medidas	30
6. Resultados	31
6.1. Resultados del Experimento 1	31
6.1.1. Tiempos de Generación	31
6.1.2. Comportamiento durante la generación continua	32
6.1.3. Rendimiento general del sistema	33
6.2. Resultados del Experimento 2	34
6.2.1. Comparación entre LBS y Runtime LBS	34
7. Conclusiones	36
7.1. Discusión de resultados	36
7.2. Principales Aportes	37
7.3. Limitaciones y Trabajo Futuro	38
Glosario	40
Bibliografía	42
A: Capturas	47
B: Pseudocódigo	49
C: Diagramas	50
D: Gráficos y figuras	52

RESUMEN

La generación procedural de contenido (PCG) se ha consolidado como un recurso clave para la creación de mundos virtuales dinámicos y adaptativos en videojuegos. Entre las herramientas que utilizan este enfoque se encuentra Level Building Sidekick (LBS), una herramienta de Generación Procedural de Contenidos de iniciativa mixta (MI-PCG), creada como un editor de niveles de videojuegos asistido por IA para el motor de videojuegos Unity. Sin embargo, LBS fue concebido para operar únicamente dentro del editor, impidiendo su utilización durante la ejecución del juego.

Esta memoria propone una adaptación de LBS que permite generar niveles en tiempo de ejecución mediante modificaciones en las implementaciones de sus métodos de PCG, con el fin de expandir su utilidad. Para ello, la investigación siguió la metodología Design Research in Information Systems, abarcando el diseño, desarrollo y evaluación iterativa de la solución. Como resultado, se eliminaron las dependencias del editor, se incorporó un modelo de ejecución basado en tareas asíncronas y se reorganizó el proceso de generación e instanciación para permitir una ejecución continua sin bloquear el hilo principal.

Los experimentos realizados evaluaron el rendimiento de la propuesta mediante la generación de chunks de distintos tamaños y una comparación directa con la implementación original de LBS. Los resultados de la comparativa muestran una reducción del tiempo de generación de un escenario de 50×50 celdas, de un tiempo de 64,6 segundos de la herramienta original hacia 3,1 segundos con la implementación, además de mantener un comportamiento estable en el uso de CPU, GPU y memoria durante la generación continua.

Las modificaciones propuestas permiten ejecutar los componentes de LBS durante el gameplay con un rendimiento estable, ampliando las capacidades originales de la herramienta y demostrando la viabilidad de adaptar herramientas de PCG diseñadas para el editor hacia un modelo de generación en tiempo de ejecución.

1. Introducción

La generación procedural de contenidos (PCG) [1] se ha convertido en una de las estrategias más importantes dentro del desarrollo de mundos, historias y personajes en varios géneros de videojuegos [2], mediante algoritmos capaces de producir resultados variados a partir de parámetros definidos por el desarrollador. La PCG ha sido utilizada en una amplia gama de géneros, desde juegos de rol hasta de exploración [3], transformándose en un componente clave dentro del diseño de entornos interactivos.

En este contexto de creciente adopción de la PCG dentro del desarrollo de videojuegos, Level Building Sidekick (LBS)[4] surge como una herramienta para el motor de videojuegos Unity ¹ orientada a apoyar el diseño de niveles mediante un enfoque de PCG de iniciativa mixta (MI-PCG) [5], la cual combina la intervención directa del diseñador con sistemas automáticos de generación que actúan como asistentes creativos.

LBS está dividida en diferentes módulos de trabajo. Para este caso, se considerará los módulos 1B y 2. El módulo 1B está orientado a generar terrenos ambientales extensos, utilizando el algoritmo de WFC [6]. El módulo 2 se orienta a poblar estos terrenos de forma coherente en base al algoritmos de MAP-Elites [7] y parámetros dados por el desarrollador. Su propósito es facilitar la construcción de entornos que incluyan elementos como *NPCs* y objetos distribuidos dentro del espacio generado.

LBS fue concebido como un sistema de apoyo dentro del entorno de desarrollo, y no como una herramienta capaz de generar contenido en ejecución del juego. Esta orientación se explica por su contexto. La primera iteración que luego dió origen

¹Unity Technologies. Unity Real-Time Development Platform. Disponible en: <https://docs.unity.com/en-us>

a LBS, la cual aplicaba MI-PCG y algoritmos evolutivos para generar biomas, se llamaba *Biome Generation Tool* [8], un proyecto de un memorista anterior. A partir de este, se propuso expandir en esta idea, capaz de construir terrenos y escenarios aplicables a diferentes tipos de videojuegos, dando origen a LBS.

Al tener un enfoque como herramienta asistente en el proceso de diseño y desarrollo, se restringe su potencial frente a los avances actuales de la PCG, donde la generación dinámica, adaptativa y en tiempo real está cobrando cada vez mayor relevancia.

1.1. Estado Actual de LBS

LBS ha evolucionado hasta convertirse en una herramienta funcional dentro del ecosistema de desarrollo de Unity. Su enfoque modular aprovecha el potencial de la PCG. No obstante, sus usos solo se extienden a la fase de desarrollo, sin intervenir en la generación de contenido durante la ejecución del juego, haciendo que su potencial como sistema adaptativo y dinámico aún no haya sido completamente aprovechado.

1.2. Problemática

A pesar de su potencial, LBS presenta una restricción fundamental que limita su aplicabilidad en escenarios más complejos. Su arquitectura actual fue diseñada exclusivamente para operar dentro del editor de Unity, lo que impide que los procesos de PCG se ejecuten durante el tiempo de ejecución. Esta limitación imposibilita la creación de mundos que evolucionen en respuesta al jugador, o que se adapten a condiciones cambiantes del entorno, como ocurre en los sistemas de generación adaptativa. [9]

Esto contrasta con las tendencias actuales del desarrollo de videojuegos, donde la generación procedural en tiempo real (denominado *online* PCG [10]) permitieron crear experiencias interactivas más ricas y personalizadas, donde el mundo responde a las acciones del jugador como en títulos de exploración abierta, así como *No Man's Sky* [11], generando proceduralmente universos enteros hasta su más mínimo detalle, o *Minecraft* [12], con sus mundos dinámicos creados a partir del uso de algoritmos y mapas de ruido que crean terrenos por partes. Con esto, *Online PCG* se ha convertido en un área de creciente interés.

1.3. Propuesta

Frente a esta limitación, la presente memoria propone ampliar las capacidades de LBS mediante el desarrollo de una versión capaz de generar contenido de forma *Online*, o durante la ejecución del juego.

Se busca adaptar los módulos existentes (módulos 1B y 2 mencionados previamente) para que puedan funcionar fuera del entorno de edición, integrándose directamente al flujo del gameplay. Con ello, la herramienta pasaría de ser un asistente de diseño a un sistema capaz de generar y modificar entornos de manera dinámica.

El propósito de esta propuesta es extender el uso de LBS hacia contextos de PCG dentro de Unity, ofreciendo una forma distinta de crear niveles para escenarios de exploración y mundo abierto.

Se plantea una evaluación del rendimiento de la herramienta durante su ejecución en tiempo real, centrada en estabilidad y fluidez. Esta evaluación medirá la velocidad de generación durante el tiempo de ejecución, y su nivel de impacto en la CPU, registrado mediante el uso de herramientas de monitoreo de rendimiento del sistema.

Ya que LBS estará disponible en la Unity Asset Store, implementar esta propuesta puede aumentar el interés de programadores de videojuegos en general, al igual que para estudiantes de la Universidad de Talca que deseen crear sus proyectos de videojuegos con esta herramienta.

1.4. Objetivos y Pregunta de Investigación

Objetivo general: Desarrollar una extensión de LBS que permita ampliar los contextos en los que se pueda utilizar la herramienta en la creación de videojuegos, habilitando la ejecución de los módulos en tiempo de ejecución (runtime).

Objetivo específico 1: Investigar sobre técnicas de optimización de PCG en tiempo real en motores de videojuegos.

Objetivo específico 2: Diseñar la arquitectura y el flujo de trabajo del nuevo apartado de LBS.

Objetivo específico 3: Adaptar dos módulos de LBS desde editor a runtime.

Objetivo específico 4: Aplicar optimizaciones investigadas a los dos módulos de LBS anteriormente adaptados.

Objetivo específico 5: Evaluar el rendimiento de uso de la herramienta mediante métricas cuantitativas de velocidad de generación y consumo de recursos.

Pregunta de investigación: ¿Qué modificaciones son necesarias de realizar en las implementaciones de WFC y los algoritmos genéticos utilizados en MAP-Elites dentro de la herramienta de LBS para poder ejecutarlos en tiempo de ejecución con un rendimiento estable?

Para los siguientes capítulos de este documento se profundizará sobre el Marco Teórico, el Estado del Arte, como se ha abordado la problemática en otras aplicaciones o herramientas, los métodos para la experimentación, sus resultados y una discusión al respecto.

2. Marco Teórico

En el presente capítulo, se establecen los conceptos fundamentales necesarios para comprender los principios y desafíos asociados a la generación procedural de contenidos en videojuegos, particularmente en el contexto de técnicas que requieren operar en tiempo real dentro de motores de videojuegos como Unity.

2.1. Generación Procedural de Contenido

La Generación Procedural de Contenido (PCG) [1] se ha definido como la generación algorítmica de *game content* (como mapas, escenarios, música, reglas, personajes, etc. [13]) con una limitada o indirecta contribución humana [14]. Su principal ventaja radica en la capacidad de producir variedad, rejugabilidad y escalabilidad [14] sin requerir una creación manual exhaustiva.

La PCG puede tomar formas simples, como variaciones de ruido para generar terrenos, o más avanzadas, como sistemas adaptativos que generan misiones [13], diálogos [15], ecosistemas o estrategias enemigas. [16]

La PCG adaptada a terrenos se fundamenta en técnicas tradicionales como Perlin Noise [17], fractales [18], heightmaps [19] o combinaciones de erosión simulada [20]. Estas técnicas permiten modelar terrenos variados de forma controlada, otorgando a los diseñadores un conjunto de herramientas para construir mundos.

Los distintos usos de estas técnicas puede afectar el rendimiento en contextos de generación en tiempo de juego, si es que no se aplican optimizaciones específicas [21]. Por otra parte, métodos que simulan procesos físicos detallados, como erosión, suelen requerir cálculos adicionales que se vuelven costosos si se pretende ejecutar dichos algoritmos sin algún tipo de optimización, como el paralelismo, en tiempo de juego [22].

Independiente de la técnica de PCG utilizada, los algoritmos requieren trabajar sobre unidades básicas de contenido que puedan combinarse para construir escenarios más complejos. Estas unidades, conocidas como *game bits*, corresponden a elementos que generalmente no afectan directamente al gameplay [23], como texturas, vegetación, sonidos o efectos visuales, y constituyen los bloques fundamentales sobre los que operan los algoritmos de generación procedural.

El uso de *game bits* permite desacoplar la generación del contenido final de sus componentes individuales, favoreciendo la reutilización, la variabilidad y el control del diseño. Por ello, son ampliamente utilizados en sistemas de PCG basados en reglas, gramáticas o algoritmos de búsqueda, donde pueden combinarse y adaptarse según las restricciones del sistema o las decisiones del diseñador.

Por otra parte, en PCG se distingue entre generación *online* y *offline* [10] según el momento en que se ejecutan los algoritmos. La PCG *online* se ejecuta durante la ejecución del juego, lo que impone limitaciones de rendimiento para evitar interrupciones en la experiencia del jugador. Este enfoque habilita la generación dinámica de contenido, pero requiere técnicas de optimización como paralelismo para asegurar una operación eficiente en tiempo real.

La PCG *offline* corresponde a procesos realizados fuera del tiempo de juego, comúnmente durante la etapa de desarrollo, donde no existen restricciones estrictas de tiempo de ejecución. Este enfoque permite aplicar algoritmos complejos y realizar múltiples iteraciones, generando contenido que luego es utilizado de forma estática durante el gameplay.

2.2. PCG de Iniciativa Mixta

La Generación Procedural de Contenidos de Iniciativa Mixta (MI-PCG) es un enfoque dentro del PCG que combina la creatividad humana con la capacidad computacional para generar contenido de manera colaborativa, proveyendo de control al diseñador para crear lo que ha visionado. [24]

A diferencia de PCG, MI-PCG propone un paradigma intermedio. En este modelo, el diseñador y el algoritmo trabajan juntos, alternando la iniciativa de creación. El sistema genera propuestas, variaciones o soluciones parciales, mientras que el diseñador selecciona, ajusta o guía el proceso según sus necesidades creativas.

LBS adopta MI-PCG como eje central de su funcionamiento, ya que fue conce-

bido para asistir al diseñador durante la creación de niveles, ofreciendo alternativas generadas por algoritmos evolutivos y de búsqueda que pueden ser aceptadas, refinadas o descartadas según la preferencia del usuario. LBS utiliza MAP-Elites en el módulo encargado de la población del nivel, para generar múltiples configuraciones posibles de distribución de NPCs y objetos. Estas variantes se muestran al usuario como opciones útiles dentro de un proceso de co-creación.

2.3. Algoritmos de PCG en LBS

Dentro de LBS, se emplean distintos algoritmos de PCG para sus módulos. Para este apartado, se enfocarán los utilizados en los módulos 1B y 2. En el módulo 1B de LBS, se utiliza *Wave Function Collapse* (WFC) [6].

WFC es un algoritmo de generación basada en restricciones que combina ideas provenientes de propagación de restricciones y principios de superposición inspirados en la mecánica cuántica. El sistema trabaja inicialmente con un estado de incertidumbre, donde cada celda de un mapa puede adoptar múltiples posibles patrones, reducidos progresivamente a través del colapso guiado por restricciones de compatibilidad entre patrones vecinos.

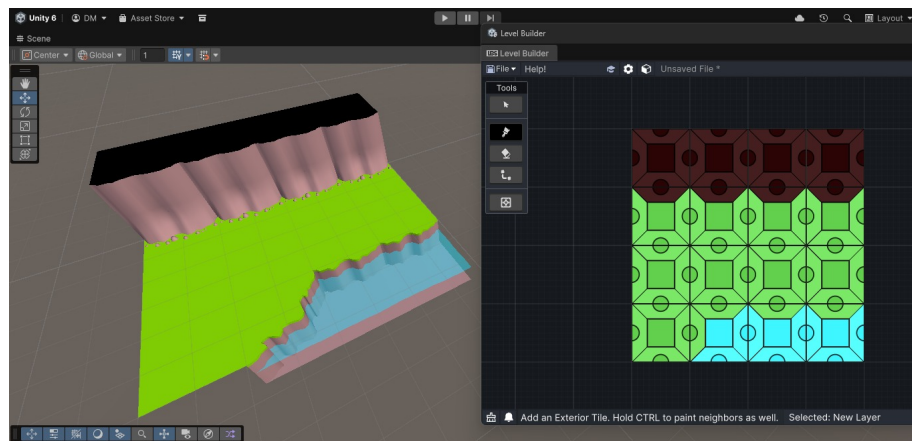


Figura 2.1: Ejemplo de ejecución de WFC en LBS.

En la figura 2.1 se puede ver el módulo 1B de generación de terrenos. El sistema genera la solución a nivel de celdas en la interfaz de diseño (apartado derecho), la cual es posteriormente instanciada como mapa dentro del motor Unity (apartado izquierdo).

La aplicación de WFC en LBS puede ser optimizada para que pueda correr en paralelo con la ejecución del juego, modularizando sus funciones e ingresando parámetros iniciales. El único coste que tendría sería la *instanciación* de terreno en Unity.

En el módulo 2, se emplea *MAP-Elites* [7]. Este es un algoritmo de búsqueda evolutiva orientado a la diversidad y la calidad, cuyo objetivo principal es poblar un espacio multi-dimensional de comportamientos con una colección diversa de soluciones representativas y eficientes.

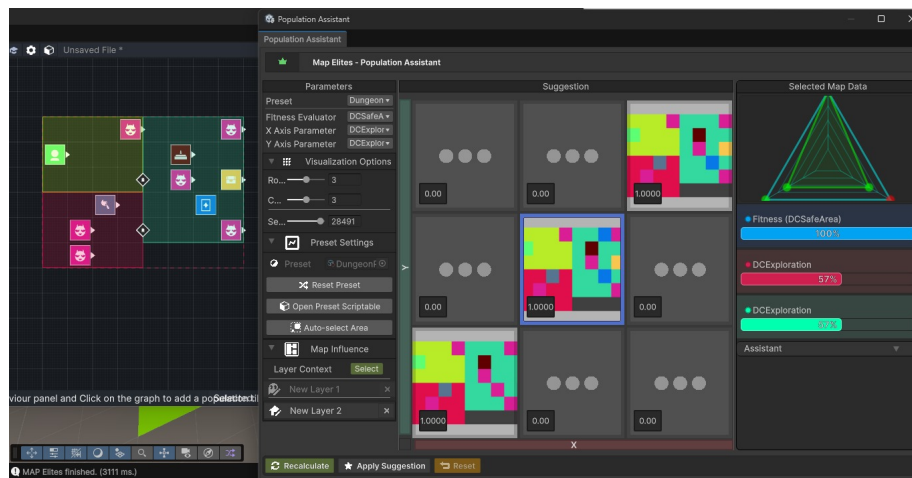


Figura 2.2: Ejemplo de ejecución de *MAP-Elites* en LBS.

Se puede apreciar en la figura 2.2 la interfaz de LBS en el lado izquierdo y MAP-Elites en el lado derecho. Se crean distintas variantes de posiciones de los cuadros de la interfaz izquierda en base a parámetros dados.

MAP-Elites pertenece a la familia de los Algoritmos Evolutivos, un conjunto de métodos de búsqueda estocástica y optimización, procesos inspirados en la evolución natural [25]. Su objetivo es encontrar soluciones de alta calidad dentro de espacios de búsqueda extensos mediante operaciones como mutación, recombinación y selección. Este evalúa las posiciones de los objetos con el fin de llegar a un *fitness* [26] adecuado dependiendo de lo que quiera el usuario. Entre los algoritmos más representativos se encuentran los Algoritmos Genéticos [27], las Estrategias Evolutivas [28], la Programación Evolucionaria [29], entre otras.

Esta variedad de enfoques permite ajustar la estrategia de búsqueda según el nivel de exploración requerido, pero el algoritmo requiere evaluaciones repetidas, lo que implica un coste computacional elevado si se ejecuta de manera asíncrona.

2.4. Motores de Videojuegos

Los motores de videojuegos [30] son entornos de software diseñados para facilitar la creación, simulación y ejecución de experiencias interactivas. Proveen un conjunto integrado de herramientas y sistemas (como renderizado gráfico, física, audio, animación y gestión de recursos) que permiten a los desarrolladores concentrarse en el diseño y la lógica del juego sin tener que implementar desde cero los componentes técnicos fundamentales.

En el caso de LBS, este utiliza *Unity*, uno de los motores de videojuegos más utilizados a nivel mundial debido a su accesibilidad, flexibilidad y ecosistema maduro. *Unity* utiliza principalmente C# como lenguaje de programación, ofreciendo un entorno de trabajo basado en la manipulación de objetos dentro de una interfaz visual del tipo *WYSIWYG* (*What You See Is What You Get*) [31], donde los diseñadores pueden arrastrar, posicionar y configurar elementos directamente en la escena.

Dentro de *Unity*, los elementos centrales del juego se representan mediante *GameObjects* [32], estructuras que actúan como contenedores para distintos componentes que definen su comportamiento, apariencia y funcionalidades. Su organización fue implementada mediante un sistema basado en composición; un *GameObject* no hereda comportamientos de clases especializadas, sino que adquiere capacidades al agregar componentes como scripts, colliders o renderizadores.

Unity tiene mecanismos fundamentales para poder hacer generación procedural, uno de estos siendo la *Instanciación*. Este corresponde al proceso mediante el cual el motor crea una copia en tiempo de ejecución de estos *GameObjects* previamente definidos, usualmente un *Prefab* [33]. Un *Prefab* actúa como una plantilla que contiene información estructural del objeto, su jerarquía, componentes, parámetros y valores iniciales.

Al ejecutar el juego, el método `Instantiate()` permite generar nuevas instancias de dicho objeto dentro de la escena, lo que habilita la construcción dinámica de entornos, la creación de entidades del juego y la adaptación del mundo virtual según las necesidades del sistema.

Además de *Unity*, existen otros motores usados en la industria, como *Unreal Engine* [34], *Godot* [35], y *CryEngine* [36]. Cada motor ofrece modelos de programación, arquitecturas y flujos de trabajo distintos, pero todos comparten el objetivo de proporcionar herramientas robustas para la construcción eficiente de videojuegos.

3. Estado de Arte

La generación procedural de contenidos (PCG) ha sido ampliamente explorada a través de herramientas y sistemas diseñados para automatizar la creación de entornos virtuales complejos. En este capítulo, se mencionarán distintas herramientas con enfoque MI-PCG. Asimismo, se verá brevemente al respecto de herramientas que funcionen en tiempo de ejecución.

3.1. Aplicaciones de MI-PCG

Diversas herramientas han surgido para explorar el potencial de la generación procedural asistida mediante técnicas de iniciativa mixta, permitiendo una colaboración fluida entre diseñador y máquina.

Ejemplos destacados incluyen *Tanagra* [37], un generador de niveles de plataformas que combina planeación reactiva y resolución de restricciones para adaptar el contenido mientras el usuario lo edita, manteniendo coherencia estructural en tiempo real. De manera similar, *Sentient World* [38] genera configuraciones completas de mundos, presentando múltiples alternativas y evaluaciones automáticas que permiten al diseñador explorar rápidamente distintos estados posibles de un entorno. Se puede ver su interfaz en la figura 3.1.

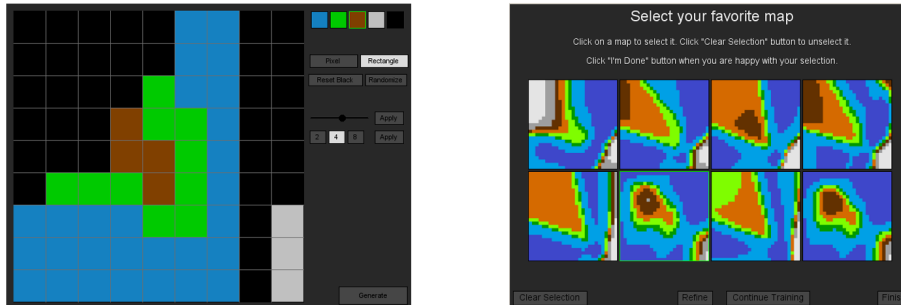


Figura 3.1: Interfaz de Sentient World; se crea terreno simple y la herramienta da alternativas de su expansión.

Ropossum [39], por su parte, se orienta al diseño de rompecabezas, ofreciendo soporte evolutivo que ayuda a ajustar la dificultad, la solución y el flujo del nivel de forma conjunta con el usuario.

Finalmente, *Cicero* [40] aborda la generación narrativa mediante iniciativa mixta, permitiendo que autores y algoritmos co-construyan historias, progresiones emocionales y decisiones relevantes dentro del arco narrativo. En conjunto, estos sistemas evidencian cómo la MI-PCG fortalece el proceso creativo al expandir el espacio de diseño y reducir la carga técnica del creador.

3.2. Generación de Terreno

Dentro de los enfoques orientados a la generación de terrenos, destacan herramientas como *Gaia Pro VS* [41] que permiten generar paisajes, texturizado, plantar vegetación y poblar automáticamente recursos mediante la combinación de nodos y capas de ruido. Su principal fortaleza radica en su integración directa con Unity y su capacidad para producir entornos visualmente realistas en poco tiempo.

Por otro lado, *World Machine* [42] se posiciona como una solución más avanzada en cuanto a control granular sobre la topografía, utilizando sistemas basados en nodos y simulaciones físicas de erosión, lo que la convierte en una herramienta ideal para diseñar terrenos de alta fidelidad.

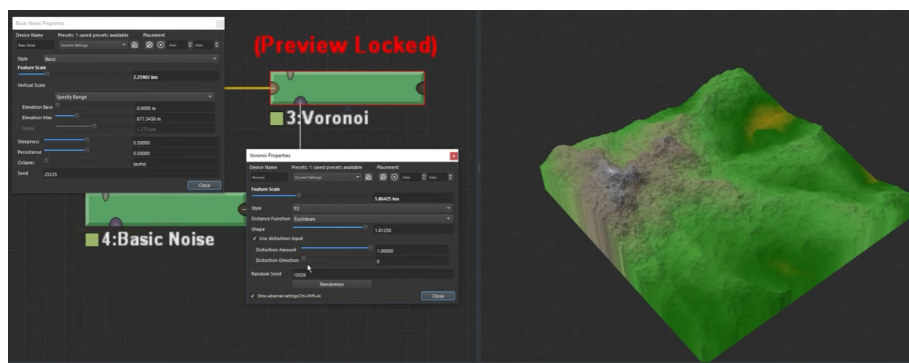


Figura 3.2: Interfaz de World Machine; utiliza un sistema de nodos para crear terreno.

Finalmente, *Voxel Farm Indie* [43] destaca por su enfoque en la generación volumétrica de mundos, permitiendo la creación de entornos destructibles y editables en tiempo real, aunque tenga un alto costo computacional.

Uno de los problemas que tienen todos es que son herramientas utilizadas en la preproducción del juego, funcionando dentro del editor y no durante la ejecución online de este.

Estudios al respecto de la generación procedural de terreno en tiempo real [21] exploraron metodologías para generar paisajes erosionados de forma dinámica durante la ejecución, demostrando la viabilidad técnica del enfoque, aunque con limitaciones en rendimiento y detalle visual para la época.

Este avance marca los primeros pasos hacia sistemas que no solo generan mundos de forma procedural, sino que los mantienen adaptativos y evolutivos durante el gameplay, un enfoque que se alinea estrechamente con la dirección que busca implementar LBS.

3.3. Generación de Población

En cuanto a la generación procedural de población y niveles, existen herramientas y proyectos de investigación que buscan automatizar la colocación de objetos, enemigos o estructuras dentro de un entorno ya generado.

GeNa Pro [44] es una extensión complementaria de *Gaia*, centrada en la distribución inteligente de objetos y vegetación sobre el terreno, utilizando reglas configurables y detección de colisiones para mantener coherencia espacial. Al igual que

Gaia, está diseñada para funcionar principalmente dentro del editor, no en tiempo de ejecución.

Dungeon Architect [45], por su parte, ofrece un enfoque modular basado en *blueprints* o grafos de nodos, permitiendo la creación automática de niveles de tipo mazmorra con estructuras lógicas y rutas de navegación. Por otro lado, *Sentient Sketchbook* [46] introdujo un enfoque de co-creación humano-IA que ofrece al diseñador sugerencias generadas en tiempo real mediante búsqueda evolutiva (novelty search) [47]. Este sistema representa un punto de partida relevante para la investigación en generación procedural con intervención humana directa.



Figura 3.3: Interfaz de Sentient Sketchbook; en base a los recuadros colocados en el lado izquierdo, la herramienta hace sugerencias en dónde pueden ser colocados en la lado derecho.

En la figura 3.3 se puede ver su interfaz, siendo inspiración directa a la versión actual del módulo 2 de LBS.

Comparando las herramientas analizadas, se puede observar que las orientadas a la generación de terrenos privilegian la calidad visual y la escala, mientras que las centradas en población o diseño de niveles enfatizan la estructura y jugabilidad. No obstante, en ambos casos la generación procedural se ejecuta principalmente en entornos de preproducción o dentro del editor, sin extenderse hacia contextos online (excepto por la herramienta *Vista*), donde los entornos puedan modificarse

activamente durante el juego.

En este escenario, la propuesta de Level Building Sidekick busca unificar ambos enfoques, integrando la generación de terreno y población de manera dinámica dentro del propio gameplay, explorando nuevas formas de generación procedural online bajo la iniciativa mixta (MI-PCG), donde tanto el diseñador como el sistema colaboren para producir mundos adaptativos, coherentes y evolutivos.

3.4. Aplicaciones PCG Online

Existen también herramientas de PCG que incorporan mecanismos de generación durante la ejecución del juego. *Dungeon Architect* [45] permite la generación procedural de niveles orientados a *dungeon crawlers*, incluyendo funcionalidades para generar contenido en tiempo de ejecución, visible en la figura 3.4. *Daedalus* [48], aunque actualmente se encuentra abandonado, fue un proyecto orientado a la generación procedural de distintos tipos de terrenos y entornos. Por otra parte, *Apparance* [49] permite generar objetos paramétricos mediante reglas definidas por el usuario, posibilitando la creación y modificación de contenido de forma dinámica.

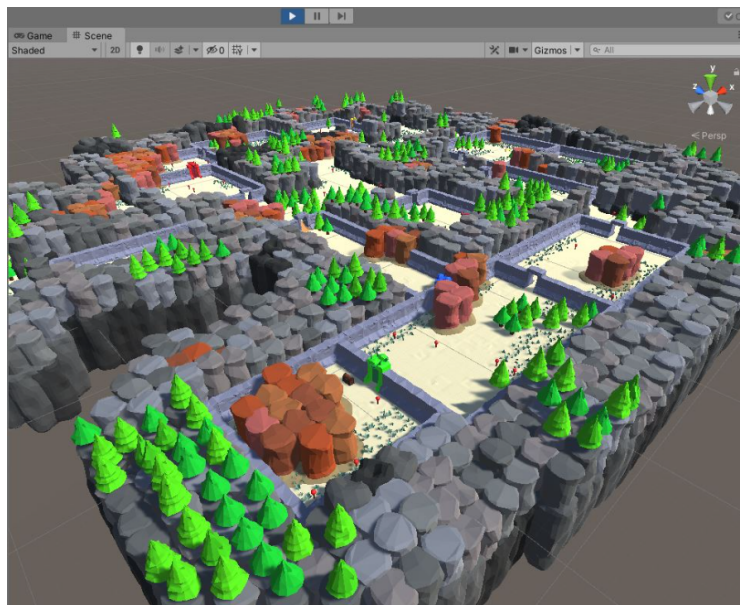


Figura 3.4: Dungeon Architect ejecutando funciones de generación de mazmorras en tiempo de ejecución en el editor de Unity.

Estas herramientas demuestran la viabilidad de utilizar PCG durante *runtime*. Sin embargo, sus objetivos se centran en dominios específicos, como la generación de mazmorras, terrenos u objetos paramétricos, sin abordar directamente la adaptación de algoritmos de generación previamente diseñados para un entorno de edición hacia una ejecución continua durante el *gameplay*.

4. Métodos

Para poder responder a la pregunta de investigación, primero se debe de cambiar el enfoque de la herramienta a una orientación que no dependa de interfaces que deban ser manipuladas por el usuario. La herramienta debe de funcionar separada de la interfaz, con la capacidad de arrancar y correr de forma continua.

4.1. Arquitectura propuesta

La arquitectura propuesta para la implementación, apodada **RuntimeLBS**, se basa en el retiro de la intervención que debe dar el usuario al utilizar LBS, la separación entre la lógica atada al editor de Unity, y utilizarla de manera que no dependa de este.

Además, se separará la lógica de generación procedural de la instanciación de objetos dentro de Unity. Mientras que los algoritmos de generación realizan los cálculos en segundo plano mediante tareas asíncronas (*Async Tasks*), la creación de los objetos en escena permanece en el hilo principal del motor, respetando las restricciones impuestas por Unity para la manipulación de `GameObjects`.

Con el propósito de mantener la compatibilidad con la implementación original de LBS, se optó por desarrollar una extensión independiente en lugar de modificar directamente sus clases. Para ello, se reutilizó parte de la lógica existente mediante copias de los componentes responsables de la ejecución de los algoritmos, eliminando las dependencias exclusivas del editor e incorporando nuevas clases encargadas de coordinar la ejecución durante *runtime*. Esta decisión permite conservar intacta la versión original de LBS y facilita el mantenimiento de ambas implementaciones.

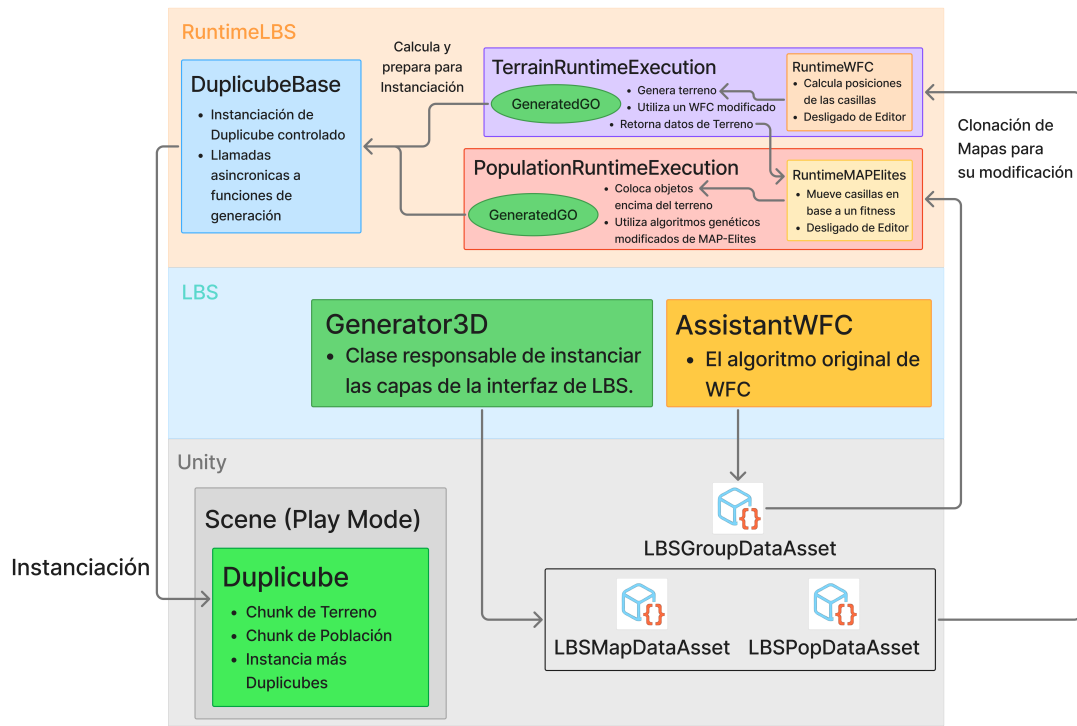


Figura 4.1: Diagrama de bloques de la estructura de *RuntimeLBS*, mostrando sus conexiones con el código original y Unity.

La estructura de *RuntimeLBS* se presenta en la Figura 4.1. El proceso comienza con la herramienta LBS, desde donde se crean los mapas base y las referencias de terreno y población, almacenados en los *Scriptable Objects* (*LBSMapDataAsset*, *LBSPopDataAsset* y *LBSGroupDataAsset*). Estos datos son utilizados por las clases *TerrainRuntimeExecution* y *PopulationRuntimeExecution*, las cuales ejecutan las versiones adaptadas de los algoritmos *RuntimeWFC* y *RuntimeMAPElites*, respectivamente. Para evitar bloquear el hilo principal de Unity, ambas clases delegan los cálculos a tareas asíncronas y, una vez finalizado el procesamiento de un *chunk*, envían los resultados a *DuplicubeBase*, encargado de coordinar la instanciación del terreno y la población. Finalmente, los elementos son instanciados sobre objetos *Duplicube*, los cuales detectan la expansión del escenario y solicitan la generación de nuevos *chunks*.

Los parámetros necesarios para la generación son obtenidos desde un conjunto

de *Scriptable Objects*, los cuales almacenan la configuración base del mapa, los pesos utilizados por *Wave Function Collapse* y la información de los elementos que pueden ser distribuidos por el algoritmo genético empleado en *MAP-Elites*.

4.2. Adaptación del módulo de generación de exteriores (Paso 1B)

La implementación original de LBS dependía de componentes propios del entorno de edición, ejecutaba la totalidad del algoritmo sobre el hilo principal y asumía un flujo de trabajo orientado a la generación previa al inicio del juego, lo que impedía su utilización durante *runtime* debido a los bloqueos producidos durante el cálculo del terreno.

Para adaptar este módulo a un contexto de ejecución en tiempo real, se decidió conservar la lógica principal del algoritmo Wave Function Collapse (WFC) e intervenir únicamente sobre aquellos aspectos relacionados con su arquitectura de ejecución, eliminando las dependencias del editor y reorganizando su flujo para permitir una ejecución asíncrona sin modificar el comportamiento esperado del algoritmo.

Las modificaciones realizadas pueden resumirse en los siguientes puntos:

- Eliminación de las dependencias exclusivas del editor de Unity.
- Eliminación de llamadas y callbacks asociados a la interfaz del editor.
- Simplificación de algunas estructuras de datos utilizadas durante el cálculo.
- División de la generación del terreno en chunks independientes.
- Incorporación de procesamiento asíncrono mediante *Async Tasks*.
- Separación entre el cálculo del algoritmo y la instanciación de los objetos en la escena.

Estas modificaciones permitieron que el algoritmo pudiera ejecutarse durante el tiempo de juego sin alterar su funcionamiento interno ni bloquear el hilo principal del motor.

4.2.1. Arquitectura interna

Para implementar estas modificaciones se incorporaron tres componentes principales, cada uno con una responsabilidad específica dentro del proceso de generación:

Componente	Proposito
RuntimeWFC	Contiene una adaptación de la implementación original de WFC, eliminando las dependencias con el editor de Unity y conservando la lógica principal del algoritmo.
TerrainRuntimeExecution	Coordina el proceso de generación del terreno. Calcula el contexto necesario para cada chunk, ejecuta el algoritmo mediante tareas asíncronas y envía el resultado para su posterior instanciación.
GeneratorGO	Modificación del instanciador de terrenos. Recibe la información generada por el algoritmo y realiza la instanciación de los <i>GameObjects</i> correspondientes dentro del hilo principal de Unity.

Tabla 4.1: Componentes principales presentes en la implementación de *RuntimeLBS* de la generación de terrenos.

4.2.2. Flujo de Ejecución

Cuando el sistema requiere generar un nuevo chunk, la clase *TerrainRuntimeExecution* obtiene la información necesaria para construir el contexto del terreno utilizando los datos almacenados en los *Scriptable Objects* correspondientes. Posteriormente, crea una tarea asíncrona encargada de ejecutar *RuntimeWFC*, permitiendo que el algoritmo realice todos sus cálculos sin interrumpir la ejecución del juego.

Una vez finalizado el procesamiento, el resultado obtenido es enviado a una cola de instanciación. Este mecanismo garantiza que la creación de los objetos se realice posteriormente desde el hilo principal de Unity, evitando violar las restricciones del motor respecto a la manipulación de *GameObjects*. Puede visualizar el flujo de ejecución a través de la figura C.1.

4.3. Adaptación del módulo de generación de población (Paso 2)

Si bien el enfoque de MAP-Elites resulta adecuado para una herramienta de edición, introduce etapas de interacción y evaluación que no son compatibles con un sistema de generación automática durante el tiempo de ejecución.

La propuesta desarrollada en esta memoria implementa una versión distinta del algoritmo genético. A causa de la falta de tiempo, se optó por hacer un distribuidor de objetos con posiciones aleatoria, eliminando las etapas de exploración y selección de múltiples soluciones que caracterizan a la implementación original.

Las modificaciones realizadas pueden resumirse en los siguientes puntos:

- Eliminación de las dependencias exclusivas del editor de Unity.
- Eliminación de la interfaz asociada al proceso de selección de soluciones.
- Conservación del algoritmo genético utilizado internamente por MAP-Elites.
- Simplificación del proceso evolutivo para producir una única solución.
- Adaptación del sistema para recibir como contexto el chunk generado previamente por el módulo de terreno.
- Incorporación de procesamiento asíncrono mediante *Async Tasks*.
- Separación entre el cálculo de la distribución y la instanciación de los objetos en la escena.

Con este cambio, el módulo de población dejará de funcionar como un sistema de exploración de soluciones asistido por el usuario y pasará a comportarse como un proceso automático orientado a generar una única distribución de objetos adecuada para cada chunk.

4.3.1. Arquitectura Interna

La adaptación del módulo se estructuró en torno a tres componentes principales: *RuntimeMAPElites*, *PopulationRuntimeExecution* y una reimplementación del *GeneratorGO*. El primero contiene un algoritmo que mueve partes de forma aleatoria, desacoplada del editor de Unity, el segundo coordina su ejecución dentro del flujo general de Runtime LBS, y el tercero se encarga de colocar en escena los objetos generados por el algoritmo.

Componente	Proposito
RuntimeMAPElites	Ejecuta una función aleatoria, sin estar ligada a dependencias del editor y simplificando el proceso de búsqueda a una única solución.
PopulationRuntimeExecution	Recibe el chunk generado por el módulo de terreno, coordina la ejecución del algoritmo mediante tareas asíncronas y prepara el resultado para su posterior instanciación.
GeneratorGO	Implementación propia del sistema de instanciación, encargada de crear los GameObjects de población en el hilo principal de Unity y sincronizar su aparición con el flujo de generación de chunks.

Tabla 4.2: Componentes principales presentes en la implementación de *RuntimeLBS* de la generación de población.

4.3.2. Flujo de Ejecución

Una vez finalizada la generación del terreno, el chunk resultante es utilizado como contexto para la generación de población. A partir de esta información, *PopulationRuntimeExecution* inicia una tarea asíncrona encargada de ejecutar *RuntimeMAPElites*, el cual genera una distribución inicial de objetos y aplica una distribución aleatoria en este.

Esta versión simplificada se salta varios pasos de evaluación y distribución que contiene la implementación de MAP-Elites original, ya sea la selección de cómo se organizan los objetos, la evolución de la distribución y la selección de una de las variantes que prefiera el usuario.

Una vez obtenida la distribución final, los objetos generados son enviados a la cola de instanciación, donde posteriormente son incorporados a la escena desde el hilo principal del motor Unity. Se puede visualizar el flujo en la figura C.2

4.4. Consideraciones sobre la ejecución asíncrona

Uno de los principales desafíos de adaptar los módulos de LBS al tiempo de ejecución consistió en identificar qué partes del proceso de generación podían ejecutarse de manera asíncrona y cuáles debían permanecer en el hilo principal de Unity. Aunque los algoritmos de generación están compuestos principalmente por cálculos independientes del motor, algunas operaciones requieren acceder a objetos administrados por Unity, los cuales únicamente pueden manipularse desde el *main thread*.

En el módulo de generación de terreno, la totalidad de los cálculos asociados al algoritmo WFC pudo separarse del hilo principal sin afectar su funcionamiento. Esto permitió ejecutar el proceso completo mediante tareas asíncronas, dejando únicamente la instanciación de los *GameObjects* como una operación realizada desde el *main thread*.

Por otra parte, la adaptación del módulo de población requiere de un análisis y separación más cuidadosa. Algunas etapas previas relacionadas con la preparación de los datos debieron mantenerse en el hilo principal. En particular, el proceso encargado de construir la capa de población obtiene referencias a *Bundles* y otros recursos administrados por Unity para determinar los objetos disponibles que pueden ser utilizados durante la generación. Debido a que estas operaciones dependen de la API del motor, no pueden ejecutarse desde un hilo secundario y deben completarse antes de iniciar el procesamiento asíncrono. La organización de procesos y cómo se ejecutan se pueden ver en la figura 4.2.

La ejecución asíncrona se implementó mediante la clase *Task* de *.NET*, la cual permite delegar operaciones computacionalmente intensivas a hilos administrados por el *ThreadPool*. Esta aproximación resultó adecuada para los algoritmos de generación, ya que sus cálculos no requieren acceso continuo a la *API* de Unity y pueden ejecutarse de manera independiente del hilo principal. Una vez finalizada la tarea, el resultado es recuperado mediante *await*, permitiendo continuar el flujo de generación sin bloquear la ejecución del juego.

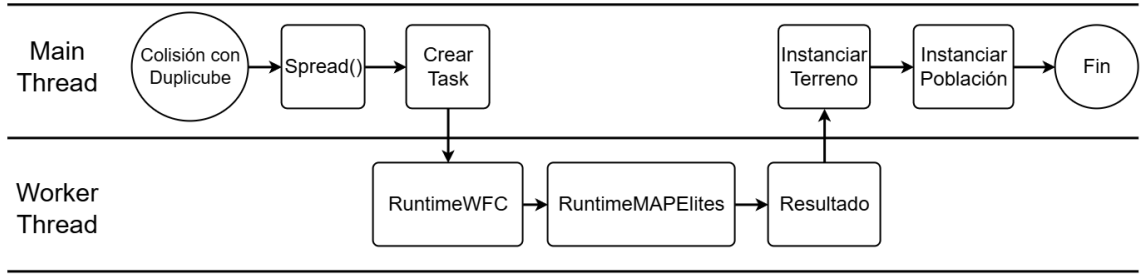


Figura 4.2: Diagrama de flujo de funciones cruzadas, la cual muestra la interacción entre el hilo principal con el hilo asíncrono. Se genera a través de Task.

4.5. Flujo de generación de un chunk

Las modificaciones realizadas sobre los módulos de generación de terreno y población permitieron trasladar ambos procesos al tiempo de ejecución. Sin embargo, fue necesario definir un mecanismo que coordinara la expansión del mundo, la ejecución de los algoritmos y la instanciación de los elementos generados. Para ello se implementó un flujo secuencial que organiza cada una de estas etapas, asegurando que cada chunk sea generado una única vez y evitando conflictos derivados de la ejecución concurrente de múltiples tareas.

Algoritmo 4.1 Proceso de generación de un nuevo *chunk*

- 1: **if** PositionExists(*position*) **then**
 - 2: **return**
 - 3: RegisterPosition(*position*)
 - 4: Spread()
 - 5: *chunk* ← CalculateChunkAsync()
 - 6: CalculatePopAsync(*chunk*)
 - 7: InstantiateTerrainChunk()
 - 8: InstantiatePopulationChunk()
-

El pseudocódigo (4.1) presenta el flujo general de la implementación para la generación de un nuevo chunk. En ella se observa la secuencia de operaciones que comienza con la detección de un nuevo sector del mapa y finaliza con la incorporación del terreno y los objetos de población a la escena.

La generación comienza cuando el jugador entra en contacto con un *Duplicube*. Este evento invoca el método **Spread()** de *DuplicubeBase*, encargado de calcular las

posiciones adyacentes donde podrán generarse nuevos chunks. Antes de crear cada nuevo *Duplicube*, el sistema verifica mediante un *HashSet* que la posición no haya sido utilizada anteriormente, evitando generar regiones duplicadas.

Una vez validada la posición, *DuplicubeBase* inicia el proceso de generación llamando a `ProcessSpawnQueue()` (lo que se puede visualizar en el pseudocódigo 4.1). Esta función coordina la ejecución secuencial de los módulos de terreno y población: primero se ejecuta `CalculateChunkAsync()`, visualizable en B.1, que genera el chunk mediante *RuntimeWFC*, y posteriormente `CalculatePopAsync()`, visible en B.2 que utiliza el terreno generado como contexto para ejecutar el algoritmo genético adaptado desde *MAP-Elites*. Ambos cálculos se realizan mediante tareas asíncronas.

Finalmente, los resultados obtenidos son instanciados en la escena desde el hilo principal de Unity, creando primero el terreno y luego los objetos de población, y mostrando lo que se puede ver en la figura A.1. Para evitar que un mismo *Duplicube* inicie múltiples procesos de generación simultáneamente, se incorporó un mecanismo de sincronización que bloquea nuevas ejecuciones hasta que el proceso asociado al chunk haya finalizado completamente.

En síntesis, se describieron las modificaciones realizadas sobre los módulos de generación de terreno y población para permitir su ejecución durante el tiempo de juego. Sin embargo, la incorporación de estas modificaciones requiere comprobar si efectivamente permiten ejecutar ambos algoritmos con un rendimiento estable, demorándose menos de *500ms* en ciertos escenarios, y sin introducir bloqueos perceptibles durante la ejecución del juego. Por esta razón, el presente capítulo describe los experimentos diseñados para evaluar el comportamiento de la implementación propuesta y comparar su desempeño con la versión original de LBS.

5. Experimentos

5.1. Diseño Experimental

Con el propósito de responder la pregunta de investigación, se diseñó un par de experimentos orientados a evaluar el comportamiento de Runtime LBS desde dos grupos:

- El desempeño temporal de los algoritmos de generación.
- El impacto de la generación sobre los recursos del sistema durante la ejecución del juego.

El primer grupo evalúa el comportamiento de **RuntimeLBS** frente a distintos tamaños de chunk, utilizando configuraciones de **5×5**, **10×10** y **20×20** celdas, durante un periodo de tiempo de **5 minutos** por cada tamaño. En cada caso se registran los tiempos de ejecución correspondientes al cálculo del terreno mediante WFC, la generación de población mediante el algoritmo genético, la instanciación del terreno, la instanciación de los objetos de población y el tiempo total requerido para completar la generación de un chunk. Estas mediciones permiten analizar cómo escala el costo computacional de la solución a medida que aumenta la cantidad de contenido generado.

El segundo grupo compara la implementación propuesta con la versión original de LBS. Para ello se genera un mapa de **50×50** celdas utilizando ambas implementaciones bajo una configuración equivalente, considerando una capa de terreno y una capa de población.

El objetivo de esta comparación es determinar el impacto que producen la eliminación de las dependencias del editor y la incorporación de ejecución asíncrona sobre el tiempo total de generación.

Adicionalmente, durante todas las pruebas se registran métricas relacionadas con el comportamiento del sistema, incluyendo el consumo de CPU, memoria RAM y la tasa de cuadros por segundo (FPS). Estas mediciones permiten evaluar si la generación procedural mantiene un comportamiento estable durante la ejecución del juego y complementan el análisis temporal realizado en los experimentos anteriores.

5.2. Configuración del entorno

Los experimentos fueron ejecutados en un computador portátil con las siguientes características:

Componente	Especificación
Procesador	Intel Core i5-11400H (6 núcleos, 2,70 GHz)
Tarjeta gráfica	NVIDIA GeForce RTX 3050 Laptop
Memoria RAM	32 GB DDR4 3200 MT/s
Almacenamiento	SSD NVMe Western Digital Black 512 GB
Sistema operativo	Windows 11
Motor de videojuegos	Unity 6 (6000.0.59f2)

Tabla 5.1: Características del equipo a utilizar para los experimentos.

Los experimentos correspondientes a la generación de chunks de 5×5 , 10×10 y 20×20 se ejecutarán dentro del *Editor* de *Unity*. Se hubiera deseado que las pruebas se hayan hecho en una *Build* del juego, pero los algoritmos utilizados están fuertemente ligados a funciones que no permiten su uso fuera de este (a menos que se modifiquen parámetros, pero no se deseó sacrificar más tiempo del debido). Igualmente, la comparación entre la implementación original de LBS y la propuesta, con el tamaño de 50×50 , se realizará completamente **dentro del editor**.

Las mediciones y métricas fueron obtenidas por las siguientes herramientas:

Métrica	Herramienta
Tiempo de ejecución	<i>System.Diagnostics.Stopwatch</i>
Tiempo de CPU	<i>FrameTiming</i>
Tiempo de GPU	<i>FrameTiming</i>
Consumo de RAM	Unity Profiler (API)
FPS	<i>Time.unscaledDeltaTime</i>

Tabla 5.2: Herramientas utilizadas dentro de Unity para medir las métricas descritas.

5.3. Escenario de Pruebas

Las pruebas serán ejecutadas dentro de un Escenario de Unity vacío, donde se encontrarán los *scripts* necesarios para que arranquen los algoritmos de *RuntimeLBS*, y un cubo que contiene un colisionador.

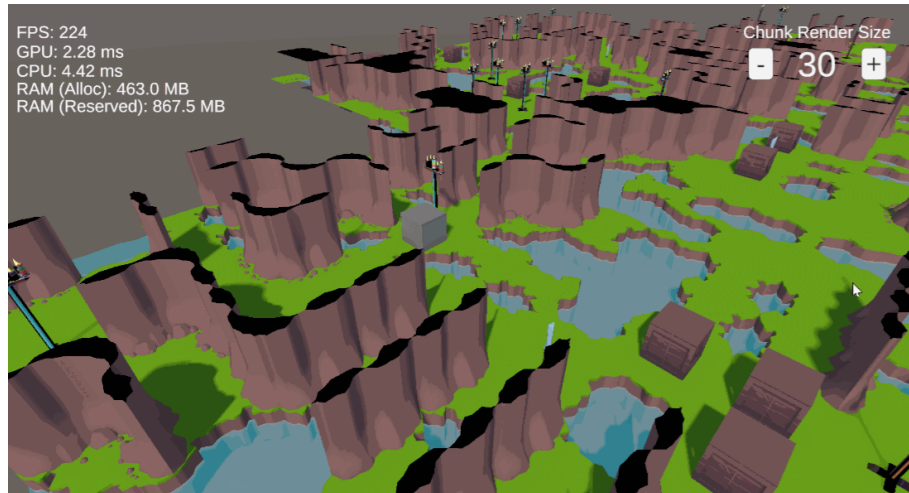


Figura 5.1: Escenario de prueba, donde el cubo se está moviendo hacia la izquierda, provocando que se genere terreno relativo a su dirección.

El cubo se podrá mover de forma automática hacia una dirección con una velocidad predeterminada, o hacia cualquier dirección multidimensional usando las flechas para controlar su navegación. Al momento de que el cubo colisione con uno de los *Duplicubes*, se ejecutará la función *Spread*, gatillando una reacción en cadena que instanciará *chunks* relativa a la dirección en el que el cubo esté moviendose.

5.4. Experimento 1: Pruebas de tiempo por tamaño de Chunk

El primer experimento tiene como objetivo evaluar el comportamiento de la implementación propuesta durante la generación procedural en tiempo de ejecución. Para ello, se analizará el rendimiento del sistema al generar *chunks* de distintos tamaños, permitiendo observar cómo varía el costo computacional a medida que aumenta la cantidad de contenido generado.

5.4.1. Procedimiento

Las pruebas se realizan dentro del Editor de Unity, configurando el sistema para que genere *chunks* de 5×5 , 10×10 y 20×20 celdas, observables en la figura A.2, mediante el mecanismo de expansión implementado por *RuntimeLBS*. Para cada tamaño, siendo estas variables independientes, el sistema genera una secuencia continua de chunks a medida que el jugador recorre el escenario, registrando las métricas correspondientes a cada proceso de generación.

Durante cada generación se mide de forma independiente el tiempo requerido para ejecutar el algoritmo RuntimeWFC, el algoritmo genético adaptado desde MAP-Elites, la instanciación del terreno y la instanciación de los objetos de población. Asimismo, se registrará el tiempo total necesario para completar la generación de cada *chunk*. Paralelamente, se obtienen métricas del comportamiento general del sistema, incluyendo el tiempo de CPU, tiempo de GPU, consumo de memoria RAM y la tasa de cuadros por segundo (FPS).

5.4.2. Variables Medidas

Las variables registradas durante este experimento corresponden a las métricas de rendimiento obtenidas durante la generación de cada *chunk*. Estas variables permiten analizar el comportamiento temporal de cada etapa del proceso de generación, así como el impacto producido sobre el rendimiento general de la aplicación.

Variable	Unidad
Tiempo de cálculo de <i>RuntimeWFC</i>	ms
Tiempo de ejecución del algoritmo genético	ms
Tiempo de ejecución de terreno	ms
Tiempo total de generación del <i>chunk</i>	ms
Tasa de cuadros por segundo	FPS
Tiempo de respuesta de CPU promedio	ms
Tiempo de respuesta de GPU promedio	ms
Consumo de Memoria RAM	MB

Tabla 5.3: Variables a considerar para el experimento 1.

Todas estas variables son dependientes del tamaño de los *chunks*. Las métricas serán registradas automáticamente desde la aplicación durante cada proceso de generación, permitiendo obtener una medición individual para cada chunk generado sin intervenir el flujo normal de ejecución del sistema.

5.5. Experimento 2: Comparación entre la implementación original de LBS y Runtime LBS

El objetivo de este experimento es evaluar el impacto que tienen las modificaciones propuestas sobre el rendimiento de la herramienta, verificando si estas permiten reducir el tiempo de generación sin alterar el comportamiento funcional de los módulos de terreno y población. En particular, se busca determinar si la eliminación de las dependencias del editor y la incorporación de un modelo de ejecución orientado al tiempo de juego reducen el tiempo necesario para generar contenido procedural.

5.5.1. Procedimiento

La comparación se realiza utilizando un escenario equivalente en ambas implementaciones. Para ello, se configura una generación de un solo chunk en *RuntimeLBS*, compuesto por **50×50** celdas. Luego, en la herramienta de LBS, se crea una *capa de terreno* y una *capa de población* con las mismas medidas, así como se ve en la figura

A.3, utilizando la misma información base y parámetros de generación en ambos casos.

Durante cada generación se registran los tiempos correspondientes al cálculo del terreno mediante WFC, la ejecución del algoritmo genético utilizado para la población, la instanciación del terreno, la instanciación de los objetos de población y el tiempo total requerido para completar el proceso de generación.

Los tiempos registrados permitirán comparar directamente ambas implementaciones y analizar el efecto de las modificaciones arquitectónicas propuestas sobre el rendimiento del sistema.

5.5.2. Variables Medidas

Para la implementación de *RuntimeLBS* y LBS se registrarán las mismas métricas temporales, siendo variables dependientes al tamaño de la generación, permitiendo realizar una comparación directa entre cada una de las etapas del proceso de generación.

Variable	Unidad
Tiempo de cálculo de WFC	ms
Tiempo de cálculo del algoritmo genético	ms
Tiempo de Instanciación del terreno	ms
Tiempo de Instanciación de población	ms
Tiempo total de generación	ms

Tabla 5.4: Variables a considerar para el experimento 2.

La única variable dependiente en este caso sería el método utilizado para hacer el experimento, siendo este *RuntimeLBS* o LBS.

6. Resultados

6.1. Resultados del Experimento 1

Tras las pruebas hechas en un tiempo de 5 minutos por cada tamaño, se obtuvieron los siguientes resultados:

6.1.1. Tiempos de Generación

Tamaño	RuntimeWFC	RuntimeM.E.	Inst. Terreno	Inst. Población
5x5	113.21ms	0.34ms	3.16ms	0.79ms
10x10	492.78ms	0.49ms	10.57ms	1.02ms
20x20	2793.87ms	0.68ms	38.39ms	1.05ms

Tabla 6.1: Tiempos de ejecución en promedio de funciones de *RuntimeLBS*, medidas en milisegundos.

Los resultados obtenidos muestran que el tiempo total de generación aumenta en relación al tamaño del *chunk*. La mayor parte del tiempo de ejecución corresponde al cálculo realizado por *RuntimeWFC*, responsable de coordinar el proceso completo de generación del terreno. En comparación, las etapas correspondientes al algoritmo genético (*RuntimeMAPElites*) presentan tiempos menores y prácticamente constantes para los tres tamaños evaluados. Esto se debe a que durante las evaluaciones, siempre se mantuvo la cantidad de objetos a instanciar en el *chunk* de población en una cantidad aleatoria de 2 a 6 objetos, sin importar el tamaño.

Además, observando los tiempos en promedio de generación de un *chunk* de terreno y población (de la figura D.2), se puede notar una gran subida de tiempo

relativa al tamaño del *chunk* en sí, hasta alcanzar los **3657.85ms** en un tamaño de **20x20**. Cabe mencionar que la generación de cada tile tuvo una gran subida entre los de **10x10** y **20x20**.

6.1.2. Comportamiento durante la generación continua

A la par que se obtuvieron los promedios de tiempos de ejecución, se capturaban los tiempos de cálculo de terreno de cada *chunk* individualmente, para así ver alguna variación en base al tiempo.

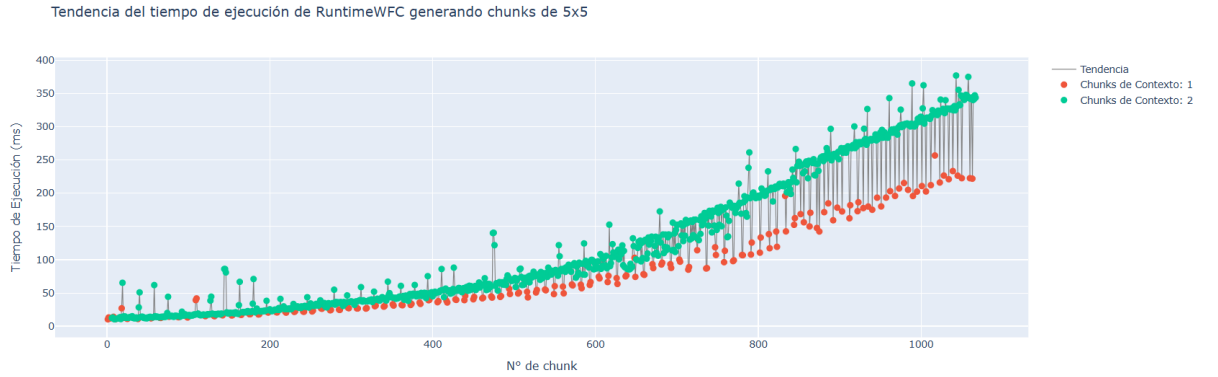


Figura 6.1: Gráfico de líneas que muestra el tiempo de cálculo de *RuntimeWFC* por cada chunk individual, de tamaño 5x5.

La Figura 6.1 muestra la evolución del tiempo de ejecución de *RuntimeWFC* durante la generación continua de *chunks* de **5x5**. Puede observarse que el tiempo de cálculo aumenta progresivamente, desde velocidades debajo de los **100ms** hasta llegar a un pico de **350ms**, a medida que se generan nuevos *chunks*. Además, en la misma figura, se pueden apreciar dos agrupaciones claramente diferenciadas por colores, correspondientes a *chunks* que poseen uno o dos vecinos previamente generados. Los tiempos capturados de los *chunks* con dos vecinos (puntos de color verde) tienen un mayor tiempo de cálculo que los que tienen un vecino (puntos de color rojo).

En el resto de gráficos de tamaños **10x10** (D.5) y **20x20** (D.9) se puede notar una menor cantidad de *chunks* generada en el transcurso del tiempo, pero el fenómeno de la alza de tiempo está igualmente presente.

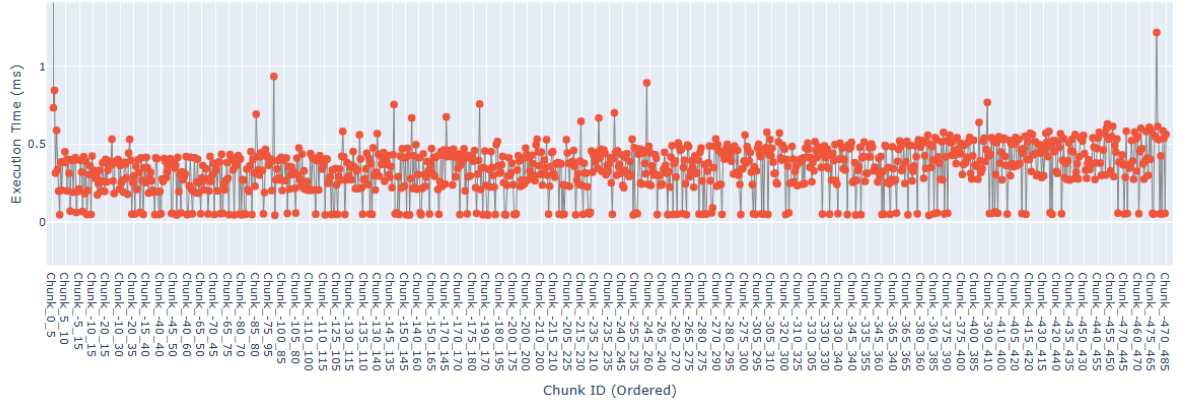


Figura 6.2: Gráfico de líneas que muestra el tiempo de cálculo de *RuntimeMapElites* por cada chunk individual, de tamaño 5x5.

En contraste, la Figura 6.2 presenta el tiempo requerido para la ejecución de algoritmo genético de *Map Elites*. A diferencia del cálculo anterior, este permanece relativamente constante durante toda la ejecución, con tiempos marcados entre **0.5ms** hasta mayores a **1ms**. Ocurre lo mismo en los tamaños de **10x10** (D.6) y **20x20** (D.10), aunque en estos se puede notar que el primer *chunk* tardó más que el resto.

6.1.3. Rendimiento general del sistema

Chunk	FPS	T. CPU	T. GPU	RAM Alloc.	RAM Reserved
5x5	164.40	6.01ms	3.62ms	746.88	1435.06
10x10	167.51	6.35ms	2.93ms	786.31	1437.71
20x20	190.90	6.24ms	2.07ms	829.92	1457.31

Tabla 6.2: Métricas promedio capturadas durante el experimento, separado por el tamaño de chunk generándose.

Las métricas generales de rendimiento, apreciables en la tabla 6.2 muestran que la generación continua de chunks mantiene un comportamiento estable durante todas las configuraciones evaluadas. El tiempo de respuesta de CPU se mantiene cercano a **6ms** para los tres tamaños de *chunk*, mientras que el tiempo de respuesta de GPU

permanece por debajo de **4ms**, sin evidenciar incrementos significativos asociados al aumento del tamaño del terreno generado.

Por otra parte, el consumo de memoria RAM presenta un incremento gradual conforme aumenta el tamaño de los *chunks*, comportamiento esperable debido a la mayor cantidad de información y objetos almacenados en memoria. Sin embargo, dicho incremento no produjo una disminución apreciable en la tasa de cuadros por segundo, la cual se mantuvo por sobre los **160 FPS** durante todas las pruebas realizadas. Estos resultados sugieren que la implementación propuesta permite mantener una ejecución estable incluso durante procesos continuos de generación procedural.

6.2. Resultados del Experimento 2

Después de crear un chunk de **50x50** en la interfaz de LBS y hacer su equivalente en el escenario de prueba de *RuntimeLBS*, los resultados fueron los siguientes:

Etapa	LBS	Runtime LBS
WFC	29641.80ms	2768.15ms
MAP-Elites / Algoritmo Aleatorio	33941.00ms	21.31ms
Instanciación	1059.00ms	287.83ms
Tiempo total	64641.80ms	3077.29ms

Tabla 6.3: Tiempos obtenidos durante la generación de un escenario de 50×50 celdas en LBS y *RuntimeLBS*.

Los resultados que aparecen en la tabla 6.3 muestran una reducción considerable en el tiempo total de generación. Mientras la implementación original requiere de aproximadamente **64,6** segundos para completar el proceso, la implementación propuesta realiza la misma tarea en aproximadamente **3,1** segundos, lo que representa una disminución superior al **95 %** del tiempo total de ejecución.

6.2.1. Comparación entre LBS y Runtime LBS

En el caso de WFC, la reducción del tiempo puede atribuirse principalmente a la eliminación de las dependencias del editor de Unity, la simplificación de parte de la implementación original y la incorporación de estructuras de datos adaptadas para su ejecución en tiempo de juego. Adicionalmente, la ejecución del algoritmo fue

trasladada a tareas asíncronas, permitiendo que los cálculos se realizaran fuera del hilo principal mientras la aplicación continuaba ejecutándose.

La mayor diferencia se observa durante la generación de la población. A diferencia de la implementación original, *RuntimeLBS* no ejecuta el flujo completo de *MAP-Elites*, sino una función aleatoria que simplemente esparce objetos por el mapa, sin pasar por un algoritmo que consuma mucho poder computacional. Esta modificación reduce significativamente la cantidad de cálculos necesarios para obtener una solución válida, disminuyendo el tiempo requerido para generar la población.

Asimismo, el proceso de instanciación presentó una reducción en su tiempo de ejecución. Si bien ambas implementaciones generan el escenario dividiéndolo internamente en 25 secciones, *Runtime LBS* incorpora una cola de instanciación que controla la creación de cada chunk de forma secuencial, evitando bloqueos y distribuyendo el trabajo durante la ejecución del sistema.

7. Conclusiones

El objetivo principal de esta investigación fue responder a qué modificaciones eran necesarias de hacer en las implementaciones de los algoritmos dentro de la herramienta de LBS para poder ejecutarlos en tiempo de ejecución con un rendimiento estable.

Los resultados obtenidos permiten concluir que la adaptación de ambos módulos requiere más que una simple migración desde el editor hacia la ejecución en tiempo de juego. Fue necesario desacoplar las implementaciones de las dependencias propias del editor de Unity, reorganizar el flujo de generación mediante tareas asíncronas, separar los procesos de cálculo e instanciación y controlar la creación de contenido mediante un sistema de colas. En conjunto, estas modificaciones permitieron ejecutar la generación procedural durante el gameplay sin producir bloqueos del hilo principal y manteniendo un comportamiento estable del sistema.

7.1. Discusión de resultados

Al respecto del Experimento 1, la diferencia de tiempo entre el grupo de dos vecinos y un vecino se explica por el mecanismo utilizado para capturar el contexto del terreno. Cuando un *chunk* posee un mayor número de vecinos, *RuntimeWFC* debe recorrer una cantidad superior de información para reconstruir el contexto necesario para la generación. Debido a que los datos de cada *tile* se almacenan en listas que aumentan continuamente con el número de *chunks* generados, el costo asociado a esta búsqueda también incrementa progresivamente, produciendo la tendencia ascendente observada en la figura.

La causa de este comportamiento no está ligada a una limitación propia del

algoritmo de WFC, sino que es a causa de la implementación del sistema de almacenamiento de *chunks*, la cual tiene potencial de ser mejorada.

Al respecto de la generación de población, la diferencia se debe a que este proceso no depende de un recorrido extenso de listas, ya que la función de *RuntimeMapElites* se le da previamente la información necesaria de forma directa para colocar objetos. Por esto, el tiempo de ejecución depende principalmente de la cantidad de objetos que deben distribuirse.

Durante las pruebas, el método de instanciación de población solo hacia aparecer entre 2 a 6 objetos, y técnicamente la mutación no ejecutaba y no modificaba el esparcimiento de objetos de forma notable. No hubo una implementación adecuada de MAP-Elites.

En el experimento 2, las razones por la gran diferencia de tiempo entre LBS y *RuntimeLBS* se debe, nuevamente, a la cantidad de pasos que fueron saltados entre procesos. LBS contiene muchas conexiones con el editor de Unity, haciendo que este no genere de forma rápida, a comparación con *RuntimeLBS*. Tampoco LBS requiere de esta velocidad al momento de estar diseñando. Adecuándose al contexto, tener velocidades de generación bajas durante tiempo de ejecución es lo preferible para no afectar el gameplay.

Durante el desarrollo de este último experimento, se pudo observar que la implementación original de LBS ya dividía internamente un mapa de 50×50 en 25 partes antes de ejecutar WFC. Esto indica que la estrategia de subdividir el terreno para reducir la complejidad ya estaba presente en la herramienta original. No obstante, las mejoras observadas en *RuntimeLBS* provienen principalmente de la adaptación de los algoritmos para su ejecución en tiempo de juego, la eliminación de dependencias del editor y la reorganización del flujo de generación mediante tareas asíncronas.

7.2. Principales Aportes

La implementación de *RuntimeLBS* consiguió adaptar dos de los módulos principales de LBS para su utilización durante la ejecución del juego. A diferencia de la herramienta original, diseñada para operar exclusivamente dentro del editor de Unity, la propuesta permite generar terreno y población de forma dinámica mientras el usuario explora el escenario.

Los experimentos demostraron una reducción significativa en los tiempos de ge-

neración respecto de la implementación original. La generación de un escenario de 50×50 celdas disminuyó el tiempo desde aproximadamente 64 segundos hasta poco más de 3 segundos. Si bien parte de esta mejora se debe a la simplificación de ambos módulos de LBS, los resultados muestran que las modificaciones arquitectónicas introducidas permitieron reducir considerablemente el costo computacional y habilitar una ejecución compatible con escenarios de generación continua.

Los resultados obtenidos fueron consistentes con el objetivo planteado al inicio del proyecto y demostraron que la implementación propuesta supera ampliamente el rendimiento de la herramienta original en los escenarios evaluados.

7.3. Limitaciones y Trabajo Futuro

A pesar de los resultados obtenidos, la implementación desarrollada presenta algunas limitaciones. El algoritmo encargado de la población no corresponde ni siquiera a una simplificación del algoritmo genético utilizado por MAP-Elites. Asimismo, el costo computacional asociado a *RuntimeWFC* continúa aumentando conforme crece la cantidad de chunks generados, debido principalmente a las estructuras de datos utilizadas para reconstruir el contexto del terreno.

Como trabajo futuro resulta interesante optimizar el manejo de información de los chunks utilizados para construir el contexto de *RuntimeWFC*, reemplazando los recorridos lineales actuales, que se vuelven muy largos a medida pasa el tiempo, por mecanismos de acceso más eficientes que permitan mantener tiempos de generación constantes a medida que aumenta el tamaño del mundo generado.

Otra línea de investigación consiste en incorporar una implementación completa de los algoritmos genéticos de *MAP-Elites* compatibles con la ejecución en tiempo real, preservando tanto la diversidad de soluciones como el rendimiento alcanzado en esta investigación.

Finalmente, *RuntimeLBS* fue concebido como una extensión de la herramienta original y no como un reemplazo de esta. Por ello, una posible evolución del proyecto consiste en integrar los módulos desarrollados directamente dentro de LBS, permitiendo que los diseñadores puedan seleccionar entre una generación tradicional orientada al editor o una generación procedural en tiempo de ejecución, ampliando así el alcance de la herramienta hacia nuevos tipos de videojuegos, incluyendo aquellos con mundos dinámicos, extensos o potencialmente infinitos.

Esta investigación demuestra que es posible adaptar herramientas de generación procedural originalmente hechas para etapas de diseño hacia un modelo de ejecución durante el gameplay. Más allá de la reducción obtenida en los tiempos de generación, el principal aporte del trabajo consiste en demostrar una arquitectura capaz de ejecutar *WFC* y un algoritmo genético derivado de *MAP-Elites* de manera continua y estable dentro de Unity, abriendo la posibilidad de incorporar este tipo de técnicas en experiencias que requieren generación dinámica de contenido. De esta forma, *RuntimeLBS* constituye un primer paso hacia la evolución de LBS desde una herramienta de apoyo al diseño hacia una plataforma de generación procedural en tiempo de ejecución.

Glosario

Online: Ejecución de procesos o algoritmos durante el tiempo de juego (runtime), mientras el videojuego se encuentra en funcionamiento y el jugador interactúa con el sistema. En este caso, el término no hace referencia a conectividad de red, sino a la generación y modificación dinámica de contenido en tiempo real, sin interrumpir la experiencia del usuario. [14] [10]

Offline: Ejecución de procesos o algoritmos fuera del tiempo de juego, generalmente durante la fase de desarrollo o preproducción, como parte del entorno de edición o preparación del contenido. En este modo, los cálculos pueden realizarse sin restricciones estrictas de tiempo de respuesta, ya que no afectan directamente la ejecución interactiva del videojuego. [10]

Paralelismo: Técnica de optimización computacional que consiste en dividir una tarea en múltiples subprocesos que se ejecutan simultáneamente, ya sea en distintos núcleos de la CPU o en la GPU. En el contexto de la generación procedural de contenidos, el paralelismo se utiliza para reducir tiempos de cómputo y evitar bloqueos del sistema durante la ejecución de algoritmos complejos. [50]

WYSIWYG: Paradigma de diseño de interfaces en el cual la representación visual mostrada al usuario corresponde de forma directa al resultado final que se obtendrá en ejecución. En motores de videojuegos como Unity, este enfoque permite a los desarrolladores visualizar y manipular objetos, escenas y niveles en el editor de manera similar a como se verán durante el gameplay. [51] [31]

Chunk: Conjunto de cuadrillas conectadas entre sí. En LBS, se utiliza en sus interfaces para generar terreno y población.

NPC: Acronimo, significando ” *Non-Playable Character*”, el cual hace referencia a personajes dentro de un juego que no pueden ser controlados por el jugador.

Bibliografía

- [1] Nicolas A Barriga. A short introduction to procedural content generation algorithms for videogames. *International Journal on Artificial Intelligence Tools*, 2019.
- [2] Barbara De Kegel and Mads Haahr. Procedural puzzle generation: A survey. *IEEE Transactions on Games*, 12(1):21–40, 2019.
- [3] Alba Amato. Procedural content generation in the game industry. In *Game Dynamics: Best Practices in Procedural and Dynamic Game Content Generation*, pages 15–25. Springer, 2017.
- [4] Camila Aliaga, Cristian Vidal, Gabriel K Sepulveda, Nicolas Romero, Fernanda González, and Nicolas A Barriga. Level building sidekick: An ai-assisted level editor package for unity. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2023.
- [5] Noor Shaker, Julian Togelius, Mark J Nelson, Antonios Liapis, Gillian Smith, and Noor Shaker. Mixed-initiative content creation. *Procedural content generation in games*, pages 195–214, 2016.
- [6] Isaac Karth and Adam M Smith. Wavefunctioncollapse is constraint solving in the wild. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 2017.
- [7] Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.
- [8] Gabriel K Sepúlveda, Felipe Besoain, Sven von Brand, and Nicolas A Barriga. Biome generation tool: A mixed-initiative software for the procedural generation of biomes. *Applied Sciences*, 13(14):8070, 2023.

- [9] Sérgio Oliveira and Luís Magalhães. Adaptive content generation for games. In *2017 24^o Encontro Português de Computação Gráfica e Interação (EPCGI)*, pages 1–8. IEEE, 2017.
- [10] Mahdi Farrokhi Maleki and Richard Zhao. Procedural content generation in games: A survey with insights on emerging llm integration. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 20, pages 167–178, 2024.
- [11] Innes McKendrick. Continuous world generation in 'no man's sky'. In *Game Development Conference*, 2017.
- [12] Kniberg Henrik. Reinventing minecraft world generation. In *Jfokus Developers Conference*, 2022.
- [13] Lazaros Lazaridis and George F Fragulis. Creating a newer and improved procedural content generation (pcg) algorithm with minimal human intervention for computer gaming development. *Computers*, 13(11):304, 2024.
- [14] Julian Togelius, Alex J Champandard, Pier Luca Lanzi, Michael Mateas, Ana Paiva, Mike Preuss, and Kenneth O Stanley. Procedural content generation: Goals, challenges and actionable steps, 2013.
- [15] Trevor Ashby, Braden K Webb, Gregory Knapp, Jackson Searle, and Nancy Fulda. Personalized quest and dialogue generation in role-playing games: A knowledge graph-and language model-based approach. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–20, 2023.
- [16] Martin Dahrén. The usage of pcg techniques within different game genres, 2021.
- [17] Shuyue Zhang, Zibo Xu, Zhenyuan Liu, and Minghe Liu. Noise algorithms in game terrain generation. In *2025 10th International Conference on Image, Vision and Computing (ICIVC)*, pages 611–618. IEEE, 2025.
- [18] Casey D Majhor and Jeremy P Bos. Multifractal terrain generation for evaluating autonomous off-road ground vehicles. *Journal of Autonomous Vehicles and Systems*, 5(2):021003, 2025.

- [19] Goanghun Kim and Nakhoon Baek. A height-map based terrain rendering with tessellation hardware. In *2014 International Conference on IT Convergence and Security (ICITCS)*, pages 1–4. IEEE, 2014.
- [20] Petros Tzathas, Boris Gailleton, Philippe Steer, and Guillaume Cordonnier. Physically-based analytical erosion for fast terrain generation. In *Computer Graphics Forum*, volume 43, page e15033. Wiley Online Library, 2024.
- [21] Jacob Olsen. Realtime procedural terrain generation. 2004.
- [22] Niko Sainio. Terrain generation algorithms: evaluation of procedural methods. <https://trepo.tuni.fi/handle/10024/147549>, 2023. Master’s thesis, Tampere University.
- [23] Gustav Johansson. Pcg in game bits and its effect on player behaviour, 2020.
- [24] Gorm Lai, William Latham, and Frederic Fol Leymarie. Towards friendly mixed initiative procedural content generation: Three pillars of industry. In *Proceedings of the 15th International Conference on the Foundations of Digital Games*, pages 1–4, 2020.
- [25] Thomas Bartz-Beielstein, Jürgen Branke, Jörn Mehnen, and Olaf Mersmann. Evolutionary algorithms. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 4(3):178–195, 2014.
- [26] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.
- [27] Stephanie Forrest. Genetic algorithms. *ACM computing surveys (CSUR)*, 28(1):77–80, 1996.
- [28] Michael Emmerich, Ofer M Shir, and Hao Wang. Evolution strategies. In *Handbook of heuristics*, pages 89–123. Springer, 2025.
- [29] Yu Qin, Zhiliang Qin, Zhongkai Zhang, Yingying Li, and Qidong Lu. Evolutionary programming: A population-based optimization algorithm for coded multiuser systems. *J. Commun.*, 16(9):369–378, 2021.
- [30] Jason Gregory. *Game engine architecture*. AK Peters/CRC Press, 2018.

- [31] Cambridge dictionary. WYSIWYG. <https://dictionary.cambridge.org/es/diccionario/ingles/wysiwyg>, 2025.
- [32] Unity Technologies. Gameobjects. <https://docs.unity3d.com/6000.0/Documentation/ScriptReference/GameObject.html>, 2025.
- [33] Unity Technologies. An introduction to prefabs. <https://docs.unity3d.com/6000.0/Documentation/Manual/prefabs-introduction.html>, 2025.
- [34] Epic Games. Unreal engine 5 documentation. <https://dev.epicgames.com/documentation/es-es/unreal-engine/unreal-engine-5-7-documentation>, 2025.
- [35] OKAM Studios. Godot documentation. <https://docs.godotengine.org/es/4.x/index.html>, 2025.
- [36] Crytek. Cryengine documentation. <https://www.cryengine.com/docs/static/engines/cryengine-5/categories/23756816>, 2025.
- [37] Gillian Smith, Jim Whitehead, and Michael Mateas. Tanagra: A mixed-initiative level design tool. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, pages 209–216, 2010.
- [38] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. Sentient world: Human-based procedural cartography: An experiment in interactive sketching and iterative refining. In *International Conference on Evolutionary and Biologically Inspired Music and Art*, pages 180–191. Springer, 2013.
- [39] Noor Shaker, Mohammad Shaker, and Julian Togelius. Ropossum: An authoring tool for designing, optimizing and solving cut the rope levels. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 9, pages 215–216, 2013.
- [40] Tiago Lemos de Araujo Machado. *Cicero-An AI-Assisted Game Design System*. PhD thesis, New York University Tandon School of Engineering, 2019.
- [41] Procedural Worlds Inc. Gaia pro vs. <https://assetstore.unity.com/packages/tools/terrain/>

- gaia-pro-vs-terrain-trees-grass-water-for-unity-6-263149?aid=11011SqC, 2025.
- [42] LLC World Machine Software. World machine. <https://www.world-machine.com/>, 2024.
- [43] Voxel Farm. Voxel farm indie. <https://www.voxelfarm.com/indie.html>, 2019.
- [44] Procedural Worlds Inc. Gena pro - level designer, villages, roads rivers for unity 6. <https://assetstore.unity.com/packages/tools/terrain/gena-pro-level-designer-villages-roads-rivers-for-unity-6-183186?aid=11011SqC>, 2025.
- [45] Code Respawn. Dungeon architect. <https://assetstore.unity.com/packages/tools/utilities/dungeon-architect-53895?aid=11011iTFw&pubref=website>, 2024.
- [46] Antonios Liapis, Georgios N Yannakakis, and Julian Togelius. Sentient sketchbook: computer-assisted game level authoring. 2013.
- [47] Joel Lehman and Kenneth O Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19(2):189–223, 2011.
- [48] Troglobytes. Daedalus. <https://troglobytes.itch.io/daedalus>, 2017.
- [49] Apparance Studios Limited. Apparance: Runtime procedural tooling. <https://www.apparance.uk/>, 2025.
- [50] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing*, volume 110. Benjamin/Cummings Redwood City, CA, 1994.
- [51] Unity Technologies. Unity Glossary. <https://docs.unity3d.com/6000.0/Documentation/Manual/Glossary.html#SectionUI>, 2025.

A. Capturas

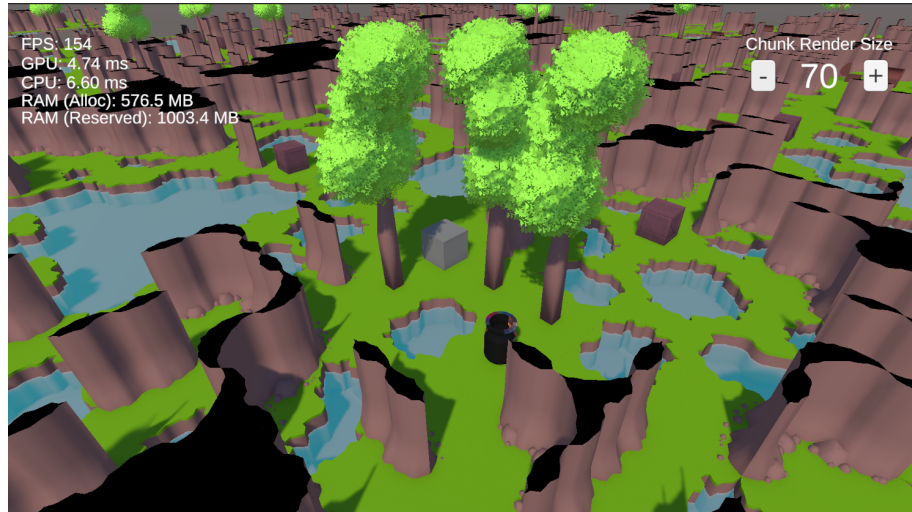


Figura A.1: Captura de generación de terreno y población en tiempo de juego.

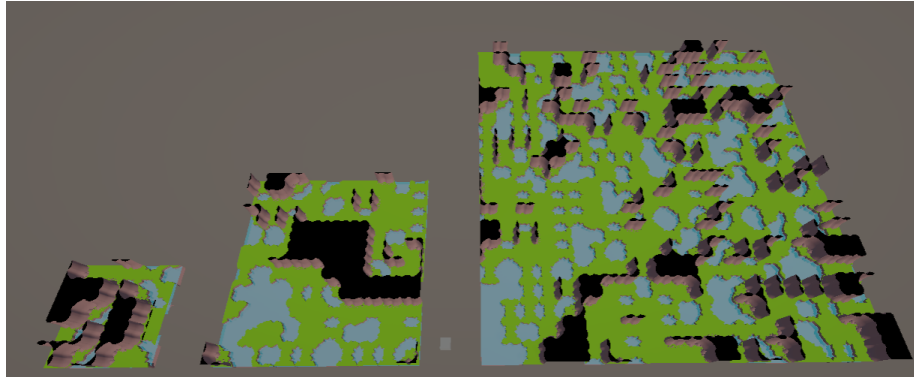


Figura A.2: Distintos tamaños de *chunks* para el experimento 1. Las medidas son, de izquierda a derecha: **5x5**, **10x10** y **20x20**.

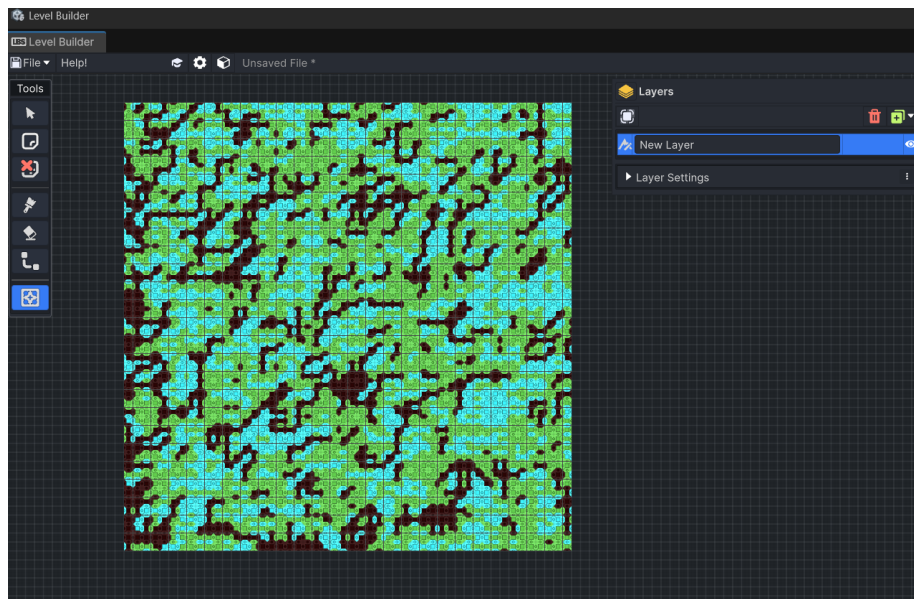


Figura A.3: Interfaz de LBS, donde se está generando un *chunk* de **50x50**

B. Pseudocódigo

Algoritmo B.1 Generación de terreno mediante *RuntimeWFC*

- 1: AddEmptyTiles()
 - 2: CaptureNeighboringChunks()
 - 3: $chunk \leftarrow \text{ExecuteRuntimeWFC}()$
 - 4: CaptureChunkForInstance()
 - 5: **return** $chunk$
-

Algoritmo B.2 Generación de población mediante *RuntimeMAPElites*

- 1: ReceiveGeneratedChunk() $\leftarrow chunk$
 - 2: RandomizeObjectPositions()
 - 3: ExecuteRandomPos()
 - 4: **return** $PopulationLayer$
-

C. Diagramas

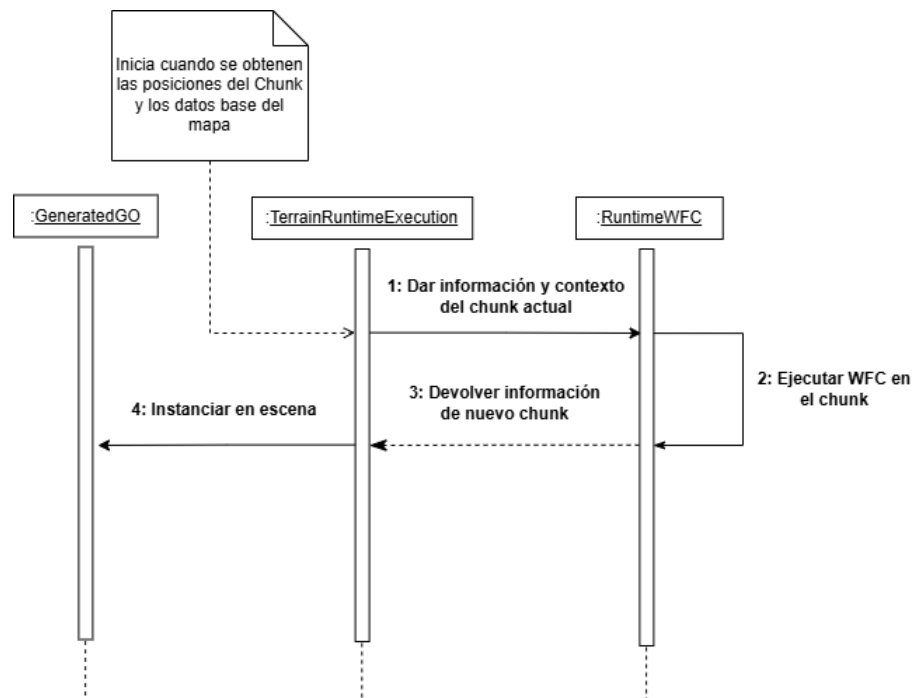


Figura C.1: Caso de Uso 1: Instanciación de Chunk de terreno en escena de Unity durante ejecución.

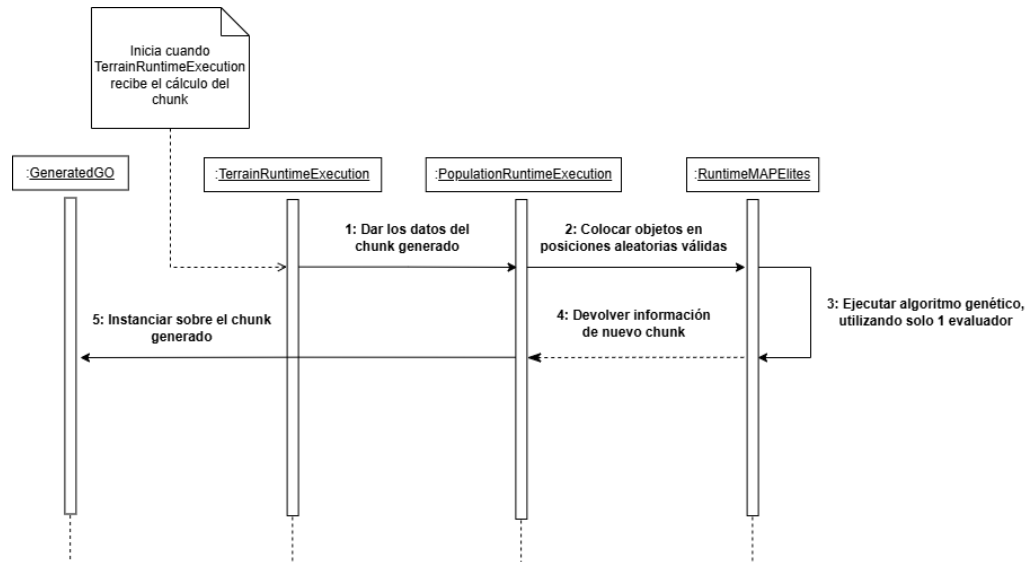


Figura C.2: Caso de Uso 2: Instanciación de Chunk de población en escena de Unity durante ejecución en base al chunk ya instanciado del módulo de terreno.

D. Gráficos y figuras

Tamaño	Cantidad de Chunks	Cantidad total de tiles
5x5	1067	26675
10x10	309	30900
20x20	82	32800

Tabla D.1: Cantidad de chunks generados en el transcurso de la evaluación.

Tamaño	Cantidad de tiles	Tiempo Total Promedio	T. de tile individual
5x5	25	167.17ms	6.68ms
10x10	100	694.96ms	6.94ms
20x20	400	3637.85ms	9.04ms

Tabla D.2: Tiempos promedio de generación de un solo chunk, obtenido de la suma de todos los tiempos de las funciones.

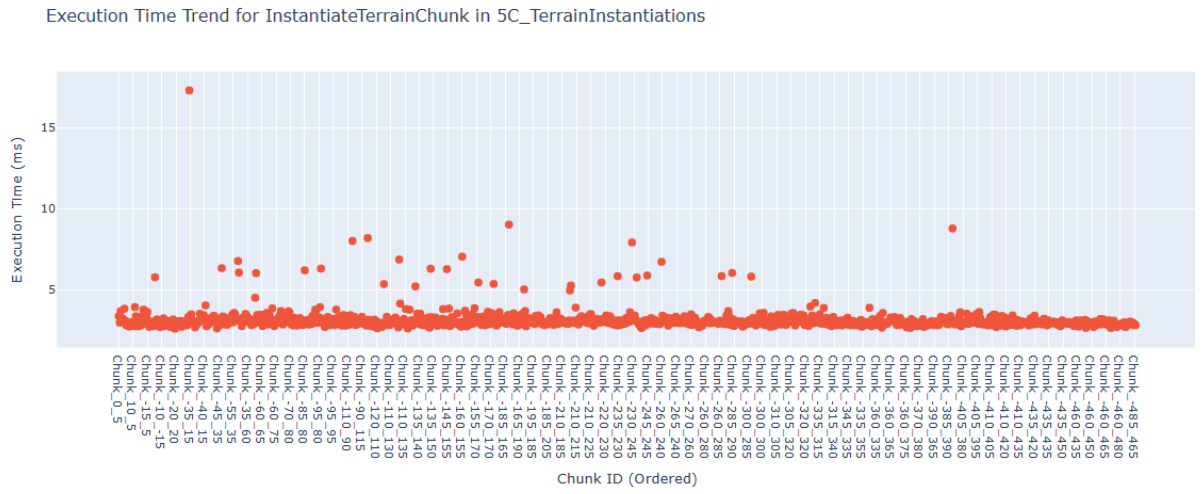


Figura D.1: Gráfico de líneas que muestra el tiempo que demoró la instanciación de terreno por cada chunk individual, de tamaño 5x5.

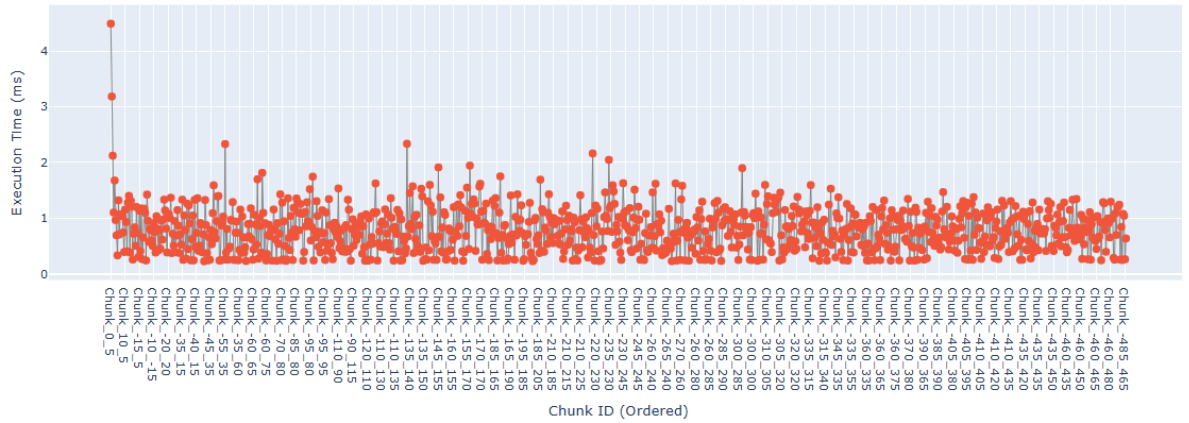


Figura D.2: Gráfico de líneas que muestra el tiempo que demoró la instanciación de población por cada chunk individual, de tamaño 5x5.

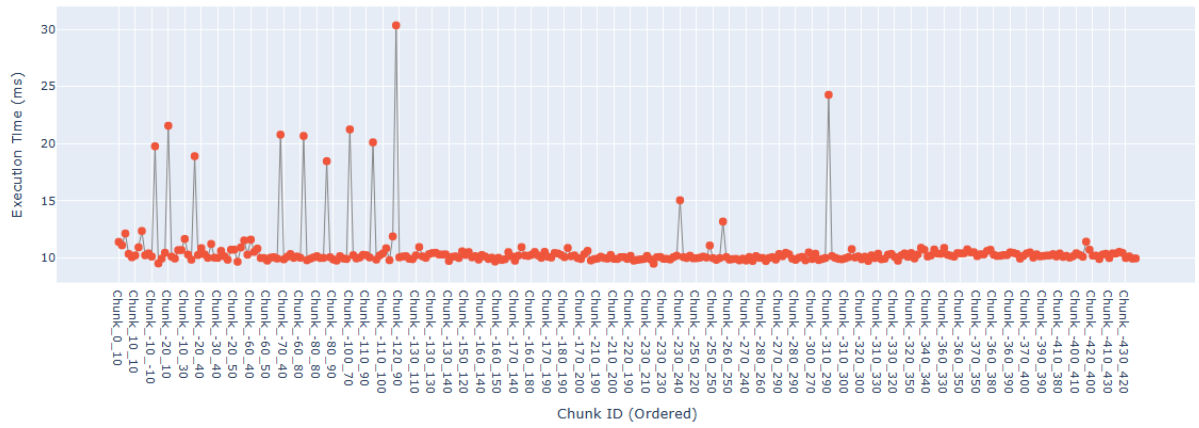


Figura D.3: Gráfico de líneas que muestra el tiempo que demoró la instanciación de terreno por cada chunk individual, de tamaño 10x10.

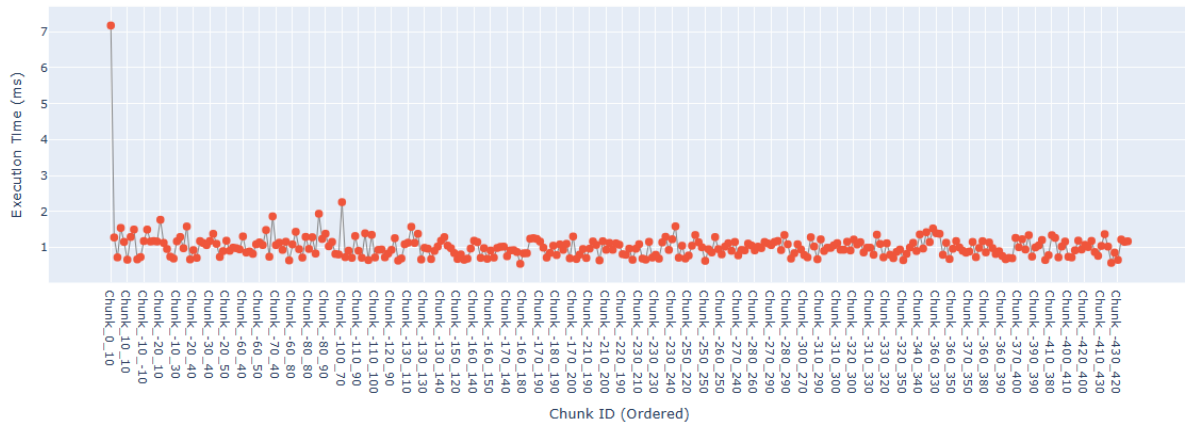


Figura D.4: Gráfico de líneas que muestra el tiempo que demoró la instanciación de población por cada chunk individual, de tamaño 10x10.

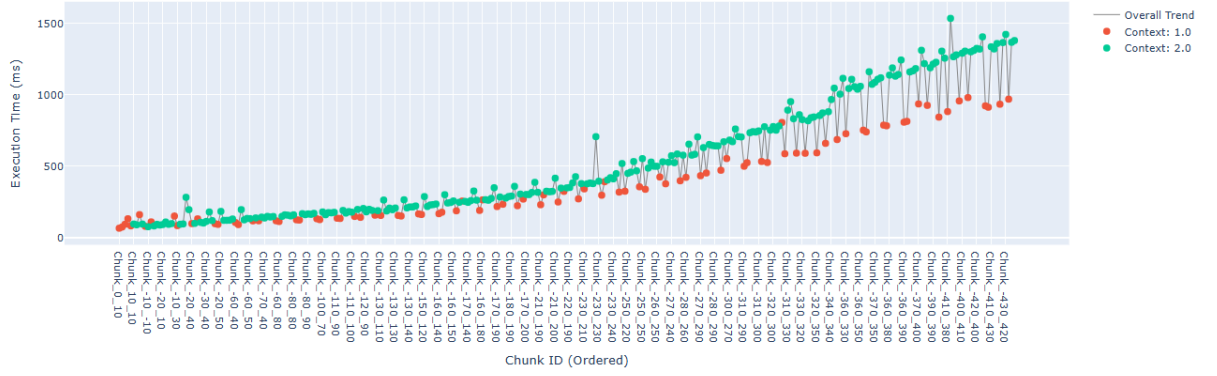


Figura D.5: Gráfico de líneas que muestra el tiempo de cálculo de *RuntimeWFC* por cada chunk individual, de tamaño 10x10.

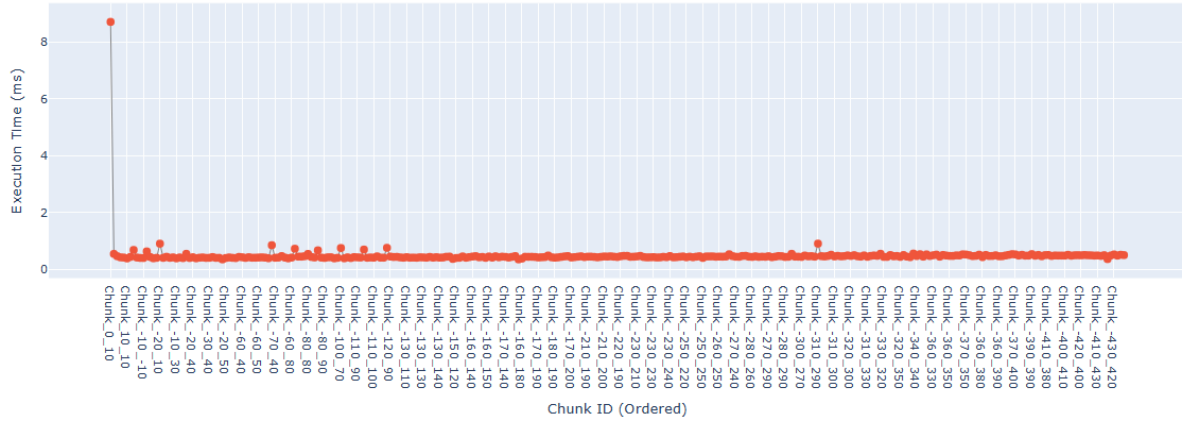


Figura D.6: Gráfico de líneas que muestra el tiempo de cálculo de *RuntimeMapElites* por cada chunk individual, de tamaño 10x10.

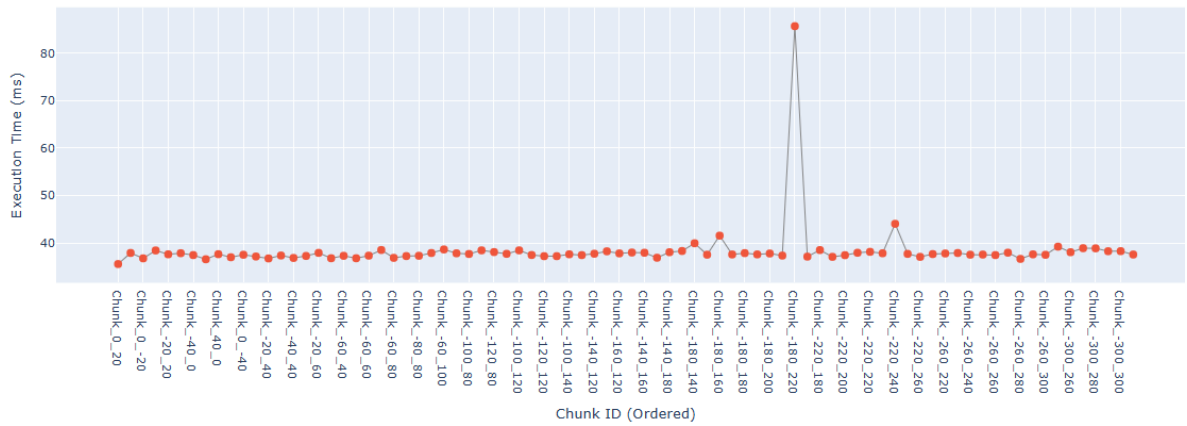


Figura D.7: Gráfico de líneas que muestra el tiempo que demoró la instanciación de terreno por cada chunk individual, de tamaño 20x20.

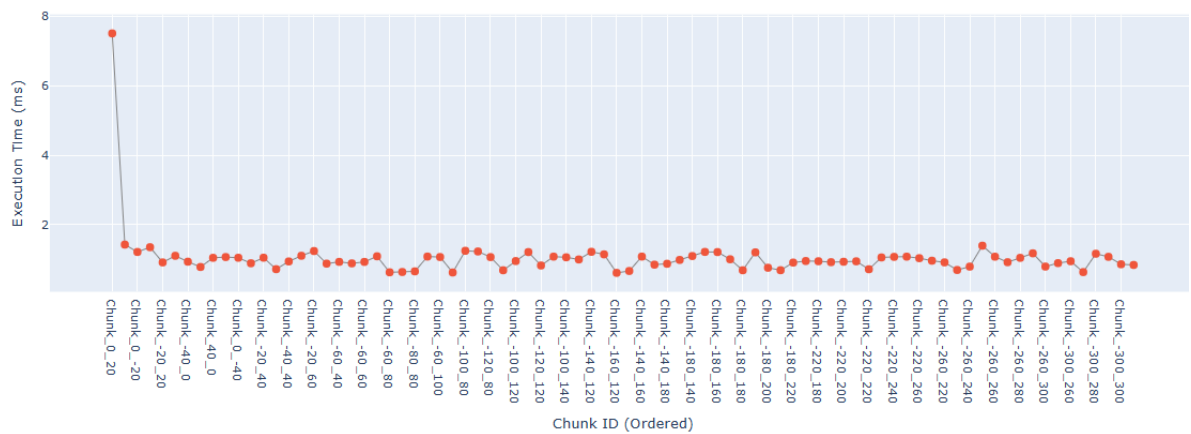


Figura D.8: Gráfico de líneas que muestra el tiempo que demoró la instanciación de población por cada chunk individual, de tamaño 20x20.

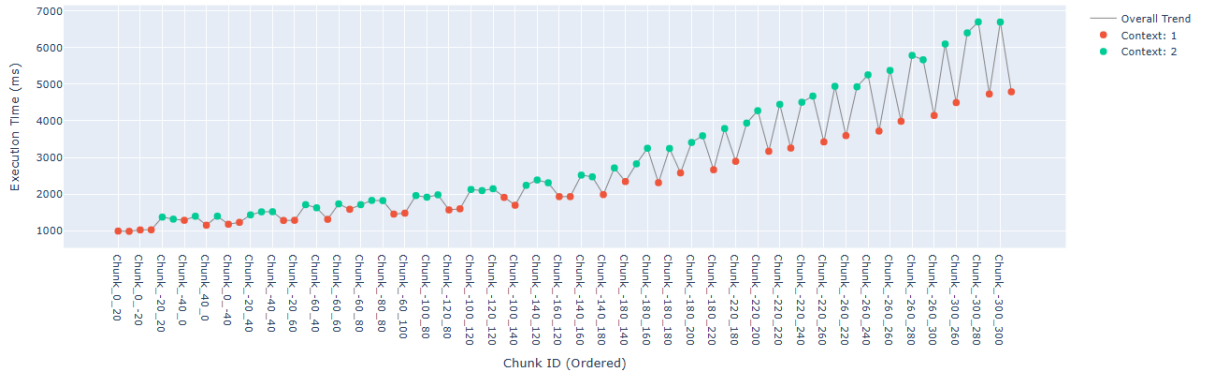


Figura D.9: Gráfico de líneas que muestra el tiempo de cálculo de *RuntimeWFC* por cada chunk individual, de tamaño 20x20.

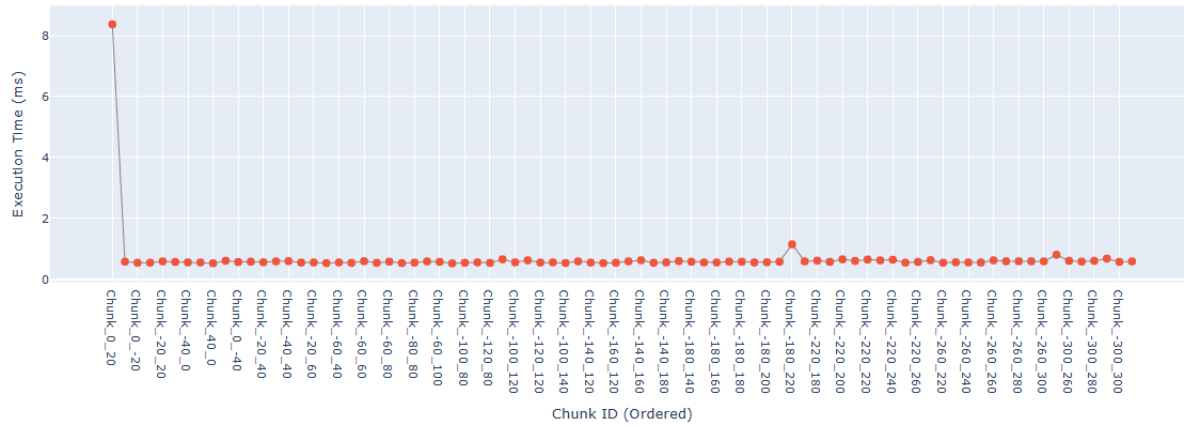


Figura D.10: Gráfico de líneas que muestra el tiempo de cálculo de *RuntimeMapElites* por cada chunk individual, de tamaño 20x20.