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

Rendimiento del algoritmo Hierarchical Wave Function Collapse en C++

JORGE SEBASTIÁN LEIVA IBÁÑEZ

Profesor Guía: NICOLÁS ARTURO BARRIGA RICHARDS

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

Talca – Chile
Julio, 2024

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

Rendimiento del algoritmo Hierarchical Wave Function Collapse en C++

JORGE SEBASTIÁN LEIVA IBÁÑEZ

Profesor Guía: NICOLÁS ARTURO BARRIGA RICHARDS

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

El presente documento fue calificado con nota: _____

Talca – Chile

Julio, 2024

Dedicado a mi madre, Alicia Ester Ibáñez Schulz, por ser mi sustento y apoyo constante en todo mi proceso universitario, por siempre creer en mi capacidad de lograr mis objetivos y de darme ánimos siempre que lo necesite.

AGRADECIMIENTOS

Agradecimientos al Fondo Nacional de Desarrollo Científico y Tecnológico, Fondecyt de Iniciación 11220438.

TABLA DE CONTENIDOS

página

ÍNDICE DE FIGURAS

página

ÍNDICE DE TABLAS

página

RESUMEN

El “Hierarchical Wave Function Collapse” (HWFC) es un algoritmo para la generación de mapas en videojuegos de forma procedural, creado originalmente en Python, un lenguaje de programación dinámico que presenta un problema importante. El algoritmo requiere de una cantidad considerable de tiempo para la generación de un mapa.

Para reducir el tiempo requerido en cada generación, se ha hecho una implementación de HWFC en el lenguaje de programación estático C++. Los resultados obtenidos presentan un incremento significativo en la velocidad con la que cada mapa es generado con respecto a la implementación original en Python. Esto representa una ventaja destacable para su uso en la creación de mapas en videojuegos, pues puede representar una reducción considerable en el tiempo de generación requerido para iniciar un nivel único y nuevo en cada sesión de juego.

Esta implementación, al estar hecha en C++, podrá ser integrada de forma relativamente fácil como una extensión en el motor gráfico “Unreal Engine 5”, otorgando una herramienta útil para la creación y diseño procedural de mapas en videojuegos.

1. Introducción

En la historia del entretenimiento, pocas veces se ha visto un crecimiento tan grande como el de los videojuegos, siendo una industria que ha presentado una masificación constante, la cual es reflejada en la producción contínua de nuevas propuestas e ideas [1]. En la actualidad existe una inmensa variedad de videojuegos, desde los videojuegos de ritmo/musicales como "Just Dance" [2] que incentivan el ejercicio, hasta los mundos virtuales que fomentan la exploración como la saga "Fallout" [3] o "Red Dead Redemption" [4].

La producción de la industria, junto con la demanda continua y cada vez mayor por parte de los usuarios por experiencias nuevas e innovadoras, ha dado lugar a una producción inmensa de contenido que ha ido aumentando con el paso del tiempo, ya sean texturas, nuevos objetos, música, niveles, etc [5]. Dicho contenido requiere de una inmensa cantidad de esfuerzo, mano de obra y recursos de la empresa por periodos que pueden extenderse por años, requiriendo a su vez de un esquema muy bien planificado y estructurado para cada uno de sus proyectos. Debido a que cada elemento dentro de un videojuego debe ser planificado, modelado, programado, implementado y probado, para asegurar su funcionamiento e interacción con los otros elementos presentes en el videojuego, se requiere de personal que realice todas estas tareas, lo que a su vez lleva a un consumo de recursos y un tiempo de producción capaz de abrumar a su desarrolladora [6].

Una de las soluciones más prometedoras a este problema es el contenido generado de forma procedural o "Procedural Content Generation" (PCG) [7], el cual refiere a la creación de nuevos elementos de forma dinámica mediante el uso de algoritmos, reglas predefinidas y valores ajustables mediante parámetros por los diseñadores para adaptarse al contexto requerido. El uso de estas técnicas ha permitido mejorar la

eficiencia en el empleo de recursos, incrementar la cantidad y variedad de contenido, reducir el almacenamiento requerido, etc [8].

Desde hace tiempo, el PCG ha estado demostrando su relevancia y eficacia en la industria, desde el armamento generado de forma aleatoria en la saga "Borderlands" [9] al derrotar a un enemigo o la creación de un mapa con una distribución nueva en cada partida, como "Minecraft" [10]. Si bien, este PCG comparte el hecho de que va cambiando en cada nueva iteración, la forma en que se genera presenta una gran cantidad de matices, desde el uso de diversos algoritmos con diferentes grados de complejidad, hasta las implementaciones más sencillas que se limitan al uso de valores aleatorios para reorganizar el contenido ya existente [8].

Un ejemplo particularmente interesante para la creación de contenido en los videojuegos es el algoritmo "Wave Function Collapse" (WFC) [11], el cual destaca por su eficiencia en la creación procedural de mapas dado su funcionamiento, aunque no se encuentra limitado únicamente a esto. Este algoritmo toma como base ejemplos del resultado esperado entregados por el diseñador, como una o más imágenes que muestren los posibles elementos que pueden aparecer en un nivel (tierra, mar, caminos, objetos, etc.). Cada ejemplo es descompuesto en pequeños patrones de un tamaño fijo de la unidad mínima que conforme el ejemplo (píxeles en el caso de una imagen), los cuales representen las relaciones entre los elementos de la imagen original, estableciendo un criterio base de cómo pueden o no pueden ser colocados. Un nuevo mapa vacío o "sin colapsar" es creado y es rellenado de forma iterativa usando los patrones obtenidos del ejemplo de entrada [11]. Aunque este algoritmo resulta una propuesta muy interesante, no está exento de problemas, principalmente por la falta de control y criterio complejo a la hora de generar contenido. Dados los problemas que presentaba el WFC, se crearon versiones mejoradas del mismo que buscaban solucionar dichos problemas [12].

Una de las mejoras realizadas sobre WFC más destacables es el algoritmo HWFC [13], el cual se destaca por añadir el concepto de "jerarquías", añadiendo un orden de prioridad con base en el tamaño de los patrones de mayor a menor. La implementación original de este algoritmo fue realizada en Python [14], un lenguaje de programación de tipo dinámico. Los lenguajes dinámicos se caracterizan por tener variables que pueden cambiar según el contexto, asignar de forma dinámica la memoria e interpretar línea por línea durante la ejecución, lo que se traduce en mayor facilidad en su uso, más flexibilidad y mejor interactividad, ofreciendo una mejor comodidad a la

hora de ser adaptados, facilitando un desarrollo o experimentación rápida. Lamentablemente, estos beneficios en los lenguajes dinámicos también presentan contras, como los errores por tipos de variables incompatibles, comportamientos inesperados durante la ejecución y, principalmente, una menor optimización y rendimiento en términos de costo computacional a comparación de un lenguaje de tipo estático [15].

Como HWFC es un algoritmo para PCG [14], el tiempo de cada ejecución es un factor importante para tener en cuenta, pues suma el tiempo y costo de generación a todos los demás procesos que se estén realizando, incrementando el tiempo de toda la ejecución y ralentizando otros procesos simultáneos. Para solucionar esto, se propone hacer una implementación del algoritmo HWFC siguiendo los lineamientos presentados en el artículo de la implementación original "Hierarchical WaveFunction Collapse" [13], haciendo uso del lenguaje de programación estático de C++. Este presenta las ventajas de un mejor control en el uso de la memoria disponible y un mejor rendimiento general dado que los procesos de verificación y optimización del código se realizan durante la compilación previa a la ejecución, a diferencia de un lenguaje de programación dinámico como Python.

A continuación, se listarán los objetivos específicos y la hipótesis que conformarán este proyecto. Se hará un repaso por el estado del arte relevante referente a la creación de contenido mediante PCG, el uso del algoritmo WFC y sus variantes, el uso de una metodología de desarrollo de software, los experimentos a realizar para medir la eficiencia y los resultados obtenidos, para finalmente presentar las conclusiones obtenidas.

2. Objetivos

2.1. Objetivo general

Replicar y evaluar el rendimiento de una implementación de HWFC en el lenguaje de programación estático C++ respecto a su implementación original en el lenguaje de programación dinámico Python.

2.1.1. Objetivos específicos

Los puntos claves a realizar para el proyecto son:

1. Investigar el estado del arte referente a implementaciones basadas en el algoritmo WFC para indagar en consideraciones sobre su funcionamiento, criterios adicionales en selección de zonas de colapso, ahorro de recursos, obtención de patrones, implementación de técnicas algorítmicas, etc. para poder mejorar la eficiencia y/o el resultado final de la generación.

2. Investigar el estado del arte referente al uso de lenguajes de tipo estático y dinámico en el desarrollo de videojuegos, y profundizar las diferencias entre el funcionamiento de ambos para poder respaldar y comprender las diferencias fundamentales entre sus rendimientos, junto con sus fortalezas y debilidades de cara a su uso dentro de videojuegos.

3. Replicar el funcionamiento y los resultados del algoritmo HWFC en C++ con el objetivo de mostrar que la implementación realizada en esta memoria es capaz de obtener resultados similares en la generación de mapas de forma procedural.

4. Recopilar información del rendimiento de las ejecuciones de la implementación en C++ y obtener métricas de rendimiento con las que contrastar con los resultados de las métricas obtenidas de HWFC en Python.

5. Comparar y analizar las diferencias entre los resultados obtenidos entre ambas implementaciones.

2.1.2. Pregunta de investigación

¿Es posible obtener una mejora significativa en el tiempo de generación de mapas de los algoritmos WFC, MWFC y HWFC al usar un lenguaje de programación estático?

2.1.3. Hipótesis

El uso de un lenguaje estático como C++ ofrecerá un mejor rendimiento general al verificar y optimizar en tiempo de compilación, lo cual resultará en un tiempo de generación de mapas considerablemente menor a la implementación original del algoritmo HWFC realizada en Python.

3. Estado del arte

El uso de PGC en videojuegos es sumamente variado y contempla múltiples acercamientos e implementaciones que pueden ser más o menos eficientes en términos de costo computacional a la hora de generar un tipo de contenido en particular, como los niveles, mapas, objetos, misiones, música, etc [8]. Sumado a esto, también se debe considerar la calidad y cohesión del contenido generado, el balance/dificultad y/o la repetitividad del contenido generado, por lo que es necesario evaluar cuáles algoritmos son los ideales para implementar en cada situación.

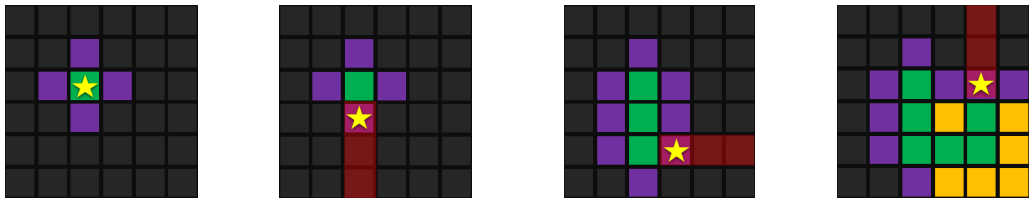
Entre los géneros de videojuegos que más destacan por hacer un uso fundamental del PCG se encuentran los "rogue-like", los cuales son juegos inspirados en la obra "Dungeon and Dragons", centrados en la exploración de mazmorras (niveles/mapas) generadas de forma procedural. Generalmente el jugador inicia en un área desprovista de peligros donde se le presentan varias posibles rutas que llevan a nuevas áreas. Cada vez que se avance, el jugador se puede encontrar con enemigos, recompensas especiales, personajes no jugables (NPC), etc., garantizando que cada partida sea de cierta forma única [16]. Las áreas pueden ser generadas sobre la marcha cuando el usuario decida tomar una ruta en caso de que los niveles sean simples, o bien, en caso de que el nivel tenga muchos elementos, requieran un alto grado de coherencia o tengan un alto grado de complejidad en su generación. Todo el nivel puede ser generado previamente al inicio de la partida, limitando al sistema a cargar únicamente el contenido generado cuando está en pantalla. Si bien no hay un algoritmo estándar para la creación de PCG, hay varios que salen a destacar en este tipo de tareas para la generación de mapas.

3.1. Algoritmos usados en la generación de niveles

Entre las múltiples aproximaciones que se pueden encontrar en el uso de PCG, existen algoritmos notables por su eficiencia, simplicidad y/o un tiempo de ejecución relativamente corto, lo cual los vuelve unos buenos candidatos para usar como base de la generación o implementarlos en diferentes puntos de otra generación para mejorar los resultados obtenidos [17].

3.1.1. Algoritmo Prim

El algoritmo de Prim [18] es un buen ejemplo de una implementación sencilla para la creación de mapas de tipo laberinto, donde todas las celdas estén conectadas de forma eficiente y sin ciclos redundantes, como se puede apreciar en la Figura 3.1. Se trata de un algoritmo para encontrar el árbol de expansión mínima, el cual es un subconjunto de las aristas de un grafo que conecta todos los vértices con el menor costo, sin formar ciclos y con la conectividad garantizada. Su funcionamiento elimina las posibilidades de celdas aisladas y requiere de pocos recursos computacionales, pero presenta la desventaja de que puede producir patrones de mapas repetitivos y con una complejidad pobre comparada a otros métodos de generación de mapas.

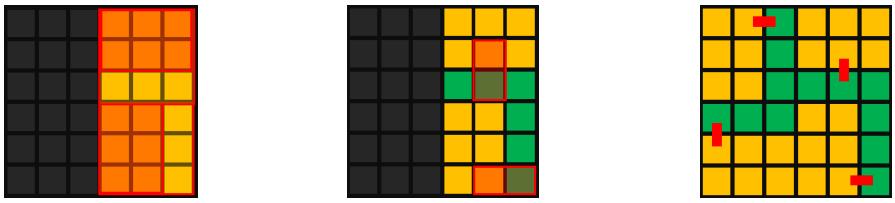


(a) Selección. (b) Verificación. (c) Expansión. (d) Eliminación.

Figura 3.1: Se selecciona una posición aleatoria para iniciar, marcándola como "camino" (a) y se añaden todas las casillas adyacentes a una lista de "posibles paredes", para luego iterar por cada una que haya en la lista, seleccionando una de estas y verificando si las casillas opuestas ya fueron visitadas (b). Si tiene al menos una casilla opuesta sin visitar, se convierte en un camino parte del laberinto, para luego añadir las nuevas paredes adyacentes a la lista (c). Si ambas celdas divididas por la pared ya están en el laberinto, se elimina la pared de la lista para evitar ciclos (d), terminando cuando todas las celdas hayan sido visitadas.

3.1.2. Algoritmo BSP

La partición binaria del espacio [17] es una técnica de división recursiva del mapa en subáreas de menor tamaño, la cual destaca en la creación de mazmorras, dividiendo el área en "habitaciones" y "pasillos" de manera jerárquica y organizada, como se puede apreciar en la Figura 3.2, lo que facilita la generación del nivel, reduce el gasto computacional y aumenta la viabilidad de la navegación en el mismo. Su implementación otorga una buena flexibilidad a la hora de crear mapas estructurados con diferentes tamaños, sumado a que garantiza la conectividad y lógica de los elementos creados. Sin embargo, este algoritmo requiere de un buen control a la hora de asignar el espacio suficiente a cada habitación y pasillo para garantizar que puedan cumplir con su función. Sin mejoras adicionales que proporcionen suficiente aleatoriedad, puede presentar una alta uniformidad, un rasgo favorable para la creación de mazmorras o entornos estructurados, pero que es desfavorable en la creación de entornos naturales, biomas o donde se busca una gran variedad de los elementos en el mapa.



(a) División del mapa. (b) Definición de habitaciones, pasillos y conexiones. (c) Mapa completado.

Figura 3.2: Se inicia dividiendo un mapa en subáreas, que a su vez se vuelven a subdividir de forma recursiva hasta el tamaño deseado o región final (a), creando una estructura de árbol binario donde cada nodo representa una división y cada hoja representa una región final del mapa. En las regiones finales, se designa a los grupos de casillas más grandes como habitaciones y a las más pequeñas como pasillos, designando puntos donde ambos se conecten (b). Esto se repite por cada subárea hasta completar el mapa.

3.1.3. Algoritmos de crecimiento

Para casos donde se necesita de algo más "natural", como podría ser la creación de cuevas y/o algunos tipos de biomas, existen múltiples aproximaciones, como los algoritmos de crecimiento o "Growth Algorithm" (GA) [19], los cuales buscan simular el crecimiento natural de estructuras, vegetación o terrenos aleatorios. Entre los algoritmos de este tipo podemos encontrar ejemplos como la generación de mapas por Autómatas Celulares (AC) [17], que consiste en sistemas de casillas, donde se tiene una lista de estados para las casillas y una lista de reglas de transición entre los estados (reglas de evolución). Su funcionamiento inicia en un punto aleatorio donde, por medio de reglas de evolución, se definirá qué estado afectará a otros y cómo irán cambiando las casillas vecinas de forma progresiva a medida que avance la generación, hasta obtener un mapa estable o hasta cumplir con un parámetro que indique el fin de la generación.

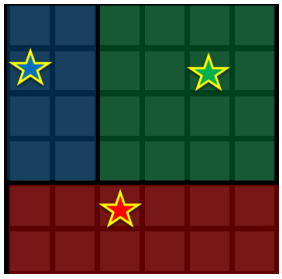
A su vez, la generación basada en "agentes" es una técnica muy similar que también puede dar resultados de carácter orgánico al igual que el AC o bien dar resultados más estructurados, presentando un mayor grado de versatilidad, permitiendo que sea usado para generar desde cuevas, pasando por bosques, hasta ciudades que requieran de mapas variados, estructurados y complejos. Esta técnica implica el uso de entidades autónomas (agentes), las cuales siguen una lista de reglas y comportamientos que les permiten modificar el mapa progresivamente, existiendo diferentes tipos para cumplir diversas tareas [17].

Si se busca una alta complejidad en la generación, tanto los agentes como los AC pueden ser perfectamente combinados con otras técnicas u algoritmos de generación para obtener mejores resultados. Por ejemplo, un mapa base creado con BSP permite la creación y distribución de áreas en el mapa, para posteriormente asignar un agente o un AC que se dedique únicamente a colocar decoraciones, objetos, enemigos o establecer caminos adicionales para obtener resultados que presenten variedad, sin sacrificar la fiabilidad y eficiencia del algoritmo base.

3.1.4. Algoritmos complementarios

Un claro ejemplo de una técnica que muestra su esplendor al ser combinada con otros algoritmos de generación es el Diagrama de Voronoi [20], el cual consiste en la implementación de puntos de generación (múltiples casillas iniciales) en un

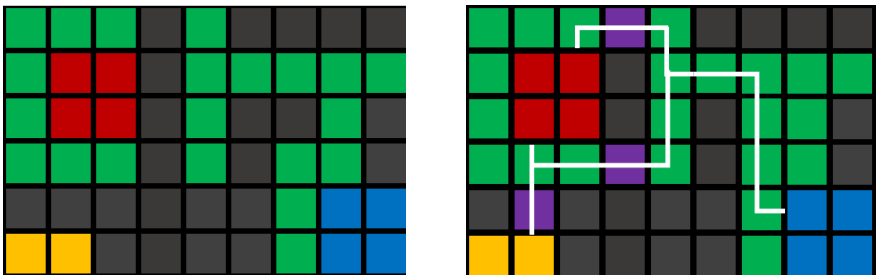
mapa para crear “zonas” las cuales se distribuyen alrededor del mapa y cuentan con una región asociada designada por la cercanía de las celdas de Voronoi (casillas adyacentes) al generador más próximo, como se representa en la Figura 3.3, dando a cada generador elementos diferenciados con los que montar un área, siendo muy útil para crear diferentes biomas o regiones, ya que cada casilla puede ser diferenciada de otras en cuanto a qué facción puede ocuparla, qué tipo de recompensas se pueden dar, que tipos de NPC pueden aparecer en esas casillas, etc. Incluso se pueden asignar diferentes tipos de generadores según se acomode a un área en particular, dando un gran control al diseñador para crear un entorno coherente, definir la cantidad de recursos disponibles o ajustar la dificultad en el nivel generado. Adicionalmente, la diferenciación permite incorporar mecánicas en torno a la diferencia de biomas como podrían ser los efectos del clima, una historia creada con base en la conquista de zonas por facciones, la generación de misiones basándose en los elementos colocados en cada área, etc.



(a) Mapa simple con Diagrama Voronoi.

Figura 3.3: Distribución de 3 zonas diferentes, cada una con un punto de generación independiente.

También los algoritmos de búsqueda como A* [21], el cual normalmente es usado para encontrar el camino más corto desde un punto inicial hasta un punto final, pueden ser usados para complementar un mapa, específicamente mapas que requieran de una red de caminos funcionales, como se muestra en la Figura 3.4. El objetivo en este caso es establecer rutas viables para las entidades que recorrerán el nivel, pero, dado que estás características pueden resultar demasiado específicas, es mucho más factible incorporar su uso a otro algoritmo base. Por ejemplo, aplicando A* en agentes que recorran un mapa previamente generado para asegurar que cuenta con rutas viables para los NPC y garantizar que el nivel es viable dentro de las mecánicas del videojuego [22].



(a) Mapa completo con rutas no viables.

(b) Mapa modificado por A*.

Figura 3.4: La imagen (a) muestra un mapa completo, pero que no cuenta con ninguna ruta navegable. En este tipo de situaciones se puede hacer uso de A*, dando como resultado un mapa con modificaciones (b) donde las zonas alteradas se muestran en morado, asegurando la viabilidad del mapa.

3.1.5. Machine Learning

Si bien es posible obtener generaciones de alta calidad por medio de algoritmos que se basan en usar reglas y restricciones para progresar, puede resultar difícil a la hora de obtener resultados más específicos o aproximados a las intenciones del diseñador. Una solución novedosa es el uso de "Machine Learning" (ML) [23], el cual es una sub disciplina de la inteligencia artificial (IA) centrada en el desarrollo de algoritmos y técnicas que permiten a la computadora aprender usando datos que se le entregan, para así hacer predicciones o tomar decisiones. El objetivo es integrar la capacidad de identificar patrones en los datos presentados y utilizarlos para realizar nuevas tareas de forma autónoma, presentando una enorme ventaja en términos de eficiencia y flexibilidad en comparación con métodos tradicionales de generación, pero aumentando la complejidad para establecer dichas medidas previas a la generación. Propuestas como los enfoques iterativos de aprendizaje por refuerzo en ML [22], donde se describe el uso de un agente para realizar cambios en los niveles ya completados. Usando un proceso de decisión de Markov, se adapta el nivel actual mediante iteraciones, obteniendo una "recompensa" por medio de la evaluación del nivel, lo cual actúa como indicador sobre si los cambios que se realizan resultan en una mayor eficiencia o si cumple mejor con los criterios esperados por el diseñador con base en una serie de reglas.

3.2. Wave Function Colapse

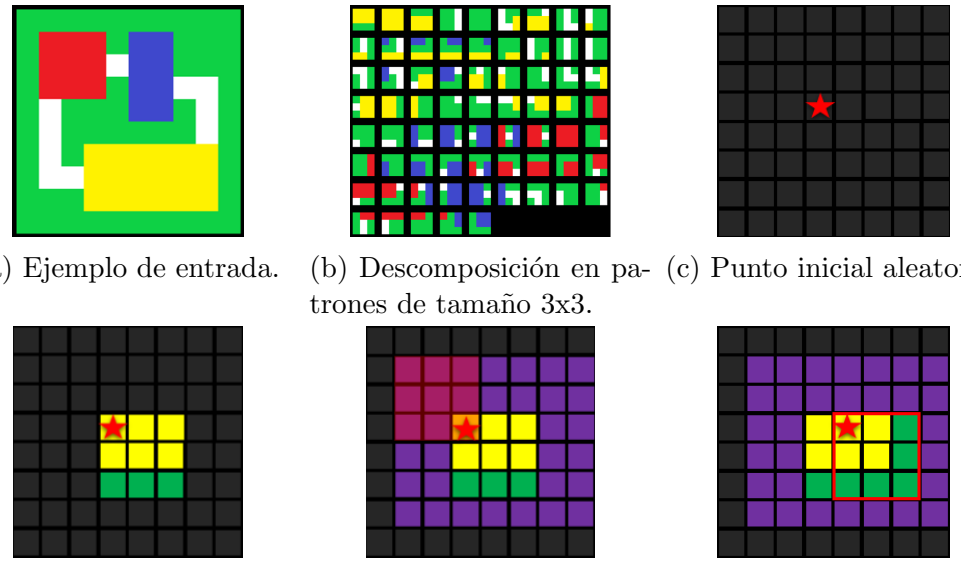
Como ya se mencionó, WFC [11] es un algoritmo que se destaca en la creación de PCG, especialmente en la generación de mapas dentro del ámbito de los videojuegos por múltiples factores y presentando múltiples similitudes con los métodos usados en el ML y los GA. El Algoritmo 3.1 describe el funcionamiento de WFC, mientras que la Figura 3.5 lo muestra gráficamente. Este requiere de un conjunto inicial de información proporcionada por el diseñador que el algoritmo usará a modo de ejemplo del resultado final esperado, permitiendo la creación de contenido que se asemeje con una precisión considerable a lo que espera obtener el diseñador. Una vez el algoritmo tiene los datos de entrada, procede a descomponerlos en patrones de un tamaño N x N de la unidad mínima (o casilla) que componga el ejemplo, guardando todos los posibles estados que puede tener una casilla. Con esta información se pasa a definir un mapa de tamaño S x S en el cual cada casilla almacenará los posibles estados que puede adquirir, pudiendo transformarse en cualquiera de los elementos presentes en los datos de entrada. Con los datos de entrada analizados y un mapa inicializado, se pasa a entrar en un proceso iterativo para ir creando el mapa. Este proceso se divide en 3 partes principales: "selección", "colapso" y "propagación".

Algoritmo 3.1 Wave Function Collapse

```
1: procedure WFC(mapa, ejemplo[x], N)
2:     ObtenerEstados(ejemplo[x])
3:     ObtenerPatrones(ejemplo[x], N)
4:     InicializarMapa(mapa, estados)
5:     while MapaIncompleto do
6:         Selección(mapa, N)
7:         Colapso(mapa, Area_Seleccionada, patrones)
8:         Propagación(mapa, Area_colapsada, patrones)
```

La **selección** consiste en escoger el área con menor entropía, lo que se define como la zona de tamaño N x N que posee la menor cantidad de posibles estados restantes, eligiendo de forma aleatoria si es que hay varias con la misma cantidad de entropía. El **colapso** consiste en seleccionar uno de los patrones que sea compatible con los estados disponibles alrededor del área seleccionada en el paso anterior, si es que hay múltiples patrones compatibles, se puede escoger según criterios como que tantas veces se repitió el mismo patrón en el ejemplo de inicio o de forma aleatoria.

Una vez se tiene una zona con casillas colapsadas, se pasa a la **propagación**, la cual consiste en, usando las casillas colapsadas en el paso anterior, eliminar posibilidades de estados no viables (que no coincidan con ningún patrón) para las casillas adyacentes que no estén totalmente colapsadas, completando la iteración y repitiendo hasta completar el mapa. Como el resultado es hecho a base de un ejemplo, es esperable un resultado aproximado, sin embargo, el algoritmo no posee criterios en su forma base para ajustar el mapa más allá de la compatibilidad entre las fronteras de los patrones, por lo que puede dar lugar a resultados poco coherentes o inviables para la implementación de las mecánicas, por lo que es habitual el uso de modificaciones conocidas como "restricciones".



(a) Ejemplo de entrada. (b) Descomposición en patrones de tamaño 3x3. (c) Punto inicial aleatorio.

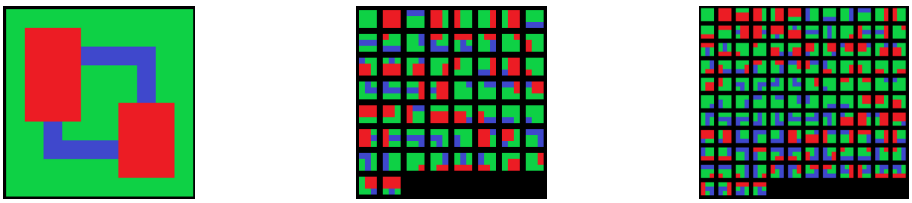
(d) Colapso usando un patrón aleatorio. (e) Propagación. (f) Iteración.

Figura 3.5: Funcionamiento general de WFC.

3.3. Uso de restricciones en WFC

Las restricciones son reglas o criterios que pueden ser integrados en diferentes partes del algoritmo WFC para mejorar el control, mejorar la eficiencia, aumentar la variedad, garantizar un resultado viable de su funcionamiento, etc [24]. Una de las restricciones usadas comúnmente es la rotación e inversión de los patrones obtenidos para aumentar la cantidad disponible durante la generación, como muestra

la Figura 3.6. Otras son la definición de relaciones de conexión entre las casillas, donde ciertos estados pueden o no quedar adyacentes o rodeados a otros estados en específicos, cantidades mínimas o máximas de casillas con un estado determinado, que determinados estados no puedan aparecer en los bordes, etc.



(a) Mapa de entrada. (b) Patrones obtenidos sin restricción. (c) Patrones obtenidos con restricción.

Figura 3.6: Ejemplo simple del uso de restricción de rotación e inversión.

La flexibilidad que ofrecen estas modificaciones lo vuelve altamente versátil en la generación de mapas en videojuegos, tal como muestra una implementación de WFC con restricciones en el videojuego "Super Mario Bros" [25]. Los resultados muestran que los niveles generados daban diseños menos densos de elementos y presentaban una mayor jugabilidad a comparación de los realizados sin restricciones. Entre las restricciones más relevantes usadas en este caso está el posicionamiento forzado de casillas, lo cual implica que una casilla específica sea puesta en una posición específica en la salida por medio de coordenadas que estén dentro del espacio del mapa generado. Las "restricciones no locales" que se aplican a elementos lejanos al espacio generado, requiriendo de información como el nombre identificador, el peso y la frecuencia de generación, indicando que no puede ser colocado más X veces o usando un valor negativo para indicar que no cuenta con un límite. Adicionalmente, se hizo uso de restricción de confinamiento en las casillas, para que determinados estados en las casillas solo puedan aparecer en determinadas áreas del mapa generado, como podrían ser elementos propios del techo o del suelo en los mapas.

Otra propuesta [24] ha añadido una modificación que cambia la forma en que el WFC elige las casillas, usando una combinación de búsqueda binaria con una acumulación de pesos, lo cual permite producir un resultado que refleje mejor la frecuencia de aparición de determinadas casillas en el mapa. La restricción de dependencia define el vínculo entre elementos y cómo pueden ser dispuestos, dando como resultados más complejos y matizados. Por su parte, el recalculado de pesos

altera la probabilidad en el espacio, forzando una saturación en determinadas áreas, reemplazando el peso anterior y afectando también a las casillas adyacentes, lo que permite una mejor localización de objetos según sus interrelaciones, como podría ser la ubicación de un objeto importante, el cual altera los valores a su alrededor para añadir mayor concentración de enemigos. La modificación del área de propagación es similar a un paso de propagación extra, donde un área específica es calculada y luego un evento es forzado. La diferencia es que en vez de elegir una casilla u objeto, se elige un grupo de objetos a ser removidos de un área en específico, como podrían ser los enemigos de un área, dando a los desarrolladores más flexibilidad al crear subáreas en el resultado final [24]. Por último, cabe destacar el uso de la técnica del "Back-tracking" que consiste en llevar una lista de los pasos seguidos hasta el punto actual, para deshacer cambios no favorables e intentar con nuevas combinaciones, esto para rehacer partes del proceso en caso de que se haya llegado a una solución sin salida donde ningún patrón puede completar la generación, evitando descartar todo el trabajo realizado como en el WFC tradicional.

3.4. Multi-Scale Wave Function Colapse

El algoritmo WFC de escala múltiple o MWFC [13] es exactamente igual en funcionamiento al WFC estándar, pero añadiendo una mejora que soluciona uno de los principales problemas del algoritmo, la que consistía en la dificultad para encontrar patrones compatibles en situaciones donde quedara poco espacio o estados disponibles. Dicha mejora consiste en la capacidad de añadir y cambiar entre patrones de múltiples tamaños, como se muestra en el Algoritmo 3.2, de los cuales se puede elegir de forma aleatoria, guiada por pesos, de forma jerárquica o desordenada.

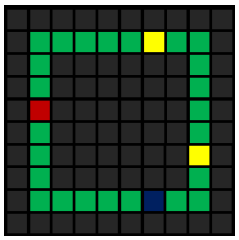
El uso de diferentes tamaños permite combinar la ventaja de los patrones grandes que añadían mayor coherencia y permite obtener un resultado más aproximado a los datos de entrada, dejando unos pocos huecos en los que se colocaban patrones pequeños que encajaban fácilmente, reduciendo en gran medida la tasa de fallos y mejorando los resultados de la generación simultáneamente.

Algoritmo 3.2 Multi-Scale Wave Function Collapse

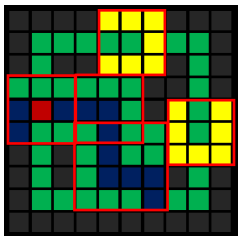
```
1: procedure MWFC(mapa, ejemplos, N[2.. x])
2:     ObtenerEstados(ejemplos)
3:     for i from 0 to N.size() do
4:         ObtenerPatrones(ejemplos, N[i])
5:     InicializarMapa(mapa, estados)
6:     while MapaIncompleto do
7:         selección_tamaño(0, N.size())
8:         Selección(mapa, N[tamaño])
9:         Colapso(mapa, Area_Seleccionada, patrones[tamaño])
10:        Propagación(mapa, Area_colapsada, patrones[tamaño])
```

3.5. Hierarchical Wave Function Colapse

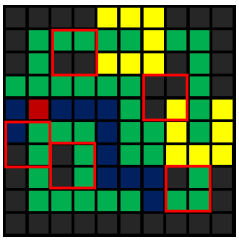
El WFC jerárquico o HWFC [13], es una mejora realizada sobre el MWFC con un enfoque orientado a la generación de niveles en videojuegos, caracterizado por la inclusión de conjuntos de patrones agrupados o separados según su tamaño o "capas", las cuales se disponen en el mapa siguiendo un orden jerárquico del más grande al más pequeño. Las jerarquías principalmente usadas por este algoritmo son las altas, medias y bajas, donde los patrones de alta jerarquía se caracterizan por estar "ahuecados" o con forma de marco, los cuales son puestos en el mapa de forma aleatoria al iniciar la generación, a modo de semillas guía para la siguiente jerarquía. La jerarquía mediana es la encargada de poblar el mapa, siendo de un tamaño intermedio, su objetivo es abarcar tanto como sea posible hasta que solo queden pequeñas áreas de casillas sin colapsar, momento en el cual se pasa a usar la jerarquía baja, constituida por los patrones más pequeños que cubrirán los agujeros y completarán el mapa, como se muestra en la Figura 3.7.



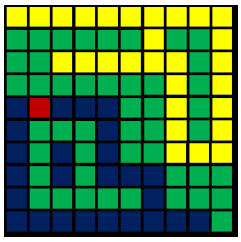
(a) Inicio con patrón de alta jerarquía.



(b) Población del mapa con jerarquía media.



(c) Rellenado con jerarquía baja.



(d) Mapa completo.

Figura 3.7: Funcionamiento general de HWFC.

Si bien esta técnica incrementa mucho las posibilidades de éxito en la generación de un nivel, no está exenta de soluciones sin salida, por lo que, en caso de sufrir de este fallo, recurrirá al back-tracking para regresar a etapas de generación anteriores con el fin de usar una nueva combinación de patrones y lograr completar el mapa exitosamente.

En el Algoritmo 3.3, al igual que con WFC, lo primero es el ingreso de los datos de entrada y los parámetros requeridos (línea 1). Usando esa información, se procede a extraer los patrones de tamaño N, catalogando todos los tipos de elementos que se pueden encontrar como los posibles estados de una casilla (línea 2 - 4) e inicializando el mapa con dicha información (línea 5).

Una vez se tiene esta información, se entrará en un ciclo o función recursiva que irá generando el mapa de forma progresiva (línea 7). En el caso de HWFC, se iniciaría en el nivel de mayor jerarquía, seleccionando un punto en el mapa y definiendo un área alrededor de un tamaño N correspondiente al tamaño de los patrones del nivel jerárquico actual (línea 8). Con esta información, se pasa a revisar qué patrones son compatibles con el área definida, seleccionando uno para colapsar el área y se suma una iteración al nivel actual (línea 9 - 10).

Si se llega al máximo de iteraciones permitidas en el nivel actual (línea 11), se revisa si el nivel actual es el de menor jerarquía (línea 12), si es, se realiza back-

tracking (línea 13), caso contrario, se pasa al siguiente nivel jerárquico (línea 15).

Por último, se realiza una propagación usando los patrones del nivel jerárquico actual, por cada punto en el área que fue previamente colapsada (línea 16).

Algoritmo 3.3 Hierarchical Wave Function Collapse

```
1: procedure HWFC(mapa, ejemplos, N[2.. x], iteración_max[2.. x])
2:     ObtenerEstados(ejemplos)
3:     for i from 0 to N.size() do
4:         ObtenerPatrones(ejemplos, N[i])
5:     InicializarMapa(mapa, estados)
6:     nivel = 0, iteración = 0
7:     while MapaIncompleto do
8:         Selección(mapa, N[nivel])
9:         Colapso(mapa, Area_Seleccionada, patrones[nivel])
10:        iteración++
11:        if iteración_max[nivel] == iteración then
12:            if nivel == N.size() then
13:                Backtracking()
14:            else
15:                nivel++, iteración = 0
16:        Propagación(mapa, Area_colapsada, patrones[nivel])
```

3.6. Nested Wave Function Collapse

El algoritmo WFC anidado o NWFC es una propuesta reciente muy interesante para el desarrollo de mapas a gran escala [26]. Este se caracteriza por realizar ejecuciones de WFC interiores o "IWFC" dentro de una generación de mayor tamaño, presentando también una forma de diagrama Voronoi en su forma de división. Esta organización jerárquica permite reducir de la complejidad temporal exponencial a una polinómica, lo que da como resultado una generación de contenido más eficiente y escalable.

En su ejecución, NWFC divide el mapa en porciones más pequeñas y manejables, manteniendo restricciones entre las divisiones para que las uniones sean congruentes, mientras que en el interior de cada división, se utilizan múltiples ejecuciones de IWFC para generar subredes de un tamaño fijo. IWFC opera de forma muy similar a WFC estándar, pero con la diferencia de que puede tener pre-restricciones propa-

gadas desde otras subredes, lo que significa que puede presentar áreas previamente colapsadas por otros procesos IWFC en las zonas adyacentes.

3.7. Aplicaciones de WFC

Desde la publicación del código abierto de WFC en 2016, múltiples proyectos han hecho uso del algoritmo. Un ejemplo de esto ha sido el videojuego "Townscaper" [27] lanzado el año 2020, el cual es descrito por su creador como un juguete más que un videojuego. Este no presenta objetivos explícitos o una meta que alcanzar, simplemente presenta un pequeño asentamiento y deja libertad al usuario de añadir o eliminar casillas dentro del mapa, dejando el resto de la generación al algoritmo WFC implementado. Esto es un sistema mixto donde el jugador decide qué posición debe ser expandida al seleccionar un color y el algoritmo es el encargado de seleccionar un elemento compatible, modificando a los de su alrededor para hacerlo congruente con el nuevo estado del mapa.

"Cavern Collapse" [28] es otro uso del algoritmo WFC. Este es un videojuego de plataformas estilo "rogue-like", el cual permite la colaboración entre el usuario y el algoritmo de generación por medio de un editor de niveles. Este videojuego muestra el uso de características jerárquicas, pues el nivel es generado en 4 etapas diferentes, iniciando por las paredes o bordes rígidos del mapa, la adición de plataformas y escaleras, la colocación de recompensas (monedas) y peligros (pinchos), y, por último, las decoraciones como plantas o señales, realizando un pulido durante cada iteración para mejorar el resultado final y verificando que los elementos estén en posiciones válidas para el flujo del nivel.

"Caves of Qud" [29] es otra propuesta interesante, pues también es un videojuego "rogue-like" lanzado en el 2023. Está ambientado en un mundo postapocalíptico que hace uso de WFC, el cual nos presenta un mundo abierto que es en parte prefabricado y parte PCG, que también hace uso de un sistema de misiones que también combina contenido prefabricado y PCG. El mundo creado presenta una gran diversidad, dado que implementa un sistema de biomas y regiones separadas, cada una con sus propias posibilidades de terreno y reglas de generación, de una forma similar al del diagrama Voronoi y al NWFC, que le dan como resultado un mundo congruente entre todos los elementos que lo componen [30].

WFC también puede ayudar en la creación de contenido educativo, como nos

presenta la propuesta en desarrollo de "MECH.AI" [31], un videojuego de tácticas por turno que busca presentar los fundamentos de la programación a estudiantes de nivel secundario de una manera más atractiva y lúdica. En este, los jugadores podrán tomar el control de los objetos no solo mediante la interface, sino que también podrán hacerlo mediante líneas de comando que buscan simular la programación. Bajo este contexto, se usó WFC con el objetivo de crear mapas congruentes que presentaran una buena diversidad, evitando la monotonía y requiriendo que el usuario piense constante en una solución nueva para el problema presentado. Con esto, el diseñador podrá ayudarse del PCG para facilitar la creación de múltiples niveles válidos a un mayor ritmo que con una planificación y diseño tradicional realizada únicamente por el diseñador, reduciendo así su carga de trabajo y acelerando el proceso de creación del videojuego.

Nuevas aproximaciones a HWFC también buscan mejorar la capacidad de generación del algoritmo, tal como se detalla en "Semantic HWFC" [32], donde se nos presenta el concepto de las "Meta-tile", el cual expande el concepto de las casillas de una entidad que simplemente contiene estados a una estructura similar a un grafo. Estas nuevas casillas contienen simultáneamente la información de a qué jerarquía pertenece y las restricciones que tiene con otros tipos de casillas entre las diferentes jerarquías, esencialmente integrando restricciones dentro de cada casilla, lo cual permite tener un mayor control y capacidad de interacción con el proceso de diseño de niveles.

Como último, es importante recordar que WFC y sus derivados están enfocados al diseño de niveles/mapas en videojuegos, pero no se encuentran limitados a esta tarea. Una muestra de esto se puede ver en una implementación de HWFC orientada a la composición procedural de música [33], donde se muestra la capacidad y flexibilidad que ofrece el algoritmo para abarcar desde la generación de elementos que no cambian a lo largo de la composición, hasta segmentos detallados y de gran complejidad, permitiendo la generación de música a diferentes niveles. Esta aproximación del algoritmo HWFC proporciona un marco flexible en el cual el compositor/diseñador pueda trabajar, destacando la importancia de las restricciones y prototipos, permitiendo integrar notas o acordes directamente en la composición, mientras que el algoritmo "rellena vacíos" para crear una nueva interpretación de los parámetros de entrada.

Sin duda, la técnica de hacer uso de patrones obtenidos a partir de ejemplos

de entrada creados por un diseñador, junto con el uso de restricciones que afectan a la generación, sumado a un enfoque iterativo, representa una herramienta extremadamente útil y con un potencial enorme para ser investigada en el ámbito del PCG.

4. Metodología de desarrollo

A continuación, se pasará a explicar la planificación y metodología de desarrollo para el proyecto.

4.1. Metodología FDD

La metodología de desarrollo ágil Feature-Driven Development (FDD) se enfoca en la entrega de características concretas y significativas en el desarrollo de software [34]. Se divide a grandes rasgos en una serie de etapas: Creación de un modelo de dominio, útil para evaluar el alcance/dominio de las características, desarrollar una lista maestra de las características que compondrán el proyecto y posteriormente planificar, diseñar y construir por cada uno de los elementos en la lista.

4.2. Código del proyecto

La implementación original de HWFC [14] fue realizada en Python, un lenguaje de programación dinámico e interpretado caracterizado por su enfoque general que enfatiza la legibilidad del código y la simplicidad sintáctica fácil de aprender. El código se compila mediante un código fuente ".py" el cual es pasado a una forma intermedia conocida como "Bytecode", una representación de bajo nivel para ser usada por la máquina virtual de Python o "PVM", la cual ejecuta las instrucciones una por una. Cada instrucción es una operación simple como cargar un valor, realizar una operación aritmética o llamar a una función. Esto lo hace independiente a la plataforma usada, facilitando la depuración y desarrollo rápido, donde no es necesario declarar el tipo de una variable, ya que Python lo determina durante el tiempo

de ejecución del código. Lamentablemente, esto también supone una sobrecarga adicional en el tiempo de ejecución, como en el uso de más memoria en comparación con un lenguaje estático.

Un lenguaje de programación estático se caracteriza por realizar la verificación de los tipos de variables durante el tiempo de compilación, lo que significa que la detección de errores relacionados con los tipos de datos usados se realiza antes de la ejecución del programa, lo cual puede ayudar a evitar errores y mejorar la eficiencia del código[15]. Dentro de este tipo de lenguajes se encuentra el altamente conocido lenguaje C++ [35], destacado por su eficiencia en tiempo de ejecución, por su capacidad de administración de memoria y de una optimización avanzada, resultando en un rendimiento superior en comparación a otros lenguajes de alto nivel. Adicionalmente, el lenguaje presenta múltiples ventajas de uso general como la programación orientada a objetos, modularidad que ayuda a la organización del código, una extensa cantidad de bibliotecas para ayudar en el desarrollo de aplicaciones o funciones complejas, una alta compatibilidad, junto con una buena portabilidad multiplataforma.

Como repositorio se usó "GitHub" [1], una plataforma gratuita para que los desarrolladores puedan administrar sus proyectos, ordenando y permitiendo recuperar, en caso de ser necesario, cada nueva versión de un archivo que sea subida.

Para cumplir con las características requeridas en el funcionamiento del algoritmo HWFC y su posterior evaluación de rendimiento, se llevará una lista de todos los elementos que se deben completar, los cuales se detallan en la Tabla 4.1.

[1] Link al repositorio del proyecto: https://github.com/Kaylians/Implementacion_HWFC_C_PlusPlus.

Cuadro 4.1: Listado de los elementos principales que componen el código del proyecto y las características a desarrollar en cada uno de estos.

Elementos del código	**Características a desarrollar**
Lógica de funcionamiento	Lectura e ingreso de valores. Lectura de patrones predefinidos. Lectura, descomposición y creación de patrones. Recomposición y escritura de imagen de salida. Sistemas de impresión por consola. Sistema de impresión de los patrones para información. Sistema de métricas.
WFC	Selección. Colapso. Propagación.
MWFC	Procesamiento de patrones de tamaño N variable. Capacidad para cambiar entre diferentes tamaños durante la ejecución.
HWFC	Implementación de valores jerárquicos en los patrones. Procesamiento de patrones con forma de anillo (solo bordes).
Restricciones	Selección guiada por entropía. Peso basado en la repetición de patrones. Límite de uso jerárquico. Restricción de rotación y simetría.
Backtracking	Guardado del estado de la generación por cada iteración. Peso basado en la repetición de patrones. Retroceso de los valores a un estado anterior.
Métricas	Guardado del tiempo de cada ejecución realizada. Lectura de mapas para el cálculo de la métrica “Hamming”. Lectura de patrones usados para el cálculo de la métrica “Divergencia KL”.

5. Experimentos y resultados

A continuación, se pasará a detallar las condiciones bajo las cuales se desarrollarán los experimentos y cuáles serán las métricas usadas para medir los resultados obtenidos.

Se hará usando el subsistema Linux de Windows (WSL) con el sistema operativo de código abierto "Ubuntu", específicamente en la versión "22.04.3 LTS" disponible de forma gratuita en la tienda de aplicaciones de Microsoft, o también disponible en su página oficial [36]. Para compilar el código de C++ dentro de Ubuntu se usó la versión "11.4.0", mientras que en Python se usó la versión "3.10.12".

El computador en el que se están realizando estas pruebas cuenta con un procesador AMD Ryzen 9 5900HX, 3,3 Ghz, 8 procesadores principales, 16 procesadores lógicos y 16 GB de memoria RAM.

5.1. Métricas

El principal parámetro para evaluar y comparar será el tiempo promedio que tardan en realizarse las ejecuciones por cada uno de los modos (WFC, MWFC y HWFC) en ambos algoritmos. Se realizarán un total de 30 ejecuciones en cada modo de la versión creada en C++ y de la versión de Python, usando un mapa de tamaño fijo de 40 x 40.

Adicionalmente, se realizará una "t-test" pareada [37], la cual es una herramienta para evaluar cuán significativa es la diferencia observada entre las dos muestras. Cuanto más cercano sea a 0 el valor, mayor es la diferencia de las muestras, mientras que la cercanía a 1 indica mayor similitud. Si el valor resultante es menor a 0.05, se considerará como una diferencia estadísticamente significativa.

Todos los mapas generados en C++ serán comparados entre sí usando las métricas de "Hamming" y "KLD", la evaluación se hará respecto a 100 ejecuciones por cada una de las variantes del algoritmo, las cuales serán calculadas de forma automática por el código del proyecto y guardadas dentro de hojas de cálculo en la carpeta "generatedLevels", que a su vez estará separada según el modo y el tamaño del mapa solicitado, junto con los mapas generados en formato "Map_X.ppm" y con una hoja de cálculo "Map_X.csv" que guarda la información de los patrones usados.

5.1.1. Hamming

La distancia de Hamming [38] es una métrica utilizada para comparar dos cadenas de igual longitud, midiendo cuantas diferencias presentan. En el contexto de este proyecto, representa cuantas casillas tienen el mismo estado entre 2 mapas, por lo que, si un mapa es exactamente igual a otro, tendrá un valor equivalente 0, mientras que, si todas las casillas son diferentes, tendrá un valor igual a 1.

5.1.2. Kullback-Leibler Divergence

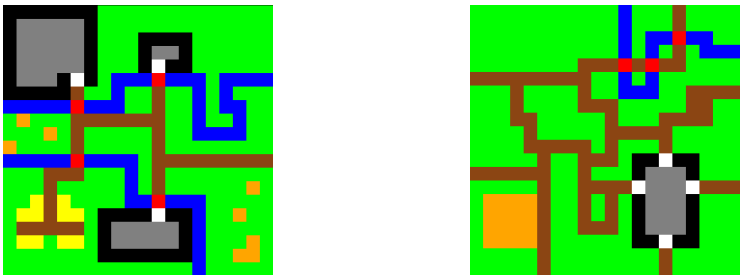
La divergencia de Kullback-Leibler o "KLD" [39] es una medida de la diferencia entre dos distribuciones de probabilidad. En el contexto de este proyecto, representa cuanto fue la diferencia entre los patrones (de un mismo tamaño N x N) usados para generar un mapa, comparando los patrones de baja jerarquía usados para generar los mapas de un mismo algoritmo.

5.2. Resultados

5.2.1. Tiempo

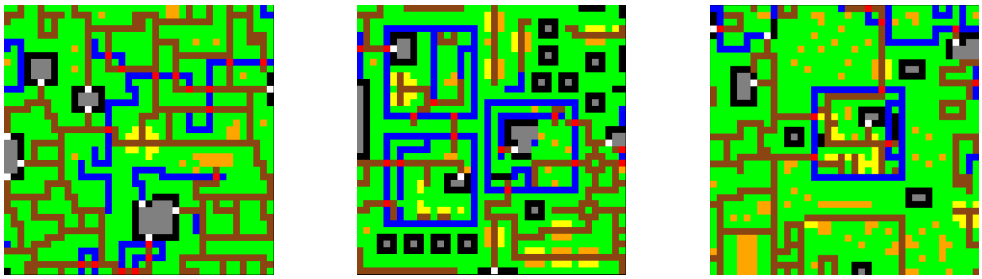
La información de entrada para la generación de los mapas fue exactamente la misma en ambos algoritmos, usando los dos ejemplos originales de Python que se muestran en la Figura 5.1, así como sus rotaciones y los patrones de tamaño alto y medio predefinidos en formato ".txt".

Los tiempos obtenidos de la generación muestran un margen muy amplio entre cada uno de los algoritmos en C++ con respecto a su implementación original, como se muestra en el gráfico comparativo Figura 5.3. Ejemplos de mapas creados durante la prueba en la Figura 5.2.



(a) “example_1/Rot1”. (b) “example_2/Rot1”.

Figura 5.1: Mapas usados como ejemplos de entrada en la implementación original en Python. Convertidos a formato “.ppm” de su forma base de mapa de coordenadas en “.txt”.



(a) WFC, T: 85.953 seg (b) MWFC, T: 4.215 seg (c) HWFC, T: 7.043 seg

Figura 5.2: Mapas creados durante los experimentos de medición de tiempo.

WFC en C++ mostró que su tiempo de ejecución promedio para un patrón de tamaño 3 es de 139 segundos, mientras que en Python es de 563 segundos en promedio, siendo más de 4 veces más rápido, presentando una desviación estándar de 103.3 segundos en C++ y de 30.2 segundos en Python.

La diferencia de tiempo en WFC con respecto a los otros algoritmos es significativa (tanto en C++, como en Python). Esto muestra la limitación que supone el uso de un único tamaño de patrón, pues retrasa en gran medida la creación de un mapa por la dificultad que añade el encontrar soluciones viables. El valor de la “t-test” para WFC fue de: $4.40558e{-}20$.

MWFC resultó unas 21.9 veces más rápido, mostrando un promedio de 5.6 segundos, mientras que en Python el promedio de ejecución fue de 122.8 segundos. Presentando una desviación estándar de 2.3 segundos en C++ y de 80 segundos en Python. El valor de la “t-test” para MWFC fue de: $8.08376e{-}9$

HWFC mostró el incremento más drástico de todos, siendo 35.1 veces más rápido

en C++, dando un promedio de 4.4 segundos para cada generación, mientras que en Python dio un promedio de 157.4 segundos. Presentando una desviación estándar de 2.9 segundos en C++ y de 31.3 segundos en Python. El valor de la "t-test" para HWFC fue de: 3.05991e−22.

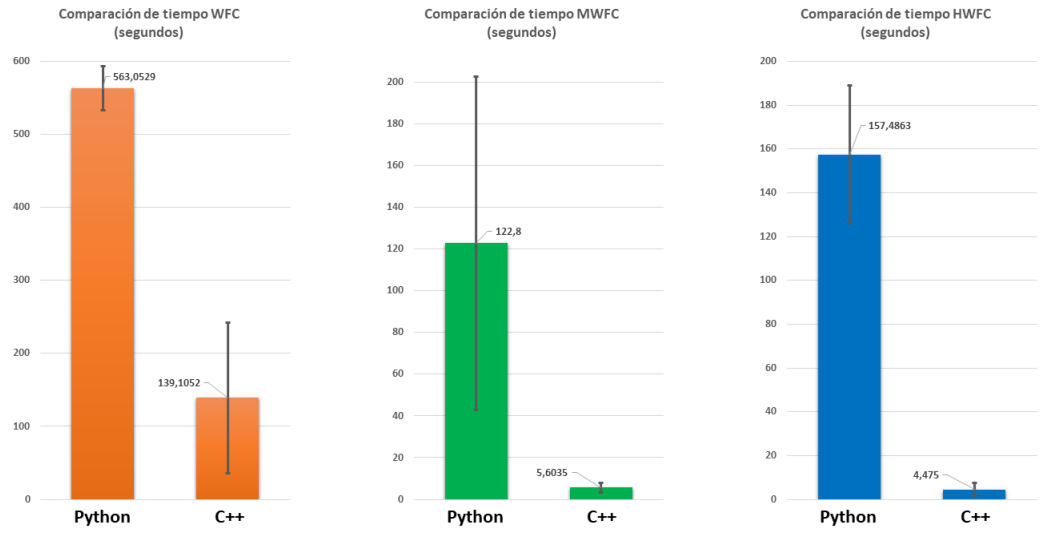


Figura 5.3: Gráfico comparativo entre los tiempos de los 3 algoritmos en C++ y Python. Las barras de error muestran la desviación estándar.

5.2.2. Hamming

Los resultados de la distancia Hamming dentro de una margen de 0 a 1, muestran que los promedios obtenidos por los algoritmos en C++ están en un margen de entre 0.7 y 0.5, mientras que Python tuvo un margen de entre 0.8 y 0.4, como se muestra en el gráfico comparativo Figura 5.4.

Esto supone la suficiente variedad entre las ejecuciones como para no presentar resultados idénticos, pero sin ser disparatadamente desiguales, lo que supondría poca coherencia en los mapas creados.

El valor de la "t-test" para WFC fue de: 0.207737, para MWFC fue de: 0.187771 y para HWFC fue de: 0.12615, lo cual comprueba que los resultados no presentan una diferencia estadísticamente significativa.

Tanto en Python, como en C++, el algoritmo que mostró una mayor diferencia fue

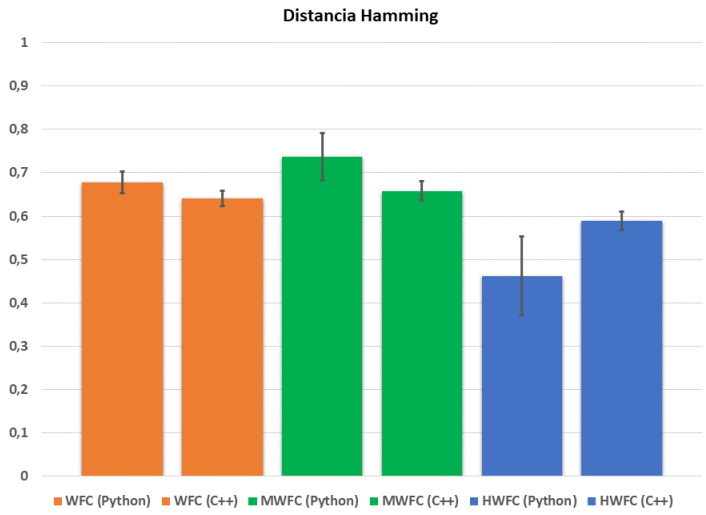


Figura 5.4: Comparación de las distancias de Hamming entre los 3 algoritmos realizados en C++ y Python.

MWFC, dado que, al usar patrones de todos los tamaños disponibles sin restricciones, podía ofrecer una mayor variedad en los resultados. WFC resultó en un punto medio, pues al usar únicamente patrones de tamaño 3, se encuentra con una mayor libertad de dar resultados diferentes, pero limitado a las mismas combinaciones de patrones. HWFC es el que presentó la menor distancia, pues, al usar un único patrón grande a modo de semilla, fue el que mayor probabilidad tenía de repetir áreas con los mismos estados para las casillas del mapa.

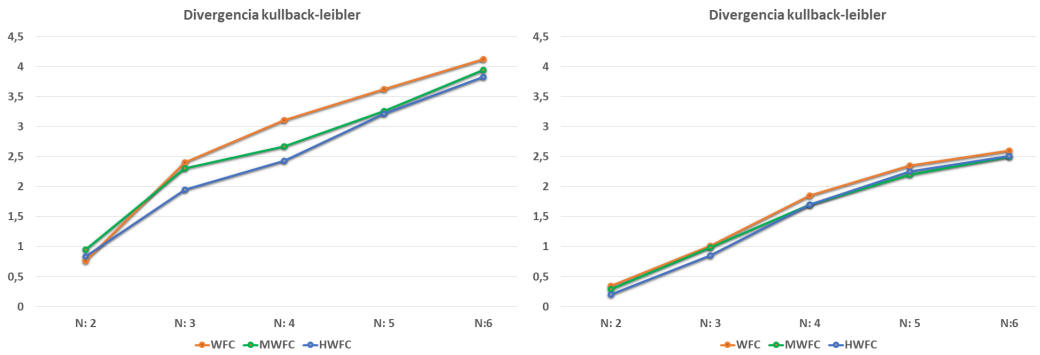
5.2.3. Kullback-Leibler Divergence

Dada la baja tasa de éxito que presenta WFC a medida que aumenta su tamaño, para los valores de N:4, N:5 y N:6 en WFC, se usó una variante de MWFC específicamente modificada para la obtención de la métrica. Funciona exactamente igual al WFC estándar, completando el mapa con el tamaño N requerido, pero permitiendo que, si el nivel llega a fallar por falta de patrones compatibles (extremadamente común en las etapas finales para este tamaño de patrón), en lugar de recurrir inmediatamente a back-tracking o a reiniciar la generación descartando el trabajo realizado, intente usar de forma limitada el tamaño mínimo posible (N:2)

para completar la ejecución.

Como la fórmula de KLD requiere de la misma cantidad de información en ambos grupos de datos, los cuales deben ser diferentes de 0; Se usó la misma constante de 0.001 usada en Python para representar que un mapa no está presente.

Los valores de KLD obtenidos muestran un resultado esperado, pues a medida que incrementa el tamaño de los patrones para el algoritmo, también aumenta la cantidad de patrones disponibles, lo que a su vez lleva a una mayor cantidad de posibles elecciones, como se muestra en el gráfico comparativo Figura 5.5. También es importante recordar que, aunque el valor de KLD aumente, siempre se tendrán una pequeña cantidad de patrones predominantes con una gran cantidad de usos, dadas las restricciones basadas en pesos, según que tan frecuente fue el patrón en la imagen de entrada.



(a) C++. (b) Python.

Figura 5.5: Gráficos de los resultados de KLD entre los diferentes algoritmos. La información usada en el gráfico correspondiente a Python fue obtenida directamente del documento original de HWFC [13].

6. Conclusiones y trabajo futuro

Dados los tiempos obtenidos y los valores de la ejecución de la "t-test" en los algoritmos de C++ a comparación de sus ejecuciones originales en Python, usando un mismo entorno, con una lógica de código similar y usando un mismo equipo para realizar las ejecuciones, muestran de forma incontrovertida que un lenguaje de tipo estático resulta en una mejor opción en términos de eficiencia con respecto a un lenguaje de tipo dinámico como lo es Python en el tiempo de generación de PCG, por lo que se ha verificado la hipótesis.

Esta ventaja en términos de tiempo representa también una oportunidad de expandir las posibilidades de la generación a mayor escala y sumando elementos de complejidad adicionales, sin comprometer los tiempos de generación de cada una de las ejecuciones.

6.1. Trabajo futuro y mejoras

Dentro de los planes de mejora para el algoritmo se encuentra la adición de nuevas restricciones para la mejora de los resultados, principalmente para el perfeccionamiento de los niveles una vez ya creados en busca de una mayor congruencia entre los elementos presentados en el resultado final.

Una de las posibilidades contempladas para esto es la implementación de agentes autónomos que cuenten con información de patrones específicos, los cuales exploraran el mapa moviéndose únicamente por estados definidos en sus reglas, permitiendo que modifiquen el mapa para habilitar caminos. Permitiendo mejorar las uniones en zonas que no conectaran bien o añadir una cantidad de estados específicos en algunos puntos compatibles.

Otra mejora de mucho interés sería la integración de los principios del NWFC/diagrama Voronoi para una creación rápida en mapas de tamaño varias veces más grande al de los usados en los experimentos, pues los algoritmos de WFC, MWFC y HWFC irán perdiendo eficiencia de forma exponencial a medida que se aumenta el tamaño del mapa.

Mientras que con un tamaño de 40 x 40 el algoritmo debe colapsar un total de 1.600 casillas, un mapa de tamaño 400 x 400 tendría un total de 160.000 casillas por colapsar, lo cual representa 100 veces más carga de trabajo, además de que, si se llegara a un mapa inviable, el algoritmo debería realizar back-tracking, lo que no haría más que aumentar el tiempo de generación.

Para esto, la combinación de los elementos mencionados, NWFC para manejar diferentes áreas de generación simultáneas, el uso de reglas adicionales para facilitar la unión de dichas zonas y el uso de agentes para asegurar la coherencia entre las diferentes zonas permitiría crear mapas que incluso podrían llegar a catalogarse dentro del apartado de "mundo abierto", permitiendo generarlos en un tiempo relativamente corto.

6.2. Conversión a Plugin

Dado el potencial que presenta el algoritmo dentro del ámbito del PCG, se buscará convertirlo en un "Plugin", es decir, un complemento que añade funciones extra a otros programas, en este caso, al motor gráfico "Unreal Engine 5" [40]. Este presenta ventajas muy significativas como plataforma a la que integrar el algoritmo desarrollado, ya que, se trata de un motor gráfico altamente popular en el desarrollo de videojuegos, tanto de forma independiente como a nivel empresarial [41].

Por último, y como fundamento principal para su elección, el motor gráfico funciona en el mismo lenguaje que el código actual, es decir, hace uso de C++ para el funcionamiento, lo que lo convierte en un ambiente ideal para implementar la generación de mapas, teniendo que realizar modificaciones como añadir la funcionalidad 3D al algoritmo para aumentar su utilidad potencial como herramienta en la creación de PCG dentro del ámbito de los videojuegos.

Bibliografía

[1] Future Market Analytics. Video games market by segments, by region and companies - market analysis, trends, revenue opportunity, competitive analysis, and forecast 2023-2032. https://futuremarketanalytics.com/report/video-games-market/, 2023.

[2] Ubisoft. Just dance+ official web page. https://www.ubisoft.com/es-mx/game/just-dance/plus.

[3] Bethesda. Fallout official web page. https://fallout.bethesda.net/es/guide.

[4] Rockstar Games. Red dead redemption 2 official web page. https://www.rockstargames.com/es/reddeadredemption2.

[5] PwC. Top 5 developments driving growth for video games. https://www.pwc.com/us/en/tech-effect/emerging-tech/emerging-technology-trends-in-the-gaming-industry.html, 2024.

[6] Alena Porokh. How long does it take to make a game? https://kevurugames.com/blog/how-long-does-it-take-to-make-a-game/#:~:text=AAA%20games%20typically%20take%20two,players%20and%20require%20extensive%20modifications., 2024.

[7] John Gibbs, Stefan Seidel, and Nicholas Berente. Video games, procedural designing with autonomous tools: Video games, procedural generation, and creativity. *International Conference on Information Systems (ICIS) 2019 Proceedings*, 2019. https://aisel.aisnet.org/icis2019/future_of_work/future_work/14/.

[8] Boyan Bontchev. *MODERN TRENDS IN THE AUTOMATIC GENERATION OF CONTENT FOR VIDEO GAMES*. Serdica, Journal of Computing, 2016.

[9] Gearbox Software. Borderlands official web page. `https://www.gearboxsoftware.com/game/borderlands/`.

[10] Mojang Studios. Minecraft official web page. `https://www.minecraft.net/en-us`.

[11] Maxim Gumin. Wave function collapse algorithm, 2016. `https://github.com/mxgmn/WaveFunctionCollapse`.

[12] Isaac Karth and Adam Smith. Wavefunctioncollapse is constraint solving in the wild. *association for computing machinery*, 2017. `https://doi.org/10.1145/3102071.3110566`.

[13] Michael Beukman, Branden Ingram, Ireton Liu, and Benjamin Rosman. Hierarchical wavefunction collapse. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 19(1):23–33, Oct. 2023. `https://ojs.aaai.org/index.php/AIIDE/article/view/27498`.

[14] Michael Beukman, Branden Ingram, Ireton Liu, and Benjamin Rosman. Hierarchical wavefunction collapse, 2022. `https://github.com/Michael-Beukman/HWFC`.

[15] Fronty. The difference between dynamic and static code. `https://fronty.com/the-difference-between-dynamic-and-static-code/`, 2024.

[16] Richard Moss. ASCII art + permadeath: The history of roguelike games. `https://arstechnica.com/gaming/2020/03/ascii-art-permadeath-the-history-of-roguelike-games/`, 2020.

[17] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016. `https://www.pcgbook.com/chapter03.pdf`.

[18] Félix Santos and Eulogio Santos. Algoritmo de prim para la implementación de laberintos aleatorios en videojuegos. *Industrial Data*, 15:80–89, 2012. `https://www.redalyc.org/articulo.oa?id=81629470011`.

[19] Ricardo Lopes, Tim Tutenel, Ruben M Smelik, Klaas Jan De Kraker, and Rafael Bidarra. A constrained growth method for procedural floor plan generation. In *Proc. 11th Int. Conf. Intell. Games Simul*, pages 13–20, 2010. https://www.academia.edu/13992421/A_constrained_growth_method_for_procedural_floor_plan_generation?sm=b.

[20] Kati Steven Emmanuel, Christian Mathuram, Akshay Rai Priyadarshi, Rishu Abraham George, and J. Anitha. A beginners guide to procedural terrain modelling techniques. In *2019 2nd International Conference on Signal Processing and Communication ICSPC*, 2019. https://ieeexplore.ieee.org/document/8976682.

[21] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. https://ieeexplore.ieee.org/document/4082128.

[22] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. PCGRL: Procedural content generation via reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 16, 2020. https://ojs.aaai.org/index.php/AIIDE/article/view/7416.

[23] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K. Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural content generation via machine learning PCGML. *IEEE Transactions on Games*, 10(3):257–270, 2018. https://ieeexplore.ieee.org/document/8382283.

[24] Arunpreet Sandhu, Zeyuan Chen, and Joshua McCoy. Enhancing wave function collapse with design-level constraints. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3337722.3337752.

[25] Quentin Edward Morris. Modifying wave function collapse for more complex use in game generation and design. *Trinity University. Computer Science Honors Theses. 58*, 2021. https://digitalcommons.trinity.edu/compsci_honors/58/.

[26] Yuhe Nie, Shaoming Zheng, Zhan Zhuang, and Julian Togelius. Nested wave function collapse enables large-scale content generation. *IEEE Transactions on Games*, 2024. https://doi.org/10.48550/arXiv.2308.07307.

[27] Tommy Thompson. How townscaper works: A story four games in the making. https://www.gamedeveloper.com/game-platforms/how-townscaper-works-a-story-four-games-in-the-making#close-modal, 2022.

[28] Darcy Matheson. Cavern collapse. https://prometheusgamedev.itch.io/cavern-collapse.

[29] Freehold Games. Caves of qud. https://freeholdgames.itch.io/cavesofqud.

[30] Roguelike Celebration and Brian Bucklew. Brian bucklew - dungeon generation via wave function collapse. https://www.youtube.com/watch?v=fnFj3dOKcIQ&ab_channel=RoguelikeCelebration.

[31] Muhammad Alifian Aqshol, Ardiawan Bagus Harisa, Pulung Nurtantianto Andono, and Wen-Kai Tai. Pacing control in strategy game map generation using wave function collapse. https://journals.uob.edu.bh/handle/123456789/5707, 2024.

[32] Shaad Alaka and Rafael Bidarra. Hierarchical semantic wave function collapse. In *Proceedings of the 18th International Conference on the Foundations of Digital Games*, New York, NY, USA, 2023. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/3582437.3587209.

[33] Pal Varga and Rafael Bidarra. Procedural mixed-initiative music composition with hierarchical wave function collapse. In *Proceedings of PCG 2023 - Workshop on Procedural Content Generation for Games, co-located with the 18th International Conference on the Foundations of Digital Games*, 03 2023. https://doi.org/10.1145/3582437.3587209.

[34] Virender Singh. Feature driven development FDD : An agile methodology. https://www.toolsqa.com/agile/feature-driven-development/.

[35] Kholmatov Abrorjon. C and C++ programming languages capabilities and difference. *Galaxy International Interdisciplinary Research Journal GIIRJ*, 2023. https://internationaljournals.co.in/index.php/giirj/article/view/4533.

[36] Canonical ubuntu. https://ubuntu.com/download.

[37] Statistical Discovery JMP. T-test. https://www.jmp.com/es_cl/statistics-knowledge-portal/t-test.html, 2024.

[38] Xin-She Yang. Chapter 6 - Genetic Algorithms. In *Nature-Inspired Optimization Algorithms (Second Edition)*, pages 91–100. Academic Press, second edition edition, 2021. https://www.sciencedirect.com/science/article/pii/B9780128219867000135.

[39] Nikolaj Buhl. Kl divergence in machine learning. https://encord.com/blog/kl-divergence-in-machine-learning/#:~:text=KL%20divergence%20is%20defined%20as,distributions%20under%20observation%20are%20identical.

[40] Epic Games. Unreal engine 5. https://www.unrealengine.com/en-US/unreal-engine-5.

[41] Abelardo González. Unreal engine 5, ¿por qué es la gran revolución de los videojuegos? https://www.pccomponentes.com/unreal-engine-5-que-es.