UNIVERSIDAD CATÓLICA DE CUENCA

UNIVERSIDAD CATÓLICA DE CUENCA

Comunidad Educativa al Servicio del Pueblo

FACULTAD DE INFORMÁTICA, CIENCIAS DE LA COMPUTACIÓN E INNOVACIÓN TECNOLÓGICA

CARRERA DE REALIDAD VIRTUAL Y VIDEOJUEGOS

Generación de mapas con diseño modular para juegos de estrategia 3D en Unity

PROYECTO DE INTEGRACIÓN CURRICULAR PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN REALIDAD VIRTUAL Y VIDEOJUEGOS

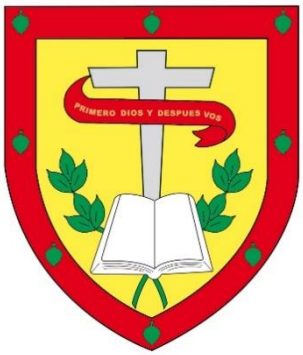
AUTOR: ADRIAN NESTORIO ABAD TENORIO

DIRECTORA: DIS. TATIANA ALEXANDRA CABRERA SILVA, MSC.

CUENCA - ECUADOR

2026

DIOS, PATRIA, CULTURA Y DESARROLLO



UNIVERSIDAD CATÓLICA DE CUENCA

Comunidad Educativa al Servicio del Pueblo

FACULTAD DE INFORMÁTICA, CIENCIAS DE LA COMPUTACIÓN E INNOVACIÓN TECNOLÓGICA

CARRERA DE REALIDAD VIRTUAL Y VIDEOJUEGOS

Generación de mapas con diseño modular para juegos de estrategia 3D en Unity

PROYECTO DE INTEGRACIÓN CURRICULAR PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN REALIDAD VIRTUAL Y VIDEOJUEGOS

AUTOR: ADRIAN NESTORIO ABAD TENORIO

DIRECTORA: DIS. TATIANA ALEXANDRA CABRERA SILVA, MSC.

CUENCA - ECUADOR

2026

DIOS, PATRIA, CULTURA Y DESARROLLO

Declaratoria de Autoría y Responsabilidad

Adrian Nestorio Abad Tenorio portador de la cédula de ciudadanía N.º **0150834216**. Declaro ser el autor de la obra: "Generación de mapas con diseño modular para juegos de estrategia 3D en Unity", sobre la cual me hago responsable sobre las opiniones, versiones e ideas expresadas. Declaro que la misma ha sido elaborada respetando los derechos de propiedad intelectual de terceros y eximo a la Universidad Católica de Cuenca sobre cualquier reclamación que pudiera existir al respecto. Declaro finalmente que mi obra ha sido realizada cumpliendo con todos los requisitos legales, éticos y bioéticos de investigación, que la misma no incumple con la normativa nacional e internacional en el área específica de investigación, sobre la que también me responsabilizo y eximo a la Universidad Católica de Cuenca de toda reclamación al respecto.

Cuenca, **31 de agosto del 2026**

F: .....

Adrian Nestorio Abad Tenorio

C.I. 0150834216

Universidad Católica de Cuenca

CERTIFICADO

Certificado

Certifico que el presente proyecto de titulación fue desarrollado por **Adrian Nestorio Abad Tenorio,** portador de la cedula de ciudadanía **N.º 0150834216** con el tema "Generación de mapas con diseño modular para juegos de estrategia 3D en Unity", bajo mi supervisión.

MSc. Tatiana Alexandra Cabrera Silva

DIRECTORA DEL PROYECTO DE TITULACIÓN

UNIVERSIDAD CATÓLICA DE CUENCA

Dedicatoria

Con profundo cariño y gratitud, dedico este trabajo a las personas que han sido fundamentales en mi vida y en el proceso que me ha llevado hasta este momento.

A toda mi familia, desde abuelas a sobrina, pero especialmente a mi madre, quien nunca dudó de mi carrera; a mi padre, quien me apoyó a pesar de sus dudas iniciales; y a mi hermano, quien me enseñó a buscar lo que yo quería de mi vida.

A mis compañeros de tesis, que, sin ellos, este proyecto nunca hubiera tenido la posibilidad de existir.

A mi tutora de tesis, y fundadora de la carrera, Tatiana Cabrera, quien logró convencer a mi padre a permitirme entrar a esta carrera y perseguir mis sueños, siempre apoyándome a través de toda esta etapa académica.

A mis hermanos de sangre, Andres y Galo, quienes nunca me dejaron abandonar el sueño de hacer de esta carrera mi vida profesional.

A Christine, la luz de mis ojos. Una persona quien, sin importar la distancia, siempre me da todo su corazón, y me ha dado la energía, acompañándome durante las largas noches de realizar código y escribir este texto.

Finalmente, a aquellos quienes programaron juegos antes de mí, y a quienes aparecerán a programar juegos después de mí. Espero que este texto les sirva de alguna manera en su ámbito profesional o educativo.

Agradecimiento

Al haber terminado esta etapa significativa de mi vida, quiero agradecer a mis seres queridos por ser un apoyo incondicional en mi vida y ayudarme a alcanzar este logro importante. Agradezco también a mi tutora de Tesis, cuyo apoyo y enseñanza me han guiado durante este trabajo significativo. Agradezco también a los docentes universitarios quien con su sabiduría me han enriquecido de conocimientos, inspirándome a seguir avanzando día a día.

Resumen

En el desarrollo de videojuegos, la generación de mapas sufre una falta de herramientas para realizarlo de manera ágil y simple, aumentando los costos en la producción de un videojuego. El objetivo del proyecto es desarrollar un método de generación procedural de mapas para el motor de videojuegos de Unity en proyectos 3D, utilizando la geometría modular para su creación con polígonos para la creación del mapa en 3 categorías: triangulares, cuadrangulares, y polígonos de 5 lados y más. Con cada categoría se utilizaron diferentes fórmulas para generar correctamente mapas mediante geometría modular, previniendo un procesamiento excesivo con una fórmula general para todas las figuras. La propuesta fue insertada dentro de un juego prueba, con el fin de crear niveles de manera aleatoria a partir de casillas que determinan la ruta del jugador en el mapa, para obtener mejoras o dificultades, permitiendo una experiencia única a cada usuario. Los resultados fueron mapas con la forma del mismo polígono donde están hechas las baldosas que los conforman, creando un nuevo estilo de generación de mapas, fomentando las competencias tácticas del jugador, con una experiencia diferente para cada usuario.

Palabras Clave: *Unity, geometría modular, mapas, videojuegos, juegos de estrategia.*

Abstract

In video game development, map generation lacks tools that it to be implemented efficiently and easily, thereby increasing video game production costs. This project aimed to develop a procedural map generation method for the Unity game engine in 3D projects by employing modular geometry based on polygons to generate maps in three categories: triangular, quadrilateral, and five-sider or higher-order polygons. Different formulas were applied to each category to ensure the correct generation of maps using modular geometry while preventing excessive computational processing that would result from employing a single general formula for all polygonal shapes. The proposed method was integrated into a prototype game to procedurally generate levels from tiles that determine the player's path across the map, enabling the acquisition of upgrades or the introduction of additional challenges, thus providing a unique experience for each user. The results consisted of maps whose overall shape corresponded to the polygonal tiles from which they were constructed, introducing a novel map generation approach that promotes players to develop tactical skills while delivering a distinct experience for every user.

Keywords: *Unity, modular geometry, maps, video games, strategy games.*

Índice De Contenido

Índice De Figuras

Índice De Tablas

Introducción

En la actualidad, los videojuegos son un elemento de sumo impacto en la sociedad, y con la democratización de la producción de aquellos, cualquier persona puede hacer su propio juego. Sin embargo, existe la escasez de producción de géneros específicos, debido a la falta de interés de estudios grandes, o por la carencia de recursos por parte de desarrolladores independientes. Frente a esta realidad, surge la necesidad de diseñar un programa que permite facilitar la creación de estos tipos de juegos para fomentar la expansión del mercado, y que apoye a desarrolladores independientes para que puedan continuar con sus emprendimientos en la industria.

A continuación, se presenta una breve descripción de los capítulos que conforman este trabajo de titulación:

Capítulo I: En este capítulo se desarrolla el marco referencial de la investigación, en el cual se plantea y formula el problema. Se incluyen los antecedentes, la justificación del estudio, los objetivos generales y específicos, y las limitaciones y delimitaciones de la investigación.

Capítulo II: Se desarrolla el marco teórico en donde se abordan los principales conceptos que fundamentan el estudio, tales como la teselación, coordenadas polares, RAM, la generación procedural de contendido, la optimización de rendimiento del CPU, el diseño modular y motores de videojuegos.

Capítulo III: Se diseña el manual de usuario del producto que sale en base a la investigación, incluyendo los requisitos para poder correr el juego prueba, el público objetivo a la cual se dirige, las instrucciones de uso, y evidencias de uso con un segmento del público objetivo.

Capítulo IV: Se detalla el manual de programación, en donde se explican los *scripts*, líneas de código, patrones de diseño, y demás elementos que fueron utilizados en la creación del programa que sirve de solución para la problemática investigada. Además, se defiende el uso de aplicaciones y componentes necesarias para su elaboración, y se presentan capturas de pantalla del código. Después, se explica el mantenimiento del programa, y su escalabilidad al futuro. Finalmente, se describe como se validó la eficiencia del programa mediante pruebas del juego con usuarios que forman parte del público objetivo. Se presentan los resultados, y se da un análisis a base de lo encontrado.

Capítulo I

1. Marco Referencial

1.1. Contextualización del problema

Shubin & Kugurakova[1] explican que el desarrollo de videojuegos es considerado un proceso computacional y creativo muy complejo por el requerimiento de una interacción sinérgica entre especialistas de varios campos tecnológicos y artísticos. Dentro de aquel, la arquitectura del código base es especialmente importante. Esto es debido a que, en su simplicidad, tiene que lograr captar la atención de los jugadores de hoy en día. Mecánicas como las de movimiento, los sistemas de interfaz de usuario (*User Interface*, o UI por sus siglas en inglés) y la progresión conforman lo que se denomina el núcleo del videojuego, tal como indica Zubek[2]. En este mismo contexto, Fabricatore[3] sostiene que la viabilidad y el éxito de un producto interactivo como los videojuegos dependen de su capacidad para satisfacer las necesidades de sus consumidores. Por lo tanto, el diseño de estas mecánicas base resulta ser un asunto crítico, ya que dicta la fluidez con la que el usuario percibe y asimila la experiencia virtual.

Entre estos componentes fundamentales para la experiencia del jugador (*User Experience*, o UX), el diseño de mapas se destaca como un elemento esencial. El objetivo del mapa dentro de cualquier juego es más que solo dar una posición al usuario dentro del mundo; debe guiar al jugador a través de él, y empujarlo a explorar de su propia voluntad, aumentando así de manera significativa la sensación de inmersión dentro del juego. Un mapa hecho de manera correcta permite al usuario visualizar al mundo dentro de su mente, lo cual es clave para que el jugador pueda tomar decisiones durante su progreso. Dependiendo del género específico, el mapa tiene la ventaja adicional de poder dictar cómo el jugador

interactúa con el mundo, y qué tan rápido lo hace. Con esto en mente, se puede decir que el mapa determina el comportamiento del jugador y las estrategias que debe emplear mientras avanzaba por los niveles.

Esta influencia de los mapas en la jugabilidad es más evidente en los videojuegos de estrategia por turnos (*Turn-Based Strategy Games*, o TBS, por sus siglas en inglés). En este tipo de juego los mapas no suelen ser continuos, sino que están segmentados mediante polígonos regulares o irregulares, lo que permite al jugador fácilmente establecer los caminos por donde sus unidades pueden andar. Además, permite al programa establecer zonas de obstáculos y posiciones iniciales de unidades enemigas de manera precisa, lo que aumenta la dificultad del juego. La industria recurre al diseño modular para crear estos mapas de la manera más eficiente; esta técnica utiliza unidades geométricas prefabricadas y pequeñas, normalmente en la forma de polígonos regulares, para que unidas formen mapas más complejos y extensos. Kim y Sung[4] demuestran en sus investigaciones que el uso de polígonos regulares no únicamente optimiza el rendimiento de los PCs que corren el programa, sino que ayudan a dar una claridad visual al terreno. Esto permite al jugador y a la Inteligencia Artificial (IA) del juego calcular de manera exacta el alcance de sus acciones e inferir las consecuencias de dichas acciones en siguientes turnos, y logra que el jugador desarrolle la habilidad de formar estrategias con varios turnos en mente.

1.2. Planteamiento del problema

Las técnicas existentes para el diseño de mapas no han sido utilizadas para el género de juegos TBS (juegos por turnos), lo que ha provocado la caída de su producción. Esta falta de desarrollo también se debe a la pérdida de interés para producir juegos de este estilo por parte de los grandes estudios (AAA). Aquellas compañías evitan el riesgo financiero asociado con crear nuevos títulos en este campo; más bien prefieren innovar dentro de los géneros de

acción rápida, como juegos de acción de rol (*Action Roleplay Game* o ARPG en sus siglas en inglés) o los que tienen modelos de servicio en vivo. Cuando se busca crear un juego TBS, suele ser un título con menos riesgo, que proviene de franquicias fuertemente consolidadas como Sid Meier's Civilization o Crusader Kings[5].

Por esta razón los desarrolladores independientes, o *indies*, desean cubrir esta demanda. Sin embargo, Borreli et. al.[6] explican que es difícil adquirir las herramientas para el desarrollo de algunos juegos, especialmente del género TBS en 3D, las cuales son severamente limitadas; además, ningún motor podría abarcar todos los instrumentos necesarios para todos los géneros[7]. Por esta razón, la mayoría de los juegos TBS que fueron desarrollados en Unity por *indies* eran diseñados en proyectos 2D. Esto se debe a la elaboración de sus mapas, los cuales pueden implementarse de manera fácil mediante sistemas de cuadrículas, conocidos en la industria como *tilemaps*. Por lo contrario, como nos indican Sharme et. al.[8] la generación procedural dentro de proyectos en 3D contiene complejidades matemáticas, físicas y más que nada, de optimización de rendimiento. Estas dificultades elevan de manera significativa la barrera técnica para que equipos pequeños puedan crear estos tipos de juegos, y es por esta razón que suelen abandonar proyectos 3D de este género[9].

1.3. Formulación del problema

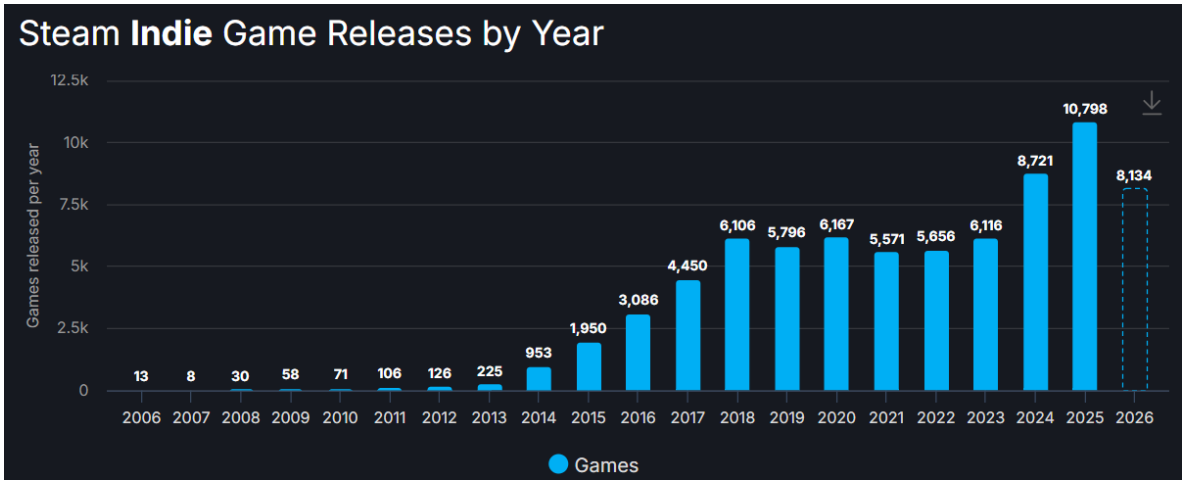
Con base al problema, se formula la siguiente pregunta de investigación:

¿Es posible desarrollar un programa para la generación de mapas con distintos polígonos en 3D para juegos de estrategia dentro de Unity, editable desde el Inspector de aquel motor?

1.4. Justificación de la investigación

Este proyecto surge como respuesta al incremento exponencial de títulos *indie* en años recientes, además de la evidente carencia de herramientas accesibles para la creación de videojuegos TBS en 3D. En el año 2025, el mercado tuvo un fuerte dominio de estudios *indie*, con 10,680 títulos lanzados bajo esta categoría en Steam, la plataforma más extensa de distribución de juegos para PC. Sin embargo, de estos, se publicaron solo unos cientos bajo el género TBS[10].

Figura 1
Ventas de juegos indie en Steam por año.



Nota. Se cuantifican la cantidad de juegos vendidos en Steam bajo la categoría de Indie por año. Obtenido de [10].

Para agilizar la producción de estos juegos, y aumentar la capacidad de disfrutarlos nuevamente (conocido en la industria como *replayability)*, desarrolladoras hoy en día suelen usar Generación de Contenido Procedural (*Procedural Content Generation*, también conocido como PCG). Sin embargo, la implementación de esta técnica es muy difícil para muchos diseñadores. En el estado actual del campo, no existen métodos estandarizados o interfaces simples que permitan crear mapas mediante esta técnica. Esto les obliga a recurrir a la manera tradicional de un diseño manual. Mientras tanto, las empresas de categoría AAA

que sí emplean sistemas automatizados lo hacen a través de sus propios motores de juegos, cuyo código fuente se mantiene cerrado para el público general.

Esta brecha entre los recursos de estudios más grandes y desarrolladores *indie* es la causante de la falta considerable de juegos TBS en 3D, en donde todos operan bajo severas restricciones de presupuesto, tiempo, o ambas cosas. Para democratizar el diseño de este género, las herramientas deben ser integradas de forma nativa o mediante complementos en motores de juegos de uso público, como Unity. Proveer un sistema accesible permitiría que desarrolladoras *indie* enfoquen sus esfuerzos en el diseño de mecánicas únicas para su juego, y su narrativa.

1.5. Objetivo general

Diseñar un sistema de generación modular de mapas en forma de polígonos regulares para la simplificación de diseño de videojuegos de estrategia por turnos para desarrolladores de videojuegos independientes.

1.6. Objetivos específicos

- Definir las reglas de segmentación y teselación necesarias en la generación de mapas modulares.
- Desarrollar el algoritmo de generación procedural en Unity, estructurando el código por categorías poligonales e implementando un sistema que facilite la importación e instanciación de modelos tridimensionales (*prefabs*) personalizados.
- Evaluar la usabilidad y el rendimiento computacional de la herramienta mediante pruebas de generación de mapas en un juego creado en Unity.

1.7. Alcance del proyecto

Este proyecto tiene como objetivo diseñar un sistema de generación modular de mapas en 3D basados en polígonos regulares (triángulos, cuadrados y figuras de cinco o más

lados) plasmados en los ejes X y Z del entorno 3D, y busca satisfacer las necesidades de desarrolladoras *indie*, por lo cual se creará el programa dentro de Unity 6. Además, tiene la meta adicional de simplificar el diseño de niveles para videojuegos TBS. El alcance de este proyecto es únicamente el desarrollo lógico del algoritmo de generación y su posible implementación en un juego del género para validar su funcionamiento.

Tabla 1
Alcance y límites del proyecto

Incluido Dentro del Proyecto	No Incluido en el Proyecto
Generación modular de mapas con figuras de 3 lados en adelante	Generación de mapas con figuras circulares
Muestra de la funcionalidad del programa en Unity	Inclusión del algoritmo en otros motores de videojuegos
Desarrollo y prueba de la generación	Comparación del método creado con métodos similares ya existentes

Fuente: *Elaboración propia*

***Nota*.** Comparación entre los límites y alcances del proyecto.

1.8. Delimitaciones

En este proyecto no se realizará una comparación con herramientas existentes que cumplen funciones parecidas, ya que la mayoría son desarrolladas por estudios grandes, y, por lo tanto, no accesibles para el público para comparar tiempos. Además, el programa no será implementado en cualquier otro motor de juegos además de Unity, por los tiempos de desarrollo permitidos, y la gran diferencia entre lenguajes de programación de cada motor. Finalmente, círculos no serán incluidos entre las figuras de las cuales se incluirán en los algoritmos.

1.9. Beneficiarios

Este proyecto dará un aporte de gran valor práctico al remover la complejidad matemática de diseñar estos tipos de algoritmos desde cero, y condensarlo en una herramienta visual e intuitiva dentro de Unity. Mediante este programa, lo que deberá hacer el diseñador es cambiar los variables para generar el mapa con sus especificaciones en mente, quitando la necesidad de escribir líneas de código.

Las desarrolladoras independientes serán las beneficiarias directas del proyecto, debido a su impacto en la reducción de tiempos de programación y costos de producción con un programa ya implementado en sus proyectos de Unity. Tendrán la libertad de enfocar sus esfuerzos en la innovación de mecánicas, IA y narrativa, y esto a su vez podría revitalizar el género de juegos TBS y fomentar la competencia en desarrollo de software independiente en la industria global de tecnología.

Los beneficiarios indirectos de este proyecto serán los aficionados de los juegos TBS, quienes tendrán una mayor cantidad de juegos para consumir, debido a una producción mayor de aquellos por parte de las desarrolladoras independientes.

1.10. Metodología general

Este proyecto tendrá un enfoque mixto, centrado en el desarrollo del algoritmo de generación de contenido procedural (PCG) dentro de Unity. Para resolver las complicaciones relacionados al posicionamiento, la metodología usará diseño modular para generar el mapa a partir de figuras geométricas más pequeñas en patrones según el cálculo de la posición de cada casilla. La estructura del proyecto será dividida en tres fases fundamentales:

1.10.1. Análisis y definición matemática

En esta etapa, se escribirá el código que realizará la generación del mapa en sí. El algoritmo determinará la posición vectorial exacta de cada casilla en el espacio 3D y recibirá los parámetros para crear el mapa, como el número de lados del polígono, el radio de cada casilla y el espacio entre ellas. Dado que todas las figuras son distintas, no se podrá usar un solo método para generar todos los mapas. Esto causaría posicionamientos incorrectos, en base a la diferencia en superficie y perímetro de cada polígono. Por lo tanto, se usará un método principal que tomará la variable de la cantidad de lados para elegir una de varias técnicas que harán la generación de la manera más optima de acuerdo con la figura: coordenadas polares y cálculos de apotema para una generación anillar mediante polígonos de 5 o más lados, cálculos basados en una disposición en figura de diamante para rejillas cuadradas y lógica condicional que invierte la figura para encajar de manera exacta para triángulos.

1.10.2. Implementación tecnológica

Una vez que el algoritmo haya sido calibrado, se implementaran las variables editables al código. Se usará Unity para implementar este *script*, por su alta prevalencia en la industria y su capacidad para producir juegos con la menor cantidad de recursos gastados en los ordenadores que los corre, para el mayor alcance posible. Además, se configurará el algoritmo para que se puedan implementar elementos prefabricados, conocidos como *prefabs* en Unity. Con esto, el estilo de arte del juego no será afectado por el programa de generación, y el artista simplemente creará casillas con la figura que desea usar para crear los mapas, con el estilo de arte de su preferencia.

1.10.3. Validación y evaluación

Finalmente, se validará el algoritmo mediante la creación de un juego prueba dentro del motor. Con aquel, se podrá verificar que los mapas anillares se instancien sin errores de posicionamiento, por ejemplo, si las casillas se instanciaran una encima de otra. Además, se podrá comprobar que el mapa resultante brinde un camino claro para la IA de navegación de enemigos en el juego, y más importante, para los jugadores, cumpliendo con los requisitos de un videojuego TBS. Esto se realiza mediante una metodología mixta, a través de la cual se aplicará una encuesta y preguntas directas a los participantes al utilizar el juego.

1.11. Resultados esperados

Al final del tiempo de elaboración del proyecto, se entregará lo siguiente:

- Un algoritmo definido por partes que calcule el posicionamiento vectorial exacto para la construcción casillas geométricas en forma de polígonos regulares, que unidos forman un mapa completo para un juego TBS.
- Un componente del *script* del algoritmo en Unity 6, que tiene las variables para la generación visibles, de manera que el diseñador pueda editarlas para cambiar el mapa a su visión.
- Un juego que utilice la generación del mapa para crear sus niveles.

1.12. Relación con el protocolo de investigación

Este proyecto se relaciona con el protocolo de investigación por el hecho de que los resultados esperan brindar una solución al problema planteado. El algoritmo que se busca generar sería disponible para que desarrolladores *indie* lo use para sus propios mapas, y con esto fomentar el desarrollo de juegos TBS en 3D, poniéndolo a la par con géneros actualmente en alta demanda en el mercado de los videojuegos.

Además, insertar este algoritmo dentro de un videojuego que entra en el género establecido por la investigación permite validar si es que el programa cumple con el objetivo deseado de simplificar el desarrollo de aquel, y por lo tanto, podría fomentar el diseño de estos títulos a futuro.

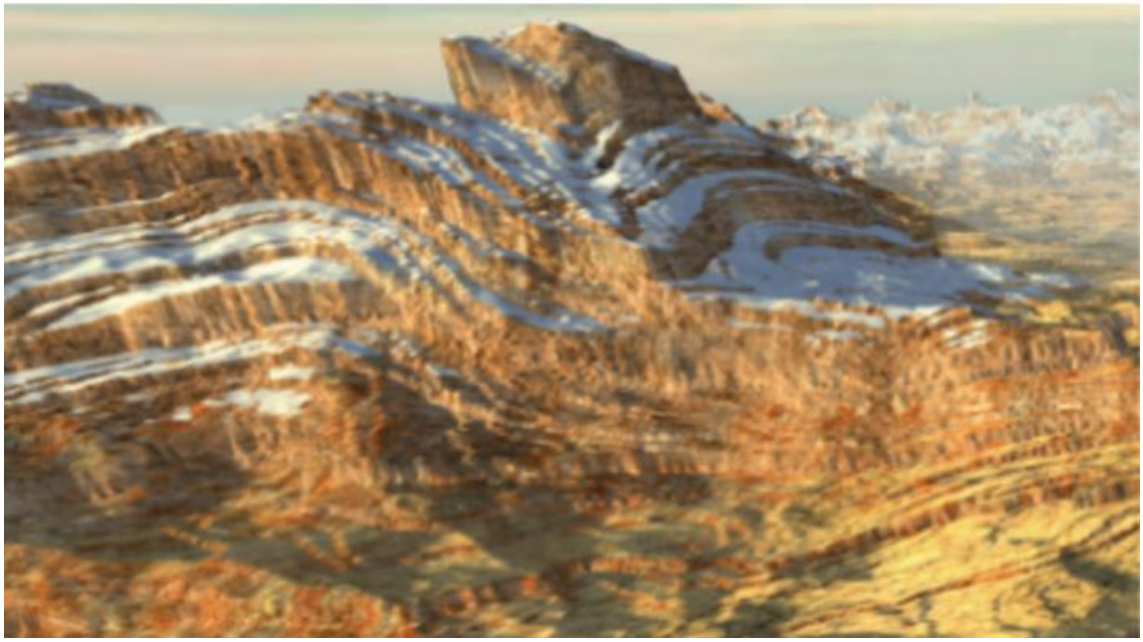
Capítulo II

2. Marco Teórico

2.1. Antecedentes de investigación

En el año 2009, Smelik et al.[11] presentaron una investigación en donde se analizaron diversos métodos de generación procedural, enfocados en el modelado de terrenos en espacios virtuales en 3D. Indicaron que la creación de pinceles procedurales permitió no únicamente generar terrenos con una sola herramienta, si no utilizar la misma para definir un patrón para que el pincel genere de manera propia el resto del espacio.

Figura 2
Generación de terreno mediante pincel procedural.

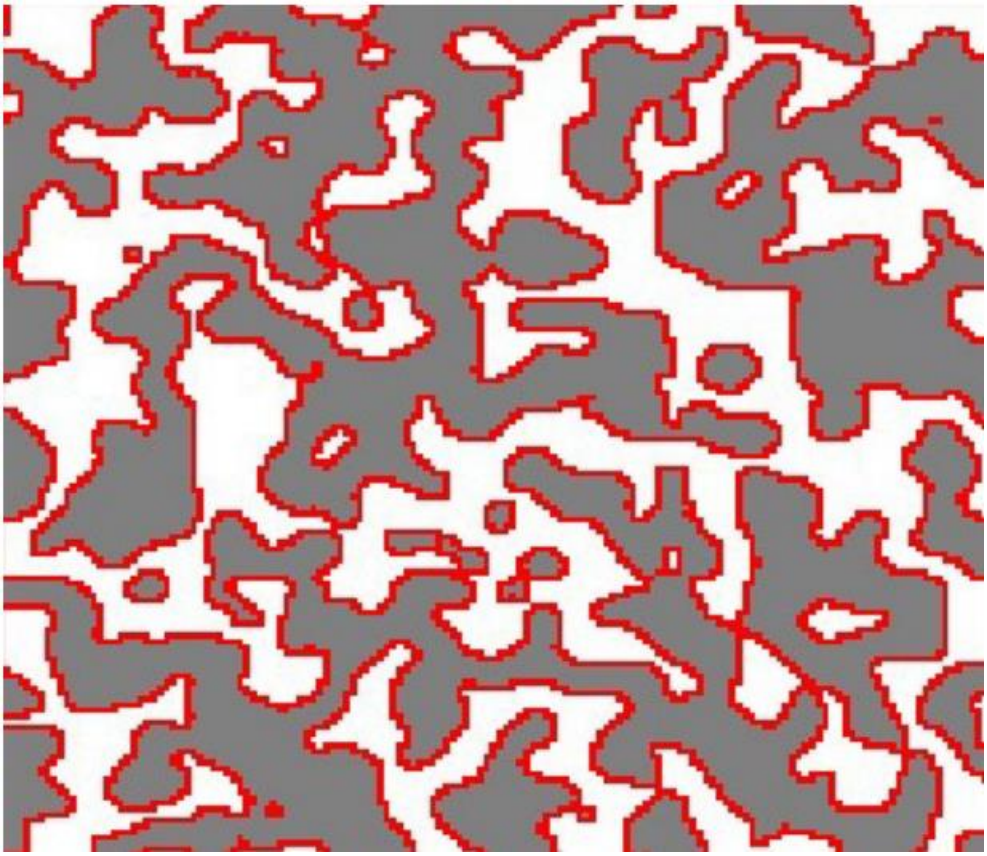


***Nota*.** Resultado de usar un pincel procedural para diseñar una topología en 3D. Obtenido de [11]

Con este trabajo, lograron identificar técnicas fundamentales para crear mundos virtuales mediante algoritmos, y se destacó la necesidad de equilibrar el control del diseñador con la naturaleza aleatoria de aquel.

Posteriormente, en el 2014, Van Der Linden, Lopes y Bidarra[12] investigaron la generación procedural de niveles en un juego enfocado en mazmorras, con la finalidad de determinar cómo las reglas del algoritmo podían integrarse con el diseño del juego para ofrecer mecánicas de exploración más variadas y detalladas. Realizaron un análisis del método de un autómata celular (*cellular automata*), realizado por Johnson et. al. [13]. Esta técnica consistía en establecer leyes que podrían afectar los diferentes estados de un mapa dividido en celdas. De ahí, el algoritmo eligió de manera aleatoria cuales leyes aplican a cada celda, causando una generación única a cada mapa. Mediante el juego Cave Crawler, Johnson pudo establecer la eficiencia de esta técnica, y esto formaría una base importante para la generación de entornos mediante código.

Figura 3
Generación de mapa Cave Crawler.



Nota. Uso de autómata celular para generar un nivel en el juego Cave Crawler. Obtenido de [13].

En un estudio del año 2024, Kim & Sung[4] realizaron una investigación para validar el uso de hexágonos en mapas con el fin de mejorar la navegación de los agentes de IA en Unity. Sus pruebas se basaron en la hipótesis de que los hexágonos serían más útiles para la dirección de agentes de IA debido a una mayor cantidad de lados hacia donde se puede dirigir el agente.

Figura 4
Boceto de posibles caminos de acción de agentes de IA.

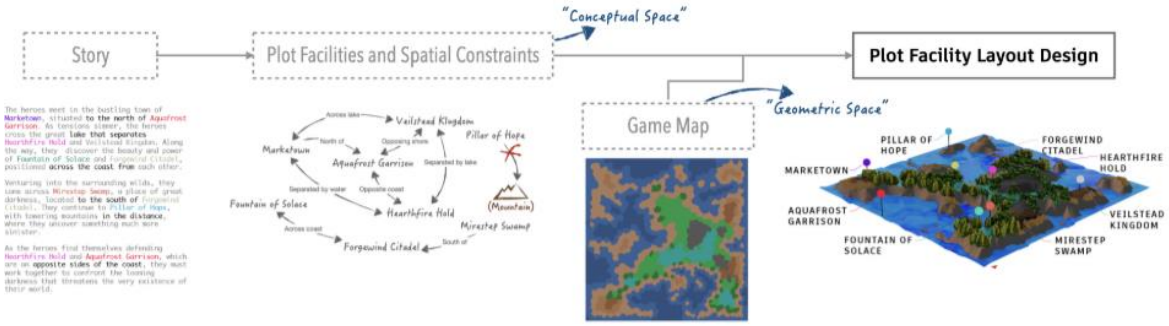
***Nota*.** Descripción visual de las distintas acciones que puede tomar un agente de IA mediante un mapa de hexágonos. Obtenido de [4]

En sus resultados, lograron determinar que la estructura de un mapa hexagonal permitía una mayor libertad de movimiento al agente, lo que dio un nivel adicional de complejidad táctica a su funcionalidad, especialmente en juegos segmentados por turnos. Esto apoyó las conclusiones de la investigación de Sahr[14] en el año 2011, cuando él determinó que patrones en figuras en sistemas virtuales servían para un mejor almacenamiento de datos, y, por lo tanto, mayor optimización en el programa.

Ese mismo año, Wang et al.[15] propusieron "PlotMap", una herramienta de diseño automatizado para la creación de mundos de videojuegos. Este programa tenía como objetivo diseñar un mapa basado en la historia que debería contar el mundo, usando la trama para determinar ciertos parámetros que se pueden ingresar al programa para crear un mapa en base a la narrativa. Con el mundo creado a base de la historia, se definieron los biomas para el mapa, y se ingresaron como argumentos al algoritmo. A partir de aquí, el sistema creaba dos

tipos de mapas: uno en 2D mediante pixeles, y uno utilizando polígonos 2D para crear una topología para un mapa en 3D.

Figura 5
Generación de un mapa con PlotMap.



***Nota*.** Proceso de generación de un mapa con pixeles y polígonos 2D mediante el programa PlotMap. Obtenido de [15]

Los autores concluyeron que automatizar la disposición espacial, o *layout*, de un mapa mediante algoritmos redujo drásticamente los tiempos de desarrollo y permitió iteraciones de diseño más ágiles para los diseñadores.

Finalmente, a principios de 2025, Yallico at al.[16] publicaron un artículo sobre el desarrollo de un videojuego de estrategia en tiempo real (*Real-Time Strategy Game*, o RTS por sus siglas en inglés) para PC utilizando el motor de videojuegos (*game engine*) Unity. Al terminar de diseñar el juego prueba, “Cosmic Crusade”, fue presentado ante 31 personas entre las edades de 19 y 40 años para determinar si el juego generó interés para los aficionados de este género de juegos. Al terminar el período de prueba, se realizó una encuesta a los jugadores para cuantificar su satisfacción.

Tabla 2
Resultados de encuesta de satisfacción del juego Cosmic Crusade.

N°	Test de usabilidad basado en los principios heurísticos	Puntaje – Escala Likert				
		1	2	3	4	5
1	¿Qué tan divertido le resultó jugar Cosmic Crusade?	0%	0%	3%	23%	74%
2	¿Qué tan difícil le resultó jugar Cosmic Crusade?	3%	6%	36%	23%	32%
3	¿Qué tan emocionado se sintió al jugar Cosmic Crusade?	0%	3%	3%	13%	81%
4	¿Volverías a jugar Cosmic Crusade?	0%	0%	0%	3%	97%

***Nota*.** Respuestas recolectadas por los desarrolladores de Cosmic Crusade para validar la satisfacción del juego con los usuarios que lo probaron. Obtenido de [16].

Las respuestas de la encuesta fueron positivas, y esto demostró la viabilidad y las ventajas de utilizar dichos motores para la gestión de mecánicas complejas de estrategia. Además, esto comprobó que el motor optimizó rendimiento mediante patrones de diseño de programación implementados en sus componentes, removiendo la necesidad del desarrollador de crear líneas nuevas de código, y diseño modular.

Analizando estos trabajos se pudo ver que, aunque existen investigaciones profundas sobre la generación procedural y los beneficios de la geometría poligonal en mapas para la IA de un juego, la mayoría de las soluciones seguían requiriendo conocimientos avanzados de programación, dado que hubo una falta de componentes incluidas en los motores de videojuegos para optimizar y simplificar dichas generaciones.

2.2. Bases teóricas

2.2.1. Teselación

El fundamento de esta generación de mapa se basa en la teselación, un aspecto único de diferentes figuras geométricas que permite ahorrar espacio, además de dar una estética agradable, lo cual es esencial para mapas de videojuegos. Si es que el juego consiste en un mapa como son los juegos TBS, no debe producir una sobrecarga visual al jugador, y con tantos polígonos conformándolo, la teselación es una herramienta útil en formar un patrón visualmente ideal.

Pickover[17] define a la teselación como la habilidad de usar las mismas figuras geométricas para llenar un espacio. La idea con esto es organizarlos en un patrón que permitía distinguir cada figura pequeña, pero unidos crear algo más grande. Un ejemplo claro es una colmena de abejas, en dónde cada celda de la colmena es un hexágono. Unidos, forman una estructura más compleja, pero por dentro, cada celda tiene su propósito.

2.2.2. Plano cartesiano en 3D (ejes x, y, z)

Dunn & Parberry[18] nos explican que un plano cartesiano es un espacio cuadrangular en donde se pueden colocar puntos mediante unidades equidistantes en todos los ejes de dicho plano, siendo denominados de la siguiente manera:

$$(x, y)$$

En esta nomenclatura, los términos *x* y *y* corresponden al valor de la distancia entre el origen del plano, y la ubicación del punto en estos ejes.

2.2.3. Coordenadas polares

Coordenadas dentro del plano cartesiano sirven para formar figuras con trazos simples, tales como triángulos y cuadrados, a base de puntos. Sin embargo, cuando se trata

de figuras más complejas con una mayor cantidad de lados, plasmarlos de manera correcta en el plano requiere el cálculo de la distancia desde su centro. Para esto sirven las coordenadas polares.

Dunn & Parberry[18] definen a las coordenadas polares como un punto y un ángulo que surgen desde un origen. Con estos tipos de coordenadas, el plano donde los puntos son ubicados solo tienen un eje, y el valor de las coordenadas se representan de la siguiente manera:

$$(r, \theta)$$

En esta nomenclatura, *r* es el valor numérico de la magnitud del vector de distancia entre el origen y el punto, y el símbolo *theta* es el ángulo que se forma entre el eje de origen y el punto.

2.3. Bases metodológicas

El enfoque de la investigación de este proyecto fue cuantitativo, a través de encuestas, debido a los objetivos de la investigación, que establecieron la cantidad de juegos *indie* desarrollados, pero la falta de juegos TBS entre aquellas. La investigación además tuvo una naturaleza aplicada por su contexto en la industria de los videojuegos, en donde existía una brecha tecnológica entre desarrolladores *indies* y estudios AAA.

Para mejor obtener los resultados para esta investigación, se ha determinado que se debe usar el desarrollo de *software* ágil. Basado en los estudios de Meckenstock et. al.[19], se pudo ver que este estilo de desarrollo era ideal para un tiempo de desarrollo relativamente corto, dada la capacidad de poder hacer pruebas con cada avance del proyecto.

2.4. Bases tecnológicas

2.4.1. Motores de videojuegos

Para la elaboración de videojuegos, se utilizan motores de videojuegos, o generalmente conocidos en la industria mediante su nombre en inglés, *game engines*. Politowski et. al.[20] nos explican que estas aplicaciones son esenciales para la creación de juegos digitales por la facilidad de implementar mecánicas sencillas con ellos, permitiendo a desarrolladoras enfocarse en las mecánicas únicas a su producto. Aunque en la antigüedad se creaban motores para un solo juego o serie, la tendencia de crear motores que sean de uso general para cualquier juego fue incrementando por su accesibilidad y reducción de costos al no tener que crear un *engine* nuevo para cada nuevo título que se desea lanzar.

Maher[21] nos detalla que los motores de videojuegos son programas que pueden abarcar todos los elementos de un videojuego, conocidos como *assets*. Haciendo esto, el motor carga todos los diferentes archivos con sus diferentes extensiones (*.json, .fbx, etc.*) y los guarda en un solo lugar. Esto no únicamente simplifica el trabajo a los creadores, permitiéndoles trabajar todo el juego con una sola aplicación, sino también reduce el procesamiento de los ordenadores que estuvieran utilizando para diseñar y desarrollar. Además, Rabin[22] hace énfasis en como los componentes preinstalados de los motores de videojuegos remueve la necesidad de escribir código para partes esenciales como los colisionadores de los objetos dentro del juego. De los motores de juegos existentes, tres fueron resaltados por su preferencia encima de los demás: Unity, Unreal Engine, y Godot.

2.4.2. Unity

Unity es la más común de las tres. Creado en el año 2005 para Mac OS X, la idea por su desarrollo fue democratizar la industria de los videojuegos, y permitir que cualquier

persona pueda crear el videojuego que imaginaban. Dos décadas después, se puede decir que este principio ha sido cumplido. Haas[23] nos detalló que Unity era una de las más utilizadas en la industria en general, pero su uso fue mayor entre los desarrolladores *indies*. En el 2019, Foxman[24] descubrió que el 45% de aquellos usaban este motor para crear sus juegos.

2.4.3. C#

Unity utiliza el lenguaje de programación C# como base para sus algoritmos. Creado en el año 2000, como parte del .NET Framework de Microsoft, se ha mantenido como un lenguaje estándar para distintos sistemas, especialmente aquellas que desean correr con el sistema operativo de Windows sin mayor complicación.

Skeet[25] nos indicó que C# se caracteriza por ser un lenguaje tipeado de manera estática; esto quiere decir que el programador determina los tipos de variables que son devueltos por los métodos. Esto permite que se pueda controlar que datos se recibe, en vez de tener que acoplarse a métodos devolviendo tipos de variables específicas, y convertirlas de otra manera.

2.4.4. Unreal Engine

Unreal Engine fue desarrollado por primera vez en 1998, para el juego de disparos en primera persona, Unreal. A pesar de la discontinuidad del juego, el motor siguió adelante, y la empresa que lo diseñó, Epic Games, lo continuó a actualizar para ofrecerlo con licencia de uso a otros estudios.

La versión más actual de Unreal es UE5, lanzado en el año 2022, caracterizado por su alta calidad visual, y el uso de *blueprints*, que permite programar mediante nodos en vez de escribir código. Sin embargo, también se ha criticado a Unreal Engine 5 por el costo de rendimiento que causa que algunos juegos corran a menor fotogramas por segundo.

2.4.5. C++

Unreal Engine corre mediante el lenguaje de programación C++. Este lenguaje fue desarrollado en el año 1985, como una extensión del lenguaje C, con el objetivo de implementar programación orientada a objetos (*Object Oriented Programming*, o OOP por sus siglas en inglés). Igual que C#, a pesar de los años, ha seguido siendo un lenguaje utilizado por varios programadores, y es considerado uno esencial para saber cómo programar.

Stroustrup [26] detalló como C++ sirve para alcanzar un nivel alto de abstracción, un diseño de programación generalmente usado. Además, su capacidad de ser útil con otros estilos de programación por su fuerte revisión de tipo estático le permite acomodarse a distintos tipos de programación. Sin embargo, recalcó que para aprovechar el lenguaje C++ al máximo, se debe tener un alto grado de programación para hacer mayor uso de su abstracción.

2.4.6. Godot

Godot Engine es un motor de código fuente abierto creado en el año 2014, con el objetivo de facilitar el diseño de videojuegos mediante la inclusión de varios lenguajes y componentes, haciendo que el motor sea más accesible para una mayor cantidad de usuarios.

La fortaleza de Godot viene de su capacidad de utilizar varios lenguajes de programación dentro de él. Su lenguaje nativo es GDScript, una variación de Python. Sin embargo, tiene extensiones que permiten que se trabaje con Python natural, además de C#, lo que permite que el usuario no tenga que aprender un lenguaje de programación nuevo para poder usarlo.

2.5. Arquitecturas, modelos, y estándares

La arquitectura principal del juego prueba tuvo una modalidad centralizada, en donde el flujo del juego es dirigido por un componente central, el GameManager. Este *script*

coordinaba los turnos entre el jugador y los enemigos, y administraba poderes que cada uno activaba. Esta arquitectura también implementó un modelo que asimila al de una máquina de estados finitos (*Finite State Machine*, o FSM), por el hecho de que el juego trabaja en estados, según el turno de quien sea, con el GameManager operando el cambio entre estos estados.

Para la generación del mapa, además del modelo de generación procedural, se usó una parte central de la arquitectura de Unity, el paradigma de la arquitectura basada en componentes (*Component-Based Architecture*, o CBA por sus siglas en inglés). Nystrom[27] explica que los componentes sirven para "desacoplar" ciertas partes del código destinado a un mismo objeto en diferentes fragmentos, para que un solo script no haga todo. A partir de ahí, los componentes podrían referenciarse uno al otro para solicitar un cambio de una parte que no puede controlar. En el *script* del algoritmo, se encargó únicamente de instanciar las casillas necesarias para el mapa, y de ahí llamó a diferentes componentes para que se encarguen de cosas como las texturas, o para generar enemigos encima de las casillas, dejando a que el código de la generación sea únicamente para propósitos de la creación del mapa.

Además, en el juego prueba se utilizó el patrón Singleton para la persistencia de objetos del juego entre escenas. Esto sirvió para prevenir la creación repetida de elementos del juego que cumplían el mismo propósito entre cada escena.

2.6. Herramientas de desarrollo

Para este proyecto, se utilizó el motor de videojuegos Unity, en la versión 6.3.13f1. Se ha elegido este motor por ser una de las herramientas más prevalentes en la industria y por su capacidad para manejar mecánicas esenciales y lógica matemática compleja mediante scripts ya preinstalados dentro de él. Además, en comparación con los demás motores con

mayor popularidad, como Unreal Engine y Godot, Unity mantiene una calidad consistente en optimización y actualización de mecánicas fundamentales en sus componentes. La razón detrás de usar la versión 6 fue por su optimización en varios otros aspectos del juego, que ayudó a ahorrar recursos, y simplificar el diseño. Esto permitió mayor agilidad, que encajó perfectamente con el estilo y ritmo de desarrollo que se eligió para este producto dado el tiempo que se tuvo para crearlo.

2.7. Marco conceptual

2.7.1. Generación procedural de contenido (PCG)

La generación procedural tuvo sus raíces en 1978, con el juego Beneath Apple Manor, la cual utilizó este método para poder sobrepasar los límites de capacidad de memoria que tenían los dispositivos electrónicos en ese entonces[28]. Esta fue la razón clave por la cual existía la generación procedural, y es aquella técnica que permitió formar mundos expansivos más y más grandes a pesar de las limitaciones técnicas de una computadora, consola de videojuegos, o dispositivo móvil.

Shariatpour et. al[29] mencionan que la generación procedural consiste en la creación de contenido en una aplicación multimedia a través de un algoritmo que repite una cantidad de veces que puede ser hasta infinita. En el campo de videojuegos, el algoritmo recibe instrucciones de cómo generar las partes de algo mediante una estructura de reglas, comúnmente organizados en un hilo de números (*seeds*). De ahí, siguiendo esas instrucciones, el programa procedió a crear dicho objeto por segmentos, siguiendo especificaciones puestas para determinar cuándo se debe generar otro parte de aquel, y hasta si se debe borrar una parte anterior. Esto permitió crear objetos que parecían ser al azar, y si hay suficientes parámetros, hasta creaba los objetos de tantas maneras que es casi imposible que dos personas reciban la misma experiencia.

Es por esto por lo que Shen & Zhenyuan[28] indican que esta técnica de programación es esencial para videojuegos, especialmente para la creación de mapas de extensiones casi infinitas. No Man's Sky y Minecraft son ejemplos claros del uso de PCG, en donde utilizan técnicas como Perlin Noise para programar inteligencias artificiales que permiten crear estos mapas que generan contenido al azar basado en los parámetros entregados. Por ejemplo, el juego anteriormente mencionado, Minecraft, realiza una generación procedural mediante segmentos llamados chunks. Cada uno genera una cantidad de bloques del juego equivalente a 16 bloques de ancho por 16 bloques de largo, con hasta 256 bloques de profundidad. En cada uno de estos chunks, el código de la generación procedural toma en consideración la rareza de los recursos, y la probabilidad de que entidades no jugables (NPCs) aparezcan, para así crear el mapa de manera única.

2.7.2. Optimización de rendimiento del CPU

Para cualquier proceso, o producto, se debía tener en cuenta siempre la capacidad de poder generar dicho proceso, o elaborar dicho producto, de una manera que gaste menos recursos posibles sin perder la calidad ni eficiencia de aquel. Para realizar esto en videojuegos, se debía depender de la optimización, usando distintas técnicas para poder reducir el poder de procesamiento que debía usar el dispositivo para correr el programa.

Garney & Preisz[30] nos explicaron que los procesos de optimización se dividen según a que aspecto del rendimiento se va a afectar. Para procesos gráficos (GPU), se utilizan técnicas que reducen la calidad de modelos y texturas hasta que el jugador tenga que acercarse a ellos, tales como la compresión de texturas o el Nivel de Detalle (Level of Detail / LOD); hasta en algunas ocasiones, se deshabilitan los modelos que el jugador no puede ver

para así reducir el rendimiento requerido de la computadora, como lo hace la técnica de *culling*.

Hablando de procesos del CPU, es decir, el rendimiento técnico de la computadora, los procesos de optimización dependen principalmente en reducir la cantidad de memoria de acceso rápido (RAM), que requiere utilizar el dispositivo para poder correr el juego de manera fluida. Mientras menos RAM se ocupe, más fácil puede ser reproducido por el sistema, y más sistemas pueden reproducirlos, sin importar si son sistemas más actuales o no. se depende principalmente de la cantidad de scripts que tiene un programa, lo que indica la cantidad de código que debe ejercer la máquina para que corra el juego. También tiene que ver el tipo de código que tienen dichos scripts. Usando técnicas de programación como la persistencia o abstracción permite ahorrar memoria, lo que sirve para reducir el procesamiento que debe ejercer el CPU. También algoritmos como la recursividad, o generación procedural, permite reducir el detalle que debe tener el código para crear cosas específicas, que ayuda a ahorrar aún más memoria[30].

2.7.3. Simplificación de diseño de videojuegos

Muchos videojuegos tuvieron como objetivo lograr que la mayor cantidad de jugadores puedan acceder a sus juegos, sin importar el sistema que tengan. Para que esto se cumpla, existían varias técnicas para simplificar el diseño de los videojuegos para reducir la cantidad de recursos que se requieren para crearlos. Estos métodos incluyeron el diseño modular, que permitió reutilizar modelos usados en niveles anteriores para crear nuevos niveles, compresión de texturas, métodos de programación que reducían el uso del CPU como el encapsulamiento, y *culling*[30].

2.7.4. Mapas de diseño virtual

Mapas han sido esenciales para la navegación del ser humano desde la existencia de las primeras civilizaciones. Hasta en los primeros juegos de mesa, ya había el concepto de mapas dentro de ellas. En el siglo XVII en Francia, un juego de cartas usaba un mapa del mundo conocido, y cada carta correspondía a un territorio. Por lo tanto, quienes jugaban se ubicaban con el terreno puesto de manera visual, y podían coordinar sus acciones[31]. En los videojuegos, los mapas cumplen un rol similar.

Además de su propósito principal, el mapa también sirve para aumentar la inmersión del jugador en la experiencia. Mientras han ido avanzando los años, los mapas fueron aumentando en calidad para poder mejorar esa claridad visual que sirve para que el jugador se sienta realmente dentro del mundo del videojuego. Sin embargo, este aumento también ha producido la consecuencia negativa de causar un estándar en donde los juegos deben tener estos mapas de alto detalle y gran extensión para ser clasificados como “buenos”, según estudios. Esto ha causado que se inflen los precios para la creación de mapas[31].

2.7.5. Diseño modular

Kaźmierczak et. al.[32] nos explicaron que el diseño modular, también conocido como el modularidad en diseño, consiste en la construcción de algo en base a segmentos de partes más sencillas, que son más fáciles de elaborar y pueden ser reutilizables en otros diseños. Originalmente era únicamente para arquitectura en la elaboración de edificios. Sin embargo, Lai & Hu[33] nos aclararon que el concepto del diseño modular podría ser aplicado en muchos otros ámbitos, incluyendo en los videojuegos. En esta área, se usaba principalmente para la creación de niveles, permitiendo que las mismas partes simples que

armaron un nivel podrían ser reutilizados para elaborar un nivel completamente distinto, mediante la reorganización de sus partes en diferentes lugares y orientaciones.

Un ejemplo de esta técnica de diseño en videojuegos se puede ver en el juego Overcooked, en donde los modelos que se puede ver en cada nivel son iguales. Mientras el jugador va avanzando los niveles, los mismos modelos son reorganizados para crear un nivel distinto al anterior, aumentando mecánicas para poder dar más originalidad a la experiencia.

2.7.6. Juegos de estrategia

Entre los géneros de videojuego, los juegos de estrategia siempre se han encontrado entre los más populares. Cada año al menos un nuevo juego de estrategia es lanzado para apelar a este mercado, que, aunque no tiene la misma atracción que tenía en su auge, sigue teniendo alrededor de 100,000 jugadores actuales entre los juegos de estrategia más populares de Steam (Hearts of Iron, Civilization, Total War, Crusader Kings, Europa Universalis, Stellaris, etc.).

Los juegos de estrategia conforman un género de videojuegos que se centran en la planificación táctica y toma de decisiones para poder cumplir los objetivos del juego. Estos juegos empezaron con la consola Invasion, publicado en 1972; sin embargo, el género realmente tomó fuerza con el lanzamiento de Starcraft en 1998, que también causó la separación de aquel en subgéneros, los de estrategia de tiempo real (RTS) y por turnos.

2.8. Relación entre la teoría y la propuesta

La propuesta dependió de las bases asentadas en la teoría para su elaboración. Su principio es matemático, su desarrollo pensado con optimización siendo una prioridad, y su objetivo para apoyar al género de videojuegos TBS. Para programar el algoritmo, se debió conocer a fondo el plano cartesiano y las coordenadas polares para escribir correctamente el código de la colocación de las casillas y su separación para el patrón de teselación. Además,

el propósito de esta generación fue para crear mapas en 3D, en donde el jugador podría seleccionar las casillas e interactuar con ellas, sea con movimiento u otras acciones. Esta mentalidad encajaba de mejor manera con juegos TBS, en donde tener este tipo de mapa, dividido en partes seleccionables y con una visualidad clara, es indispensable para una fluidez de jugabilidad.

Capítulo III

3. Manual De Usuario

Para la verificación del funcionamiento de la propuesta, se aplicaron los resultados en el videojuego “Qhapac Ñan: El Camino del Chasqui”, en el cual se muestran los resultados del funcionamiento del planteamiento del proyecto. Una propuesta realizada por un equipo de desarrolladores, que permite su aplicabilidad y verificación.

3.1. Presentación de la propuesta y perfil de usuario

“Qhapac Ñan: El Camino del Chasqui” es un videojuego de estrategia por turnos con elementos *roguelike* diseñado para centros educativos, donde los jugadores encarnan al astrólogo Chaicó-chasqui. A diferencia de juegos educativos tradicionales, este título se puede jugar varias veces, cada intento teniendo un toque único por su generación procedural y la dificultad progresiva a través de cada nivel, retando a los niños a usar el pensamiento crítico y poderes de los dioses andinos para cumplir una profecía histórica.

El estado actual del juego es un prototipo que contiene la mayoría de las escenas del juego final, siendo estas el menú principal y los tres niveles de jugabilidad. Todas las mecánicas del juego ya están implementadas, tales como la generación del mapa, el movimiento, los enemigos, y los poderes que puede usar el jugador.

El público objetivo del videojuego son niños de 8 a 12 años, quienes tienen un interés por los videojuegos, historias de aventura, aprender de manera interactiva y la tecnología. Los controles son diseñados para apelar a la juventud que interactúa rápidamente con clics o toques a la pantalla, y las instrucciones se acoplan a esta mentalidad, en vez de poner instrucciones largas que obligan al jugador a leer.

3.1.1. Buyer persona

- Nombre: Mateo
- Edad: 10 años
- Ocupación: Estudiante
- Objetivo: Quiere que sus clases no sean tan aburridas y hacer otras actividades. Quiere ver qué pasa si llega al final del mapa.
- Frustraciones: Se frustra si los controles no responden rápido.
- Escenario de uso: Mateo ve el videojuego, toma el ratón y entiende rápidamente que debe mover a Chaicó-chasquí por los hexágonos. Usa la carta de "Viento" para moverse rápido porque quiere ganar a sus amigos que están mirando.

3.1.2. Requisitos mínimos

Tabla 3
Requisitos mínimos del juego

Sistema Operativo	Windows 10 x64 bits
Procesador	Intel Core i3 (5ta Gen.) / AMD Athlon Silver 3050U
Espacio de Almacenamiento	250MB
RAM	4GB
GPU	Intel HD Graphics 5500 / NVIDIA GeForce GTX 600 / AMD Radeon HD 7000

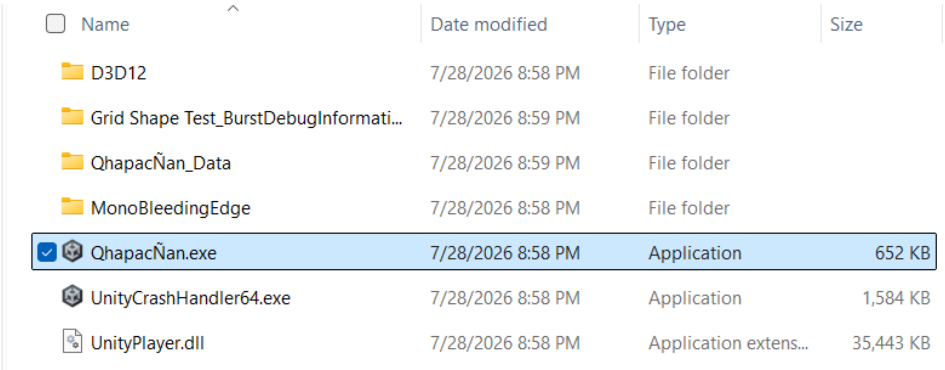
***Nota*.** Descripción de los requisitos mínimos que debe tener un PC para correr el juego Qhapac Ñan

Los componentes de *hardware* mínimos se obtuvieron mediante las funciones de Profiling y Benchmarking de Unity, comparándolo con componentes que cumplen estos requisitos mínimos.

3.2. Acceso e inicio de la experiencia

1. Copiar la carpeta completa del ejecutable del medio de distribución obtenido.
2. Hacer doble clic izquierdo sobre el archivo con nombre “QhapacÑan.exe”.

Figura 6
Contenido de ejecutable del juego.



***Nota*.** Contenido de la carpeta que se obtiene al adquirir el juego Qhapac Ñan. Se resalta el ejecutable que se debe hacer doble clic para iniciar el juego.

3. En la pantalla de inicio, hacer clic sobre el botón que tiene el ícono de “Reproducir”.

Figura 7
Pantalla de inicio de Qhapac Ñan.

Nota. Pantalla de inicio de Qhapac Ñan. Se señala el botón que se debe seleccionar para empezar el juego.

3.3. Navegación y controles

3.3.1. Controles del juego

Tabla 4
Controles del juego

Clic Izquierdo	Seleccionar
Clic Derecho + Mantener	Mover cámara
Rueda de Mouse	Acercar o alejar la cámara del jugador

Nota. Descripción de los controles del juego Qhapac Ñan.

3.3.2. Navegación del programa de generación dentro del editor de Unity

Al obtener el script del programa del algoritmo, antes de usarlo se lo debe importar al proyecto de Unity, y de ahí implementarlo dentro de la escena en donde se desea implementar la generación del mapa.

Figura 8

Muestra de escena con programa implementado.

Nota. Muestra de una escena de un proyecto de Unity con el *script* del programa de generación implementado.

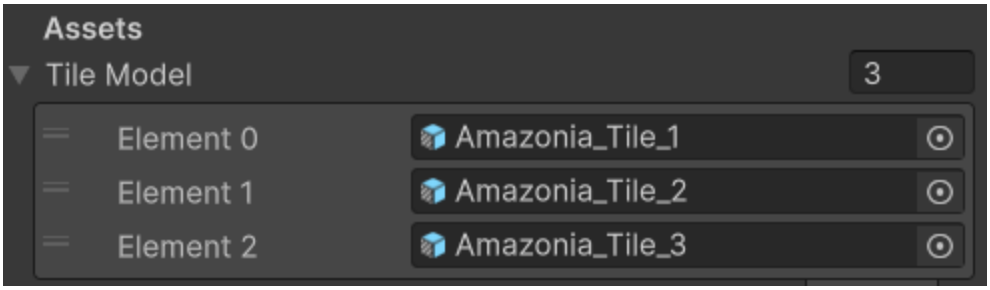
Dentro del proyecto, las variables para el algoritmo pueden ser editadas desde el Inspector, de acuerdo con el diseño deseado del mapa. Los primeros datos presentados son la cantidad de lados de las figuras que conformarán las casillas (*Number Of Sides*), el número de anillos que serán usados para generar el mapa completo (*Rings*), el radio de cada casilla (*Tile Radius*) y el espacio entre cada casilla (*Spacing*).

Figura 9
Variables iniciales del algoritmo.

Nota. Las cuatro variables que aparecen en la parte superior del componente del programa dentro del Inspector de Unity. Todos estos datos son editables desde este espacio, lo que remueve la necesidad de editar el código directamente.

Después de estos datos, se muestra una lista con el nombre de *Tile Model*, en donde el diseñador puede insertar los *prefabs* que se desean usar para la creación del mapa, permitiendo al usuario utilizar su estilo de arte usando este programa.

Figura 10
Lista de prefabs.

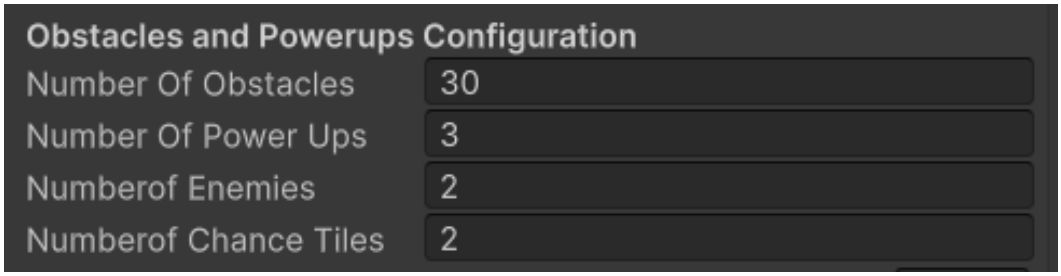


Nota. Lista visible en el componente del programa que permite al usuario insertar los *prefabs* que desea usar en su mapa.

Para el juego prueba, Qhapac Ñan, se insertaron parámetros adicionales para las mecánicas del juego que involucran de manera directa al mapa. Estas variables incluyen la cantidad de obstáculos, enemigos, casillas de poder y de fortuna que tendrá el nivel. Además,

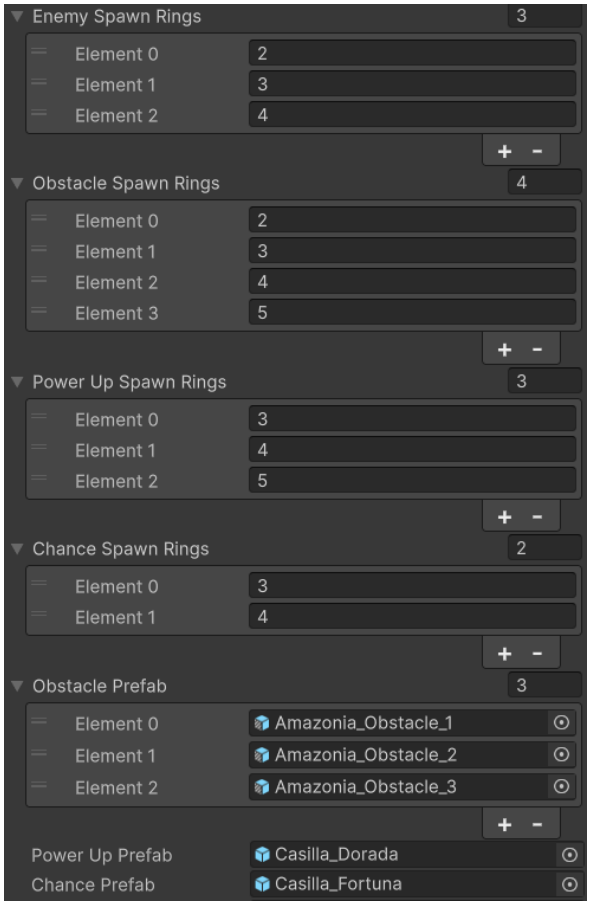
incluye listas para indicar en cuales anillos deben aparecer estos elementos, y en el caso de las casillas especiales, se presentan espacios para poder insertar los modelos que se desean usar para representarlos.

Figura 11
Variables de elementos especiales.



Nota. Variables que el diseñador puede usar para indicar cuantos obstáculos, casillas de poderes, enemigos y casillas de fortuna se tiene que poner en cada nivel, todos editables desde el Inspector.

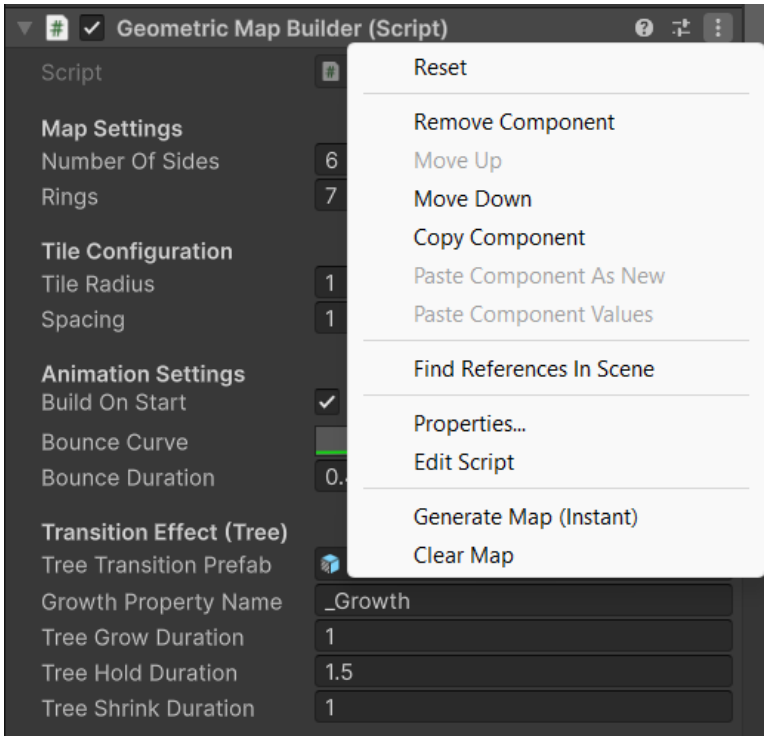
Figura 12
Listas de anillos de instanciado y prefabs.



Nota. Listas en el Inspector donde el usuario puede definir en cuales anillos debe aparecer cada elemento adicional del juego (obstáculos, enemigos, casillas de fortuna y poder.) Además, se muestra una lista para insertar los modelos que se desean usar para las casillas especiales.

En el caso de que el usuario desea visualizar una prueba de la generación del mapa con los parámetros establecidos, se presenta un botón en la parte superior derecha del componente, que indica al programa a realizar una generación de manera inmediata dentro del editor. Esto permite realizar ediciones o cambios como sean necesarios, sin tener que correr el modo prueba dentro del editor cada vez que se desea ver como el mapa será creado.

Figura 13
Comando de generación instantánea.



Nota. Presentación de la manera de poder generar el mapa de manera inmediata, mediante la selección de la opción "*Generate Map (Instant)*".

Figura 14

Muestra de generación instantánea.

Nota. Mapa creado en el editor mediante el comando de generación instantáneo. Se puede percatar que el editor no está en modo prueba.

3.3.3. Navegación del juego

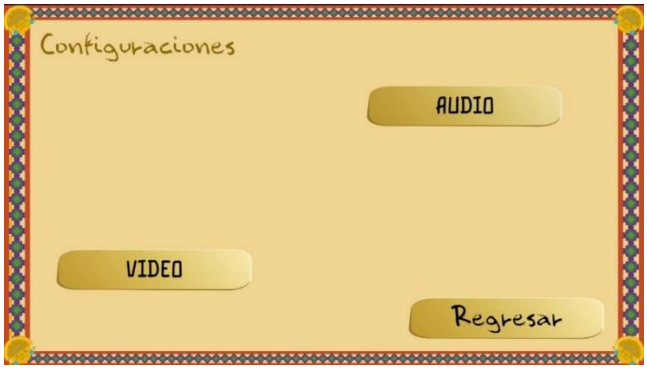
El usuario empieza el juego en el menú principal, donde tiene los botones de empezar el juego, ver y cambiar los ajustes, y salir del programa.

Figura 15
Explicación de pantalla de inicio de Qhapac Ñan.

***Nota*.** Pantalla de inicio del juego Qhapac Ñan. Se señalan los botones para empezar el juego, las configuraciones, y para salir del programa.

Presionando el botón de ajustes abre un menú en donde el jugador puede elegir entre cambiar la configuración de audio o video del juego. La opción de audio permite editar el volumen maestro del juego, y las opciones de configuración de video permiten reducir la cantidad máxima de fotogramas por segundo (FPS), cambiar la resolución de pantalla del juego, activar o desactivar VSync y cambiar el esquema de colores del juego en el caso de daltonismo por parte del usuario.

Figura 16
Menú de opciones de configuración del juego.



***Nota*.** El menú que aparece al seleccionar el botón de configuraciones en la pantalla de inicio del juego.

Figura 17
Pantalla de cambio de volumen maestro del juego.

***Nota*.** Menú que aparece al seleccionar la opción de configuración de audio.

Figura 18
Menú de opciones de configuración de video.

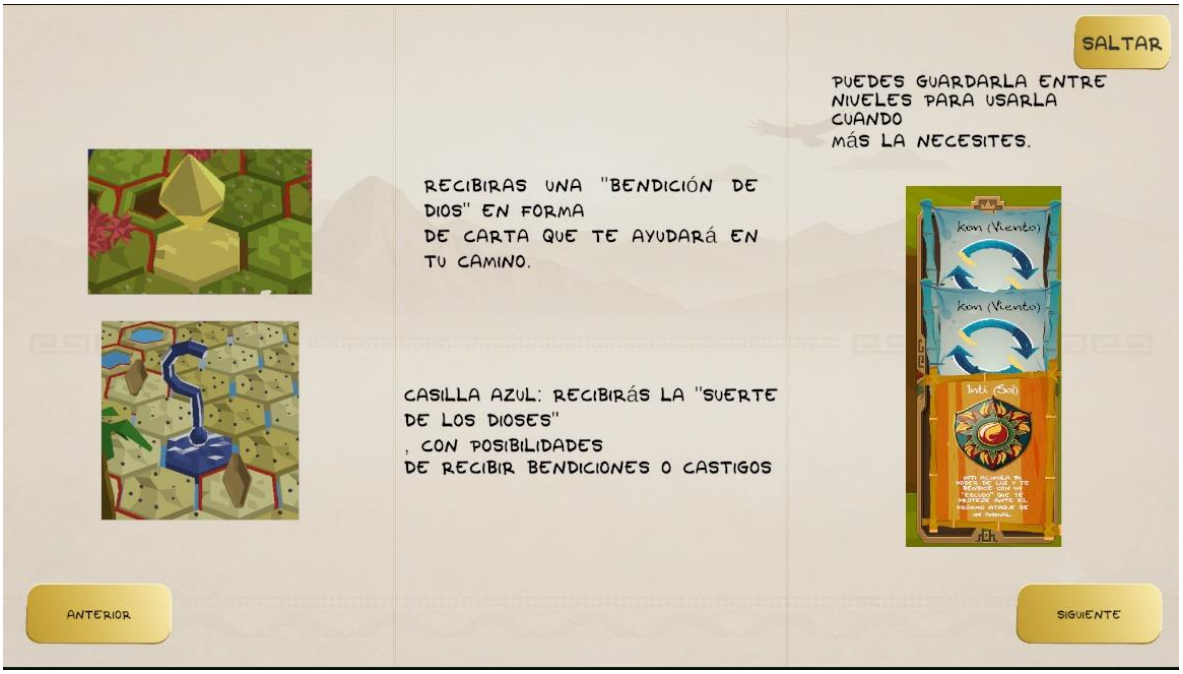
Nota. Menú de opciones al seleccionar la opción de configuración de video.

Al presionar el botón de jugar, se presenta un tutorial para que el jugador entienda la historia y el objetivo del juego. Tiene botones para avanzar y retroceder por el tutorial, y un botón que le permite saltar el tutorial.

Figura 19
Página #1 del tutorial.

Nota. La primera página del tutorial que aparece al iniciar el juego.

Figura 20
Página #2 del tutorial.



Nota. La segunda página del tutorial, que sale al presionar el botón “Siguiente” en la primera página.

Figura 21
Página #3 del tutorial.

Nota. La tercera página del tutorial, que sale al presionar “Siguiente” en la segunda página.

Dentro del juego, el usuario puede seleccionar una cantidad determinada de casillas para avanzar hacia el centro del mapa. Al llegar a una casilla dorada, aparecerán tres cartas, de las cuales el jugador podrá elegir uno. Cada carta tiene un poder que ayudará al jugador en su objetivo. Si aterriza sobre una casilla azul, le saldrán dos cartas iguales, de las cuales elegirá uno. Esta carta le puede dar inmediatamente un efecto positivo o negativo.

Figura 22
Selección de casilla

***Nota*.** Representación visual de la selección de casilla para el movimiento del jugador. Se resalta la casilla que el jugador vaya a seleccionar para dar claridad visual.

Figura 23
Panel de cartas de poder.

Nota. El panel de cartas que le sale al jugador al llegar a una casilla dorada. Tiene 3 cartas para poder seleccionar, y cada carta tiene el nombre de una deidad Inca, además del efecto que otorga.

Figura 24
Panel de cartas de fortuna.

Nota. El panel de cartas que aparece al jugador cuando aterriza en una casilla azul. Le permite elegir de entre dos cartas con efectos desconocidos.

Al llegar a las casillas que están en el centro del mapa, se carga el siguiente nivel. Si es que el jugador es derrotado, aparece un menú que le da contexto adicional y le permite reiniciar el juego o volver al menú principal.

Figura 25
Pantalla de derrota.

***Nota*.** Pantalla que aparece al jugador cuando es atacado por un animal sin tener un escudo activado. Le permite volver al menú principal, o reiniciar el juego.

3.4. Funciones e interacciones disponibles

Dentro del programa existen funciones que permiten al jugador entender los efectos de sus acciones. Estos principalmente se observan cuando el jugador activa una carta de poder, y tiene una semejanza al dominio de la deidad quien está en la carta que el usuario activó. Usando la carta de Inti, aparece un escudo como efecto visual; este mismo efecto aparece cuando el enemigo ataca al jugador mientras tenga un escudo activo. Además, al cambiar una casilla de camino a obstáculo (el efecto de Pachamama), o viceversa (Pachacamac), aparece una planta creciendo y el suelo derrumbándose respectivamente. Cuando el usuario usa el poder de Kon, al moverse, se puede notar un efecto de viento

saliendo del jugador. El efecto del poder de Illapa es representado visualmente por un rayo que cae al piso para llevar al jugador a la casilla a la cual desea teleportarse. Finalmente, al dormir un enemigo con el poder de Mama Quilla, el enemigo tendrá una indicación visual de aquel efecto activándose.

Figura 26
Activación del poder de la carta Inti.

***Nota*.** Efecto visual que aparece al activar el poder de la carta Inti.

Figura 27
Activación del poder de la carta Pachamama

***Nota*.** Representación visual de la activación del poder de la carta Pachamama.

Figura 28
Activación del poder de la carta Pachacamac.

***Nota*.** Efecto visual que aparece cuando el jugador activa el poder de la carta Pachacamac.

Figura 29
Efecto visual del poder de la carta Kon.

***Nota*.** Efecto visual que el jugador visualiza al activar el poder de la carta de Kon.

Figura 30
Efecto de la carta de Illapa.

***Nota*.** Efecto de rayo que aparece al activar el poder de la carta de Illapa.

Figura 31
Efecto visual la carta de Mama Quilla.

***Nota*.** Efecto visual puesto a los enemigos que sufren los efectos del poder de la carta de Mama Quilla.

Capítulo IV

4. Manual De Programación

4.1. Arquitectura general del sistema

El juego prueba utiliza una arquitectura de Gestor Global, centrado en el *script* GameManager.cs. En aquel se recolecta la información del turno del jugador y del enemigo, y de ahí va editando los elementos respectivos del juego según los datos y argumentos enviados a él.

Estas ediciones se realizan mediante la arquitectura Maestro-Esclavo (*Master-Slave*), en donde un componente dirige a los demás que tareas se deben realizar. Esto se implementa en el algoritmo del juego, en donde el "maestro", GameManager.cs, envía las tareas a los demás *scripts* del juego, quienes serían los esclavos, mediante el llamado a métodos dentro del componente requerido.

4.2. Tecnologías, motores y herramientas utilizadas

Este proyecto utilizó Unity 6 como el motor en la cual se trabajó el programa, por la popularidad de aquel en la industria. En comparación con otros motores, Unity tiene una mayor consistencia en rendimiento, y una actualización continua de sus herramientas que no poseen los otros motores populares. Usando la versión 6 permitió mayor accesibilidad con paquetes adicionales, además de asegurar su uso por una mayor cantidad de tiempo antes de tener que actualizarlo para nuevas versiones, a contrario con la versión 2021 o 2022.

Además, para el juego prueba se usaron componentes adicionales para reducir la cantidad de *scripts* necesarias para desarrollar el juego, tales como Cinemachine para el

funcionamiento de la cámara, y el nuevo System Input de Unity, que ayudó a simplificar la asignación de *inputs* de parte del usuario a acciones que sucedieran en el juego.

4.3. Organización técnica del proyecto

Debido a que toda la generación sucedía desde un solo *script*, este no requería mayor organización; sin embargo, dentro del juego prueba, se han distribuido los *assets*, incluyendo los *scripts*, para que fuera fácil encontrar un elemento específico del juego que el diseñador necesitara para trabajar en su parte del proyecto.

En el caso de la programación, todos los archivos .cs se almacenaron dentro de la carpeta de "Assets", y ahí guardado en otra carpeta denominada "Scripts". A partir de aquí, se subdividieron los archivos según a cuál elemento específico del juego afectaban. Por ejemplo, los *scripts* que directamente influyeron al jugador fueron a la carpeta llamada "Player". Aquellos que contenían el código que afectaba al mapa, como GeometricMapBuilder.cs fueron a la carpeta de "Mapa". Los *scripts* que daban la funcionalidad de los enemigos fueron a la carpeta de "Enemy".

4.4. Arquitectura de clases y componentes

4.4.1. GeometricMapBuilder.cs

Este *script* es el que controla la generación del mapa, además de ser la parte central de este proyecto. A partir de este código, las demás mecánicas del juego obtienen la información acerca de las casillas necesarias para activar sus métodos; y si alguna mecánica del juego requiere editar una de las casillas, informa a este script para que lo edite como sea necesario.

En el juego prueba, existen diferentes tipos de casillas: casillas normales por donde puede avanzar el jugador, obstáculos por donde no puede pasar, casillas de "Powerup", donde al aterrizar en ellas, se puede obtener una carta para activar un poder para ayudarle durante

el juego, y finalmente una de fortuna, dónde tiene que elegir una carta al azar entre dos, y le puede salir un efecto positivo o negativo. Todas estas casillas son almacenadas en listas distintas para que el programa pueda fácilmente encontrar una basada en la etiqueta y el índice dentro de la lista.

4.4.2. Spawner.cs

Este *script* tiene la funcionalidad de instanciar al jugador y los enemigos para cada nivel cuando el mapa sea creado. GeometricMapBuilder.cs informa cuando haya terminado de generar el mapa, y envía a este componente las posibles casillas donde el jugador y los enemigos se pueden instanciar, determinados de manera aleatoria dentro en anillos determinados por el diseñador en el inspector, aprovechando el uso de la programación mediante componentes para que sea más accesible editar las variables en cada nivel, que permite niveles más variadas.

4.4.3. GameManager.cs

En este *script*, se coordinan los turnos del jugador y los enemigos, además de administrar los poderes que activa el jugador, informando a los componentes encargados de que editar la parte del juego afectado. Algunos de estos poderes requerían editar las casillas del mapa, por lo que GameManager.cs también interactúa con GeometricMapBuilder.cs para que acceda a las listas de los distintos tipos de casillas para cambiar donde sea necesario. Este *script* tiene persistencia mediante el patrón Singleton; esto significa que el GameObject que tenga a GameManager.cs como componente será la misma a través de todos los niveles, lo que reduce la cantidad de procesamiento.

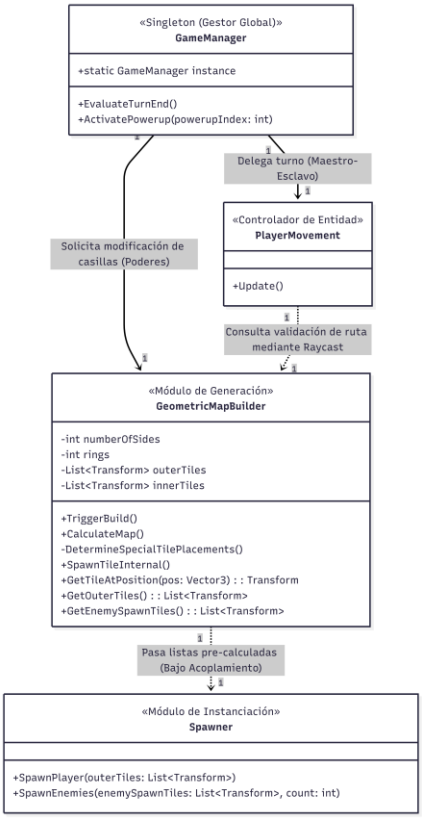
4.4.4 PlayerMovement.cs

Este *script* controla el movimiento del jugador. Como el jugador tiene que moverse a través de las casillas del juego, cada vez que el usuario pasa el cursor encima de una, un método de este código llama a GeometricMapBuilder.cs para que encuentre la casilla donde está el cursor, e indique al jugador si es que puede pasar por esa casilla, resaltándolo con el material de la casilla.

4.4.4. Acoplamiento entre *scripts*

Las funciones relevantes a la creación del mapa y la administración de sus casillas son únicamente de GeometricMapBuilder.cs. Al requerir la información sobre una casilla, los otros *scripts* acuden a este algoritmo, tal como cuando PlayerMovement.cs desea saber si la casilla seleccionada por el cursor es válida para que el jugador mueva hacia ella. Además, cada *script* realiza su propio trabajo, previniendo una red de métodos entre distintos componentes: GeometricMapBuilder.cs se encarga del mapa, Spawner.cs se encarga de instanciar entidades, y así respectivamente.

Figura 32
Diagrama de clases de arquitectura general.



***Nota*.** Diagrama de clases que muestra la estructura general del algoritmo de generación del mapa, en conjunto con las mecánicas del juego de Qhapac Ñan.

4.5. Desarrollo de sistemas principales

El sistema de la generación del mapa inicia con el método TriggerBuild()., en donde empieza a calcular el mapa mediante CalculateMap(). En esta parte del *script*, se coge la variable de la cantidad de lados que ingresó el usuario (NumberOfSides), y dependiendo del valor, se calcula el mapa teniendo en mente la figura correspondiente. Para el juego prueba se usaron 6 lados para crear un mapa hexagonal.

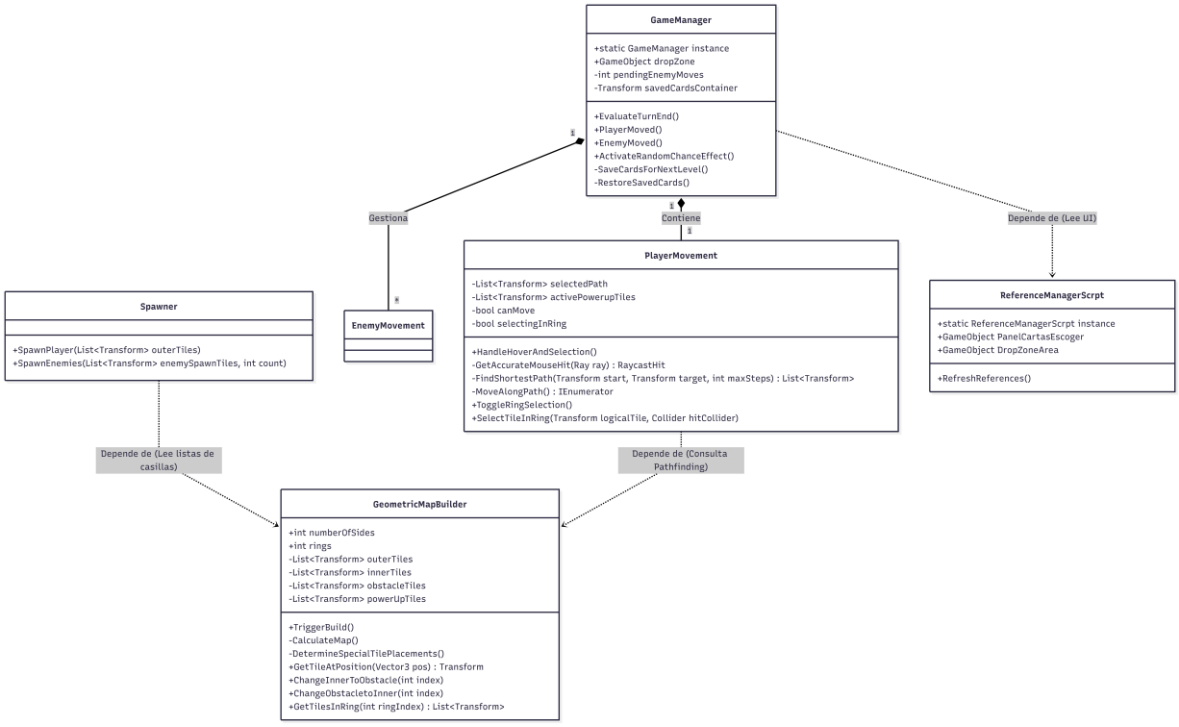
El método para el mapa hexagonal primero calcula la apotema usando el radio de la casilla ingresado por el usuario. A partir de este valor se pueden calcular los puntos de los

ángulos, y de ahí usar un patrón anillar para determinar por cual lado se tendrá que instanciar la siguiente casilla. Este espacio se guarda mediante el método QueueTile(), que recibe el vector de la posición calculada como argumento. También se almacena a cuál anillo pertenece la casilla, que sirve para asignarlas de manera correcta después. Este algoritmo continúa hasta que se hayan establecido todas las posiciones para instanciar las casillas; sin embargo, antes de que esto suceda, el programa tiene que determinar en cuales espacios van cuales casillas. Para esto sirve el método DetermineSpecialTilePlacements().

Esta parte del *script* ayuda a asignar las casillas a los espacios determinados en CalculateMap()., usando las variables de la cantidad de casillas especiales que debe aparecer en cada nivel (numberOfObstacles, numberOfPowerups, numberOfChanceTiles) y las listas de las variables de los anillos en donde deben aparecer estas casillas especiales para de ahí, llamar al método AssignSpecialTile(). Todos los argumentos mencionados anteriormente son editables desde el inspector, para mayor facilidad del diseñador.

A partir de ahí, el programa empieza a generar las casillas con el método SpawnTileInternal(), los parámetros ya establecidos por los métodos anteriores, con excepción del tiempo que pasa entre el instanciado de cada casilla (spawnDelay), que también se puede editar desde el inspector. El resultado es una generación fluida, en donde todos los tipos de casillas aparecen durante la creación del mapa, en sus posiciones correctas.

Figura 33
Diagrama de clases del sistema principal.



***Nota*.** Diagrama de clases mostrando la estructura del sistema principal del desarrollo de la generación del mapa.

4.6. Desarrollo de mecánicas del videojuego

A partir de la generación del mapa, se empieza a calcular las posiciones del jugador y de los enemigos. Para el juego prueba, se determinó que el jugador debería empezar en una de las casillas esquineras del mapa. Es por ello por lo que se llama al método GetOuterTiles(). Este calcula la distancia entre el centro del mapa y las casillas. Al encontrar las casillas que tienen la mayor distancia, los guarda en una lista, que es pasada al *script* Spawner.cs, como argumento del método SpawnPlayer(). Aquí es donde el código elige al azar una de estas casillas para que el jugador aparezca ahí.

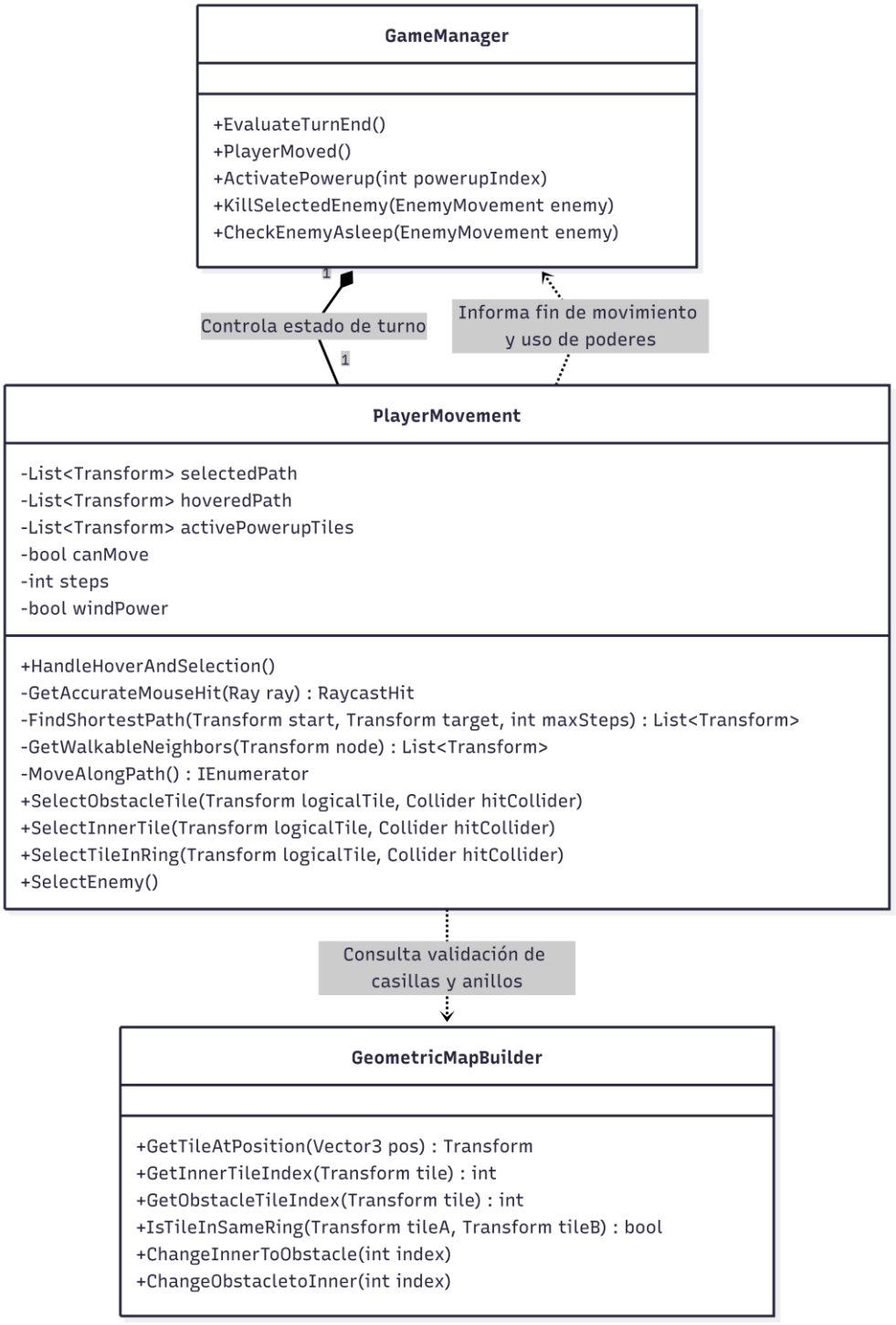
También procede a calcular en donde pueden aparecer los enemigos. Dado que ya se ha guardado una lista de las casillas normales, se usa esta lista para el método GetEnemySpawnTiles(), donde se elige al azar desde las casillas normales en los anillos en donde se ha indicado que algún enemigo puede aparecer. La cantidad de enemigos (numberOfEnemies), al igual que la lista de los anillos en donde pueden ser instanciados, son editables desde el inspector. Cuando el programa haya elegido la cantidad adecuada de casillas, se pasa esta nueva lista a Spawner.cs mediante el método SpawnEnemies().

Cuando el jugador desea moverse, el *script* PlayerMovement.cs detecta donde se encuentra el cursor del ratón mediante el método Update(). Al pasar por una casilla, el código indica a GeometricMapBuilder.cs la posición, y lo usa como argumento para que este *script* llame al método GetTileAtPosition(). Se devuelve la casilla correspondiente, y ahí se determina si es que la casilla es válida para caminar encima o no, mediante las etiquetas de las casillas. Si es así, se resalta la casilla para dar una indicación clara al jugador que puede moverse ahí. De esta manera, se usan las listas de las casillas guardadas por GeometricMapBuilder.cs para mantener un ritmo de juego fluido.

En GameManager.cs, existe la administración de los poderes que puede activar el jugador durante los niveles, en el método ActivatePowerup(). Algunos de estos poderes causan que una casilla de camino se convierta a un obstáculo, o vice-versa. Ambos son posibles gracias a la misma selección del cursor que permite al jugador moverse. Mediante un *toggle* de un *bool*, se puede editar que tiene que detectar el cursor para poder determinar que puede seleccionar el jugador. Dos de estos *bools* permiten seleccionar cualquier casilla, aunque no sea viable para movimiento, y de ahí, dependiendo si es una casilla normal o de obstáculo, se convierte de uno al otro.

Figura 34

Diagrama de clases de mecánicas.



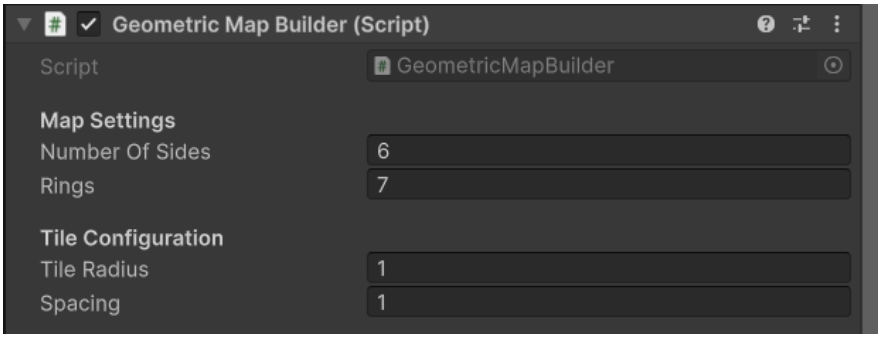
***Nota*.** Diagrama de clases que detalla la estructura de las mecánicas del videojuego Qhapac Ñan.

4.7. Integración de sistemas

El proceso resultante del desarrollo de la mecánica, y su interacción con las demás partes del juego prueba, se explica de la siguiente manera:

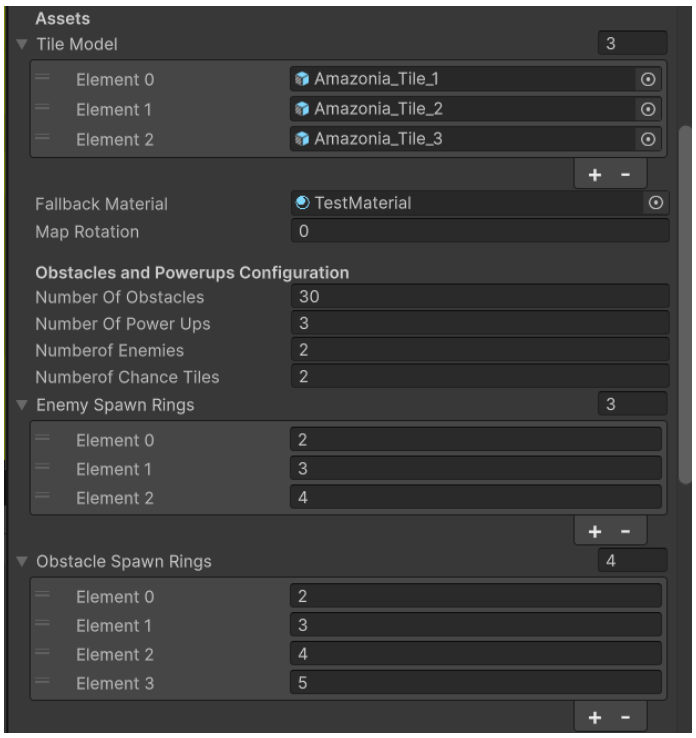
1) El diseñador ingresa los valores de las variables que desea para la generación del mapa, como la cantidad de lados de las figuras que conforman las casillas, el tiempo de demora entre la generación de cada casilla y la cantidad de anillos que desea que contenga el mapa en total. Además, si es que el diseñador desea, ingresa un modelo de la cual se debían crear las casillas. Dentro del contexto del juego prueba, el diseñador también ingresaría cuantos obstáculos, casillas especiales y enemigos deseaba en el nivel, además de en cuales anillos debían aparecer.

Figura 35
Variables para la generación del mapa.



***Nota*.** Variables visibles desde el Inspector de Unity, que permite editar de manera fácil los parámetros para la generación del mapa.

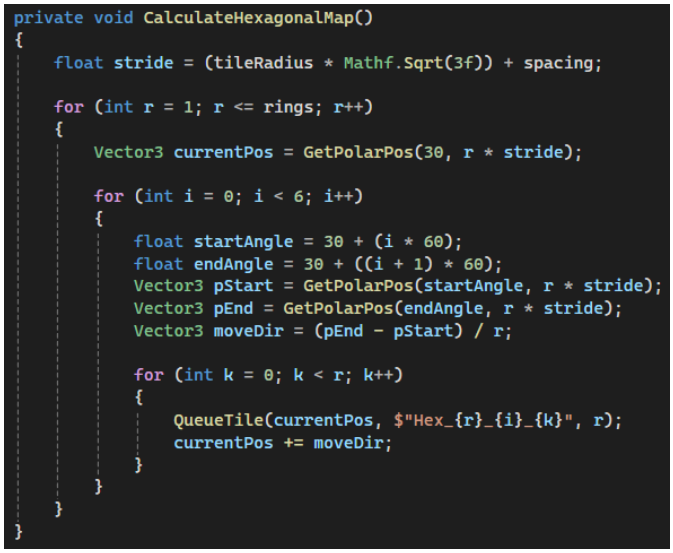
Figura 36
Lista en el Inspector de modelos prefabricados.



***Nota*.** Lista de elementos en el Inspector de Unity que permite al diseñador insertar los *prefabs* y variables de los elementos que deben ser instanciados en el mapa.

2) Al cargarse la escena de Unity, empezando el nivel, el algoritmo obtenía los datos ingresados por el diseñador, y los utilizaba para calcular las posiciones de las casillas para crear el mapa, además de determinar de manera aleatoria cual tipo de casilla va en cual posición, y guardarlo para la generación posterior.

Figura 37
Método de cálculo de posición de casillas.



```
private void CalculateHexagonalMap()
{
    float stride = (tileRadius * Mathf.Sqrt(3f)) + spacing;

    for (int r = 1; r <= rings; r++)
    {
        Vector3 currentPos = GetPolarPos(30, r * stride);

        for (int i = 0; i < 6; i++)
        {
            float startAngle = 30 + (i * 60);
            float endAngle = 30 + ((i + 1) * 60);
            Vector3 pStart = GetPolarPos(startAngle, r * stride);
            Vector3 pEnd = GetPolarPos(endAngle, r * stride);
            Vector3 moveDir = (pEnd - pStart) / r;

            for (int k = 0; k < r; k++)
            {
                QueueTile(currentPos, $"Hex_{r}_{i}_{k}", r);
                currentPos += moveDir;
            }
        }
    }
}
```

Nota. Captura de pantalla del método dentro de GeometricMapBuilder.cs que permite calcular las posiciones de las casillas para generar el mapa.

3) Se empezaba a generar el mapa de manera que el usuario pudiera visualizarlo, utilizando los modelos dados por el diseñador dentro del Inspector para sus casillas respectivas. En el caso de no tener un modelo, el programa creaba planos con forma de figuras geométricas regulares de la cantidad de lados ingresados por el diseñador.

Figura 38
Muestra del Nivel 1.

***Nota*.** Muestra de una posible generación del mapa para el Nivel 1 del juego Qhapac Ñan.

4) Al finalizar la generación, el programa procedió a encontrar posiciones válidas para el instanciado del jugador y de los enemigos. Elegía las casillas de manera aleatoria entre las validas, y los enviaba al *script* respectivo para que puedan aparecer dentro del juego.

Figura 39
Algoritmo para Spawner.cs.

```
public List<Transform> GetEnemySpawnTiles()
{
    List<Transform> spawnTiles = new List<Transform>();
    bool useRingFilter = enemySpawnRings != null && enemySpawnRings.Length > 0;

    if (useRingFilter)
    {
        foreach (int ringIndex in enemySpawnRings)
        {
            if (ringIndex <= 1 || ringIndex >= tilesByRing.Count) continue;

            foreach (Transform tile in tilesByRing[ringIndex])
            {
                if (!tile.CompareTag("Obstacle"))
                    spawnTiles.Add(tile);
            }
        }
    }
    else
    {
        List<Transform> outerTilesList = GetOuterTiles();
        foreach (Transform tile in mapContainer)
        {
            if (GetRingIndexOfTile(tile) <= 1) continue;

            if (!outerTilesList.Contains(tile) && !tile.CompareTag("Obstacle") && tile.name != "Center")
                spawnTiles.Add(tile);
        }
    }

    for (int i = spawnTiles.Count - 1; i > 0; i--)
    {
        int j = Random.Range(0, i + 1);
        Transform temp = spawnTiles[i];
        spawnTiles[i] = spawnTiles[j];
        spawnTiles[j] = temp;
    }

    return spawnTiles.GetRange(0, Mathf.Min(numberofEnemies, spawnTiles.Count));
}
```

Nota. Captura de pantalla del método de GeometricMapBuilder.cs que obtiene las posibles casillas donde los enemigos pueden ser instanciados. Estas casillas son pasadas al Spawner.cs.

5) Cuando el jugador mueve el cursor, el programa se informa acerca de sobre cual casilla estaba. Busca en la lista del tipo de casilla respectiva cual casilla es, y confirma la validez de la casilla según la acción que está haciendo el jugador (moverse, usar un poder), y le informaba al *script* de movimiento del jugador para que pudiera dar una indicación visual al usuario, confirmando que podría interactuar con esa casilla.

Figura 40
Confirmación de camino por parte del usuario.

Nota. Selección de una casilla por parte del usuario para confirmar el camino que va a recorrer el jugador. Se resalta de verde para mayor claridad visual.

4.8. Integración entre diseño, arte y programación

Para integrar los elementos de arte del mapa, se recurrió a las variables de GameObject dentro de GeometricMapBuilder.cs, las cuales eran visibles en el inspector. Los modelos asignados eran creados por el artista 3D y eran exportados de su programa de modelado como un archivo. *fbx*. A partir de ahí eran importados a Unity y hechos un *prefab*, un GameObject prefabricado que podría ser usado como el modelo final.

4.9. Gestión de datos y persistencia

El *script* GeometricMapBuilder.cs contenía listas para poder almacenar las casillas con su propia estructura, que incluía el índice del anillo donde se encontraban. Estos datos debían ser almacenados para poder ser llamados para mecánicas que involucraban a las casillas, tales como el instanciado de los enemigos y jugador. Separando las listas de las

casillas según el tipo permitió reducir la cantidad de casillas que tenía que revisar los métodos para actuar en base a ellas, lo que agilizaba el tiempo y reducía el procesamiento requerido.

Figura 41
Struct de las casillas.

```
private enum SpecialTileType { None, Obstacle, PowerUp, Chance }

8 references
private struct TileRequest
{
    public Vector3 position;
    public string name;
    public int ring;
    public SpecialTileType specialType;

    1 reference
    public TileRequest(Vector3 p, string n, int ring)
    {
        position = p;
        name = n;
        this.ring = ring;
        specialType = SpecialTileType.None;
    }
}
private List<TileRequest> buildQueue = new List<TileRequest>();
```

Nota. Elemento *struct* que establece los parámetros para la creación de una casilla, lo cual es guardado en conjunto en una lista, para mayor accesibilidad al generar el mapa.

Con el *script* GameManager.cs, existía la persistencia a través del patrón de diseño Singleton. Esta técnica de programación permitió que el GameObject que contiene a este *script* siguiera con el programa a través de las escenas. Con esto, no se tendría que crear el mismo objeto que cumplía la misma función en cada escena, reduciendo el peso de las escenas, y, por lo tanto, del juego prueba.

Figura 42
Método de Awake de persistencia.

```
void Awake()
{

    if (instance == null)
    {
        instance = this;
        DontDestroyOnLoad(gameObject);

        // Create the hidden vault to store cards when the Canvas is destroyed
        GameObject containerObj = new GameObject("SavedCardsContainer");
        containerObj.transform.SetParent(this.transform);
        savedCardsContainer = containerObj.transform;
    }
    else
    {
        Destroy(gameObject);
    }
}
```

Nota. Captura de pantalla del método Awake de GameManager.cs que establece la persistencia de este GameObject.

Figura 43
Carga de primer nivel en el editor.

Nota. Carga del Nivel 1 de Qhapac Ñan en el editor de Unity, donde se puede presenciar y verificar la presencia de la persistencia del GameManager.

4.10. Configuración de escenas y flujo del videojuego

El juego empieza con un menú principal para que el usuario pueda confirmar su deseo de iniciar mediante la presión de un botón. Al momento de hacerlo, se carga el primer nivel, y el jugador procede con la experiencia. Al llegar al centro del mapa, inmediatamente se carga el siguiente nivel, y si es que tenía poderes disponibles que no usó en el nivel anterior, estos se llevan hacia el siguiente.

Si el jugador pierde, aparece un panel indicándole que ha sido derrotado, y ofreciéndole una oportunidad para reiniciar el juego o volver al menú principal. Haciendo clic al botón de volver al menú principal le lleva a la primera escena cuando empezó el juego por primera vez; presionando el botón de reiniciar el juego regresa a la escena del primer nivel, creando el mapa desde cero, para dar al usuario un toque único a cada intento.

4.11. Pruebas técnicas y validación

Para mostrar la viabilidad del generador para crear mapas únicos para cada usuario, y probar su uso en el campo de los videojuegos, el juego de prueba desarrollado fue presentado ante niños del rango de edad en acorde con el público objetivo del juego entre 8 y 12 años. Se utilizaron cuatro laptops, cada uno corriendo el programa para poder realizar las pruebas con la mayor cantidad de jóvenes posible. Eran instruidos a intentar pasar los tres niveles del juego, e intentarlo la cantidad de veces que pueda dentro de un tiempo límite de 5 minutos. La validación se realizó a través de observación directa con el grupo objetivo del juego y para confirmar la utilidad del programa se aplicaron encuestas a los desarrolladores.

Figura 44
Prueba de juego con jóvenes.

***Nota*.** Prueba de juego con jóvenes dentro del rango de edad del público objetivo.

4.12. Optimización y rendimiento

Para mejorar el rendimiento del *software*, se implementó el uso de fórmulas matemáticas para calcular las posiciones primero en vez de instanciar la casilla y determinar la posición de cada casilla de manera singular. Esto ayudaba a que el sistema solo se enfoque en una tarea a la vez, lo que le permitía dedicar más procesamiento a cada una, y, por lo tanto, terminarla de una manera eficaz. Además, el uso de un *struct* para almacenar datos específicos de cada casilla, y listas para guardarlas en la memoria de manera separada según su tipo permitió agilizar los tiempos de búsqueda de las casillas para las mecánicas, y removió la necesidad de ver cada casilla del mapa para ver su función.

Dentro del contexto del juego prueba, se utilizó el patrón Singleton dentro del GameManager para mantener persistencia y prevenir la repetición de este elemento del juego,

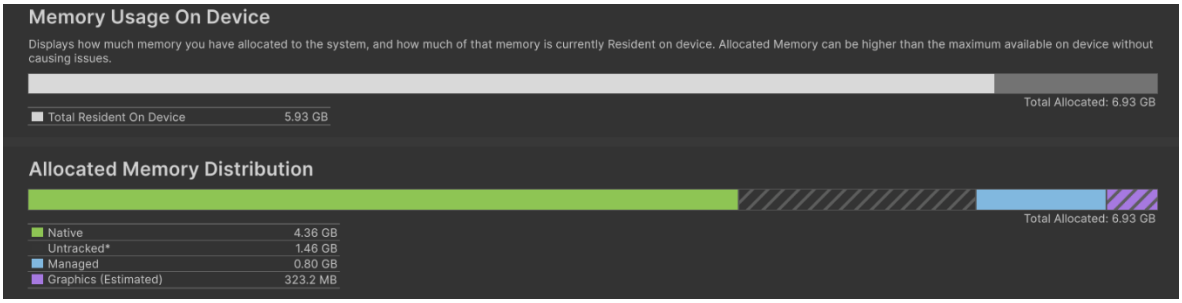
la cual seguía siendo la misma a través de todas las escenas. Además, se han utilizado modelos con una cantidad baja de polígonos, conocido como modelos *low-poly*, para prevenir una sobrecarga del procesador gráfico del dispositivo que reproduzca el juego.

Con estas técnicas, se logró que el juego corra con 2GB de RAM, entre memoria manejada por el juego (*Managed*) y la memoria dedicada a gráficas (*Graphics*), una cantidad que va a la par con juegos similares como Shadow Empire. Sin embargo, cabe recalcar que la mayoría de estos juegos, incluyendo a Shadow Empire, son en 2D, mientras que Qhapac Ñan es un proyecto en 3D, lo que demuestra la optimización del juego prueba.

Dada la variable de *SpawnDelay*, también se puede optimizar el tiempo, acortando cuanto demora que se genere cada mapa con el objetivo de mantener el enganche al jugador. Para determinar los tiempos de generación, se mandó al algoritmo a crear un mapa con 7 anillos, 30 obstáculos, 3 casillas de poder y 2 casillas de fortuna, con la variable de *Spacing* puesto en 1, y el valor de *SpawnDelay* de 0.025. El mapa fue generado en 5.31 segundos, y en total el juego consumió 800MB solo en procesos de *scripts*.

Figura 45

Muestra del uso de memoria para el proyecto de prueba: Qhapac Ñan.



Nota. Segmentación del uso de memoria dentro del proyecto Qhapac Ñan. Cabe recalcar que únicamente la memoria administrada (*Managed*), y la memoria estimada de gráficas (*Graphics*) quienes son los que determinan la cantidad de RAM utilizada dentro del ejecutable. El resto es usado en el editor de Unity.

4.13. Despliegue y compilación

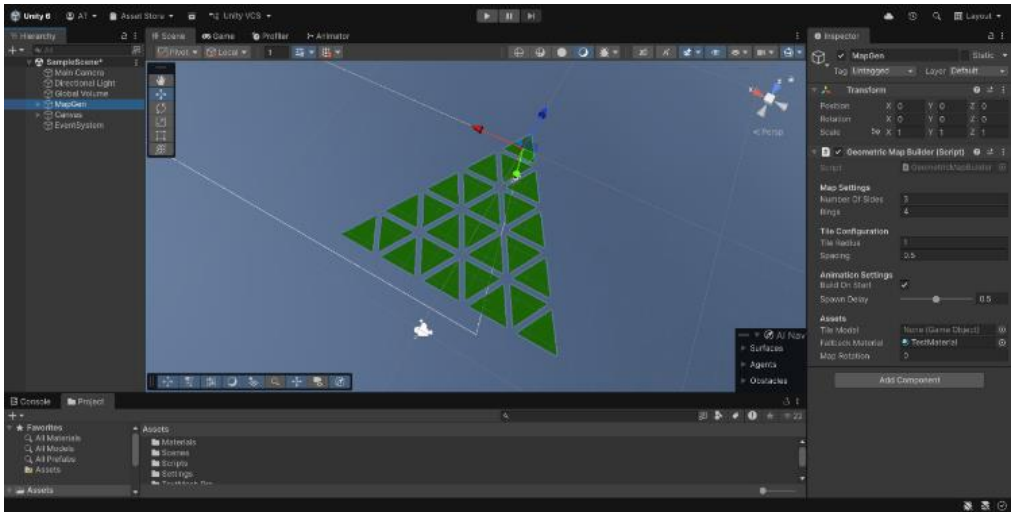
El juego prueba fue compilado para PC, con el sistema operativo de Windows. El tiempo de compilación fue de 1 minuto y 55 segundos, un tiempo óptimo considerando el promedio de tiempo de primeras compilaciones de proyectos en Unity (2 minutos). En un futuro de distribución del producto, la plataforma ideal para aquello sería por Itch.io, la razón siendo su atracción de jugadores por contenido de pequeños desarrolladores, y un resalto de ellos en esta plataforma.

4.14. Escalabilidad y mantenimiento

Este algoritmo de generación tuvo la ventaja de incluir distintas figuras geométricas como viables opciones para la creación del mapa, y las variables que determinan la forma del mapa eran completamente editables desde el inspector. Esto le daba una versatilidad alta, ya que, aunque se haya usado un mapa hexagonal como la prueba para este proyecto, mapas triangulares o cuadriculares también eran viables, y simplemente se tendría que cambiar los modelos de las casillas para dar su propio estilo artístico.

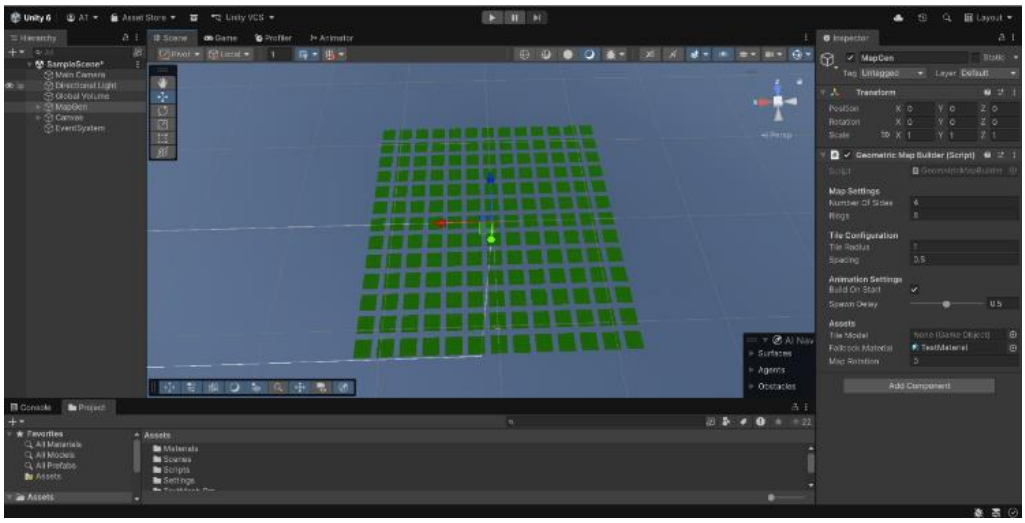
Además, el desarrollo del juego prueba ha mostrado cuanta variedad puede dar un programa de generación modular como este. Se pudo dar interacción a las casillas de una manera muy fácil, y si se deseaba cambiar, remover, o aumentar tipos de casillas, se hacía sin mayor dificultad.

Figura 46
Prueba de generación de mapa mediante triángulos.



***Nota*.** Muestra de los resultados de una generación de mapa usando triángulos.

Figura 47
Prueba de generación de mapa mediante cuadrados.



***Nota*.** Muestra de una prueba de generación de mapa usando cuadrados como las casillas.

4.15. Procedimiento para ampliar el sistema

Con la confirmación de la validez del sistema para un juego TBS, los próximos pasos involucrarían optimizar el almacenamiento de las casillas para interacciones que requieran la destrucción de aquellas. Con estos retoques, se podría probar el sistema para otros géneros

de juegos que requieren un tipo de estructura similar en algún punto de su jugabilidad. Finalmente, se planea lanzar este algoritmo al Unity Store, como un *asset* que puede ser utilizado por más personas, y así aumentar su uso, y expandir las pruebas para realizar retoques en donde los desarrolladores digan que sea necesario.

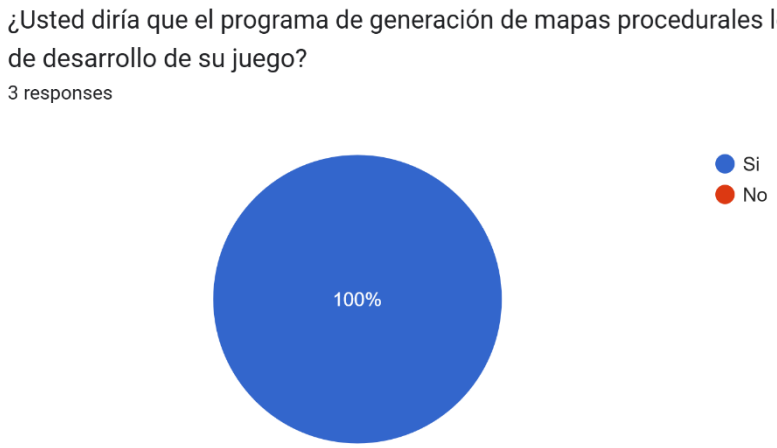
4.16. Validación

Para validar la funcionalidad del algoritmo de generación dentro del juego prueba, se presentó del juego ante un total de 12 jóvenes dentro del rango de edad del público objetivo. Se les pidió que prueben el juego y que intenten pasar los tres niveles; no se les dio indicaciones de como jugar más allá del tutorial presentado por el mismo juego al empezarlo. Se revisó la construcción del mapa y la creación de los niveles de manera aleatoria a través del algoritmo, además de la facilidad de mover por el mapa por parte del joven probando el juego.

4.17. Resultados

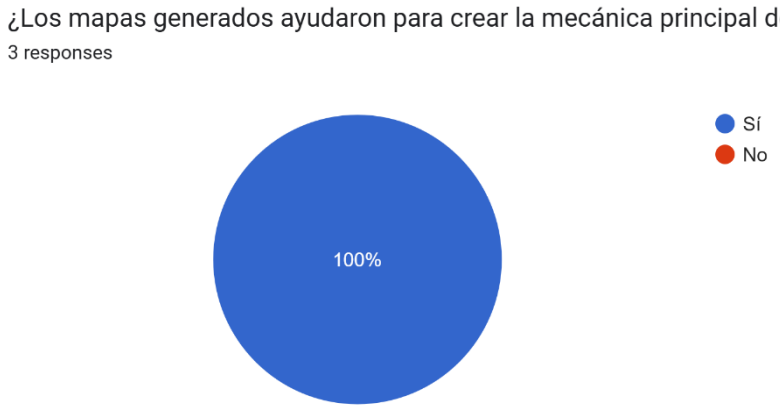
Para validar la funcionalidad del algoritmo para el desarrollo del juego, se realizó una encuesta a los diseñadores del juego Qhapac Ñan, acerca de si es que el programa de generación de mapas les ayudó en el proceso de desarrollo del juego. Sin incluir a mi persona, eran tres quienes desarrollaron el juego. Las preguntas tuvieron el objetivo de consultar su opinión si es que el programa de generación de mapas sirvió para la simplificación del desarrollo del juego; para mejorar calcular estos resultados, las preguntas fueron de estilo cuantitativo:

Figura 48
Pregunta 1 de la encuesta.



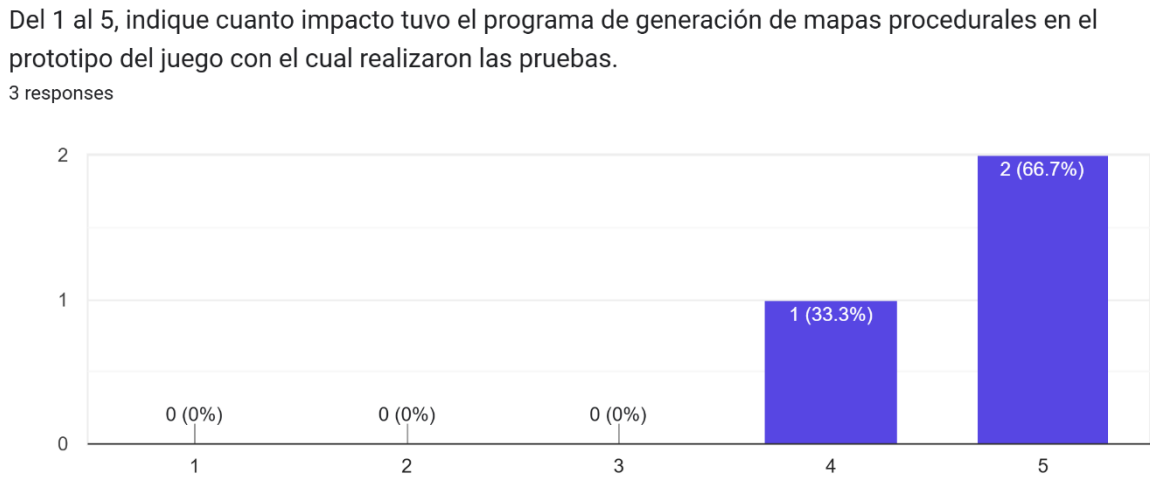
Nota. Respuestas de la primera pregunta de la encuesta, que muestran que el programa ayudó a agilizar el proceso de desarrollo del juego.

Figura 49
Pregunta 2 de la encuesta.



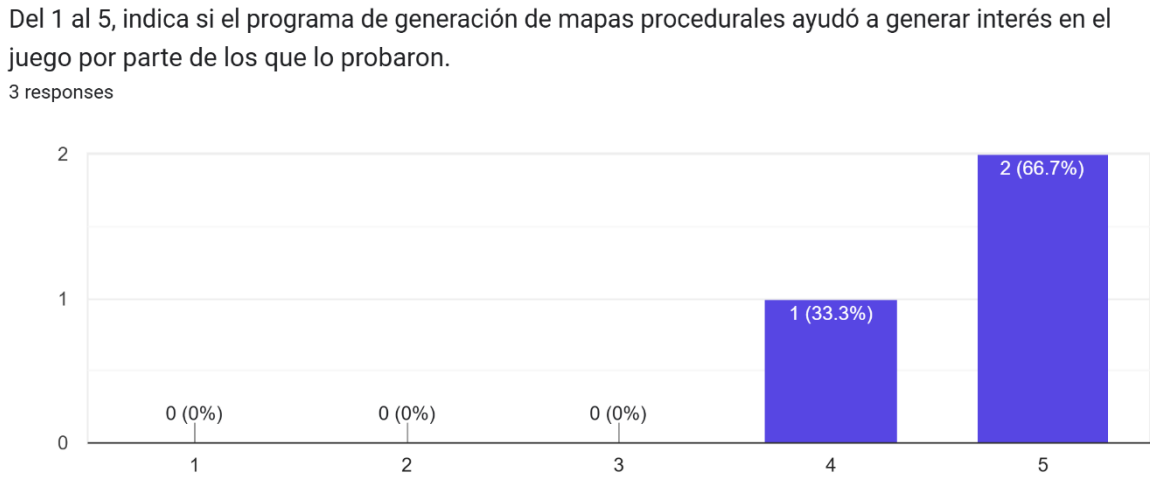
Nota. Respuestas de la segunda pregunta de la encuesta, que confirman que el mapa generado sirvió para crear la mecánica central del juego.

Figura 50
Pregunta 3 de la encuesta.



Nota. Respuestas obtenidas de la tercera pregunta de la encuesta, que muestra que el programa tuvo un impacto significativo en el prototipo del juego.

Figura 51
Pregunta 4 de la encuesta.



Nota. Respuestas de la cuarta pregunta de la encuesta, que refleja un interés por parte de los jugadores en el bucle del juego, en donde la generación del mapa fue una parte clave.

Con las respuestas de la encuesta, además del período de pruebas, se pudo demostrar la eficiencia del algoritmo en generar mapas y construir los niveles para los jóvenes. Los desarrolladores indicaron que cada usuario tenía un nivel distinto en la composición de las casillas y sus tipos, y que la mayoría de los jugadores lograron entender de manera rápida como moverse por las casillas del mapa. Cuando entendieron el concepto, se les resultó muy fácil avanzar por el mapa y utilizar los poderes otorgados en el juego para modificar el mapa a su ventaja. Los jóvenes tuvieron un interés alto por la experiencia, lo que demuestra la eficiencia de la jugabilidad.

Durante el desarrollo del juego, también se pudo evidenciar de manera directa como el algoritmo ayudó a agilizar el diseño de la experiencia. Como los mapas se generan mediante código, ya no te tenía que crear mapas especificas para cada nivel. El artista técnico solo tuvo que crear las casillas que conforman cada mapa, y de ahí el código se encargó de insertarlos de tal manera que elaboraron un mapa con el estilo deseado. Además, los elementos como la posición del jugador, enemigos, y las casillas especiales permitieron que el diseñador de niveles pueda fácilmente ver cuantos elementos deben tener cada nivel para una jugabilidad divertida y balanceada.

4.18. Conclusiones

Se pudo determinar que el programa de generación del mapa tiene un gran uso en la creación de juegos de estrategia en 3D. El desarrollo del juego prueba tuvo un apoyo por el algoritmo, permitiendo que se dedicara una mayor parte del período de diseño a las mecánicas principales del juego y sus detalles; y con las casillas, hubo un elemento con la cual basar lo que sería la mecánica de movimiento y otorgamiento de poderes.

Dentro de la validación, se comprobó con una prueba real la capacidad del programa de generación para apoyar a la producción de un juego TBS en 3D que tenga éxito. El interés por parte de los usuarios para seguir jugando hasta pasarse los tres niveles mostró un buen enganche y fluidez de jugabilidad. Como el mapa era una parte central del ritmo del juego y de sus mecánicas, este interés también ayudó a validar que el algoritmo servía para dar mecánicas útiles a un juego.

Estas observaciones dejan abierta la posibilidad de continuar con este proyecto, con el objetivo de revisar si su uso puede ayudar a expandir el mercado con un género olvidado por la mayoría de la industria, además de fomentar el emprendimiento de desarrolladores *indies* quienes desean publicar el producto de su imaginación, sin importar que tipo de juego sea.

Referencias

[1] A. V. Shubin and V. V. Kugurakova, “Designing a Tool for Creating Gameplay through the Systematization of Game Mechanics,” *Automatic Documentation and Mathematical Linguistics*, vol. 58, no. S5, pp. S299–S306, Dec. 2024, doi: 10.3103/s0005105525700384.

[2] Robert. Zubek, *Elements of game design*. The MIT Press, 2020.

[3] C. Fabricatore, “GAMEPLAY AND GAME MECHANICS DESIGN GAMEPLAY AND GAME MECHANICS DESIGN: A KEY TO QUALITY IN VIDEOGAMES,” 2007. doi: doi.org/10.13140/RG.2.1.1125.4167.

[4] J. H. Kim and H. Sung, “A Hexagon Sensor and A Layer-Based Conversion Method for Hexagon Clusters,” *Information (Switzerland)*, vol. 15, no. 12, Dec. 2024, doi: 10.3390/info15120747.

[5] A. Johnston, “The State of Indie Games in 2025 and Beyond, Part 1 | SUPERJUMP.” Accessed: Jul. 27, 2026. [Online]. Available: https://www.superjumpmagazine.com/the-state-of-indie-games-in-2025-and-beyond-part-1/

[6] A. Borrelli, V. Nardone, G. A. Di Lucca, G. Canfora, and M. Di Penta, “Detecting Video Game-Specific Bad Smells in Unity Projects,” in *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*, Association for Computing Machinery, Inc, Jun. 2020, pp. 198–208. doi: 10.1145/3379597.3387454.

[7] G. B. Gordo, “Trabajo Fin de Grado VIDEOJUEGO de PLATAFORMAS 2D en UNITY,” 2020.

[8] A. Sharma, A. Dalmia, M. Kazemi, A. Zouaq, and C. J. Pal, “GeoCoder: Solving Geometry Problems by Generating Modular Code through Vision-Language Models,” Oct. 2024, [Online]. Available: http://arxiv.org/abs/2410.13510

[9] Program-Ace, “How Much Do You Know About Games in the Unity 2D vs. 3D Battle? | by Program-Ace | Medium.” Accessed: Jul. 27, 2026. [Online]. Available: https://program-ace.medium.com/how-much-do-you-know-about-games-in-the-unity-2d-vs-3d-battle-42c1a4c19744

[10] SteamDB, “Steam Game Releases Summary for Indie · SteamDB.” Accessed: Jul. 27, 2026. [Online]. Available: https://steamdb.info/stats/releases/?tagid=492

[11] R. M. Smelik, K. Jan de Kraker, S. A. Groenewegen, T. Tutenel, and R. Bidarra, “A Survey of Procedural Methods for Terrain Modelling *,” 2009.

[12] R. Van Der Linden, R. Lopes, and R. Bidarra, “Procedural generation of dungeons,” 2014.

[13] L. Johnson, G. Yannakakis, and J. Togelius, *Workshop on Procedural Content Generation in Games : co-located with the 5th International Conference on the Foundations of Digital Games : June 18th, 2010, Asilomar Conference Grounds, Monterey, California, USA*. ACM, 2010.

[14] K. Sahr, “HEXAGONAL DISCRETE GLOBAL GRID SYSTEMS FOR GEOSPATIAL COMPUTING,” 2011.

[15] Y. Wang, J. Luo, A. Gaier, E. Atherton, and H. Koch, “PlotMap: Automated Layout Design for Building Game Worlds,” Aug. 2024, [Online]. Available: http://arxiv.org/abs/2309.15242

[16] A. A. Yallico, J. I. Chávez, and A. Barrientos, “Development of a real-time strategy genre video game for PC using Unity and Blender,” in *Memorias de la Conferencia Iberoamericana de Complejidad, Informatica y Cibernetica, CICIC*, International Institute of Informatics and Systemics, IIIS, 2025, pp. 167–173. doi: 10.54808/CICIC2025.01.167.

[17] C. A. Pickover, “The Math Book_ From Pythagoras to the 57th Dimension, 250 Milestones in the History of Mathematics (Sterling Milestones),” 2009.

[18] D. Fletcher and I. Parberry, “3D Math Primer for Graphics and Game Development Second Edition,” 2011.

[19] J.-N. Meckenstock, N. Hirschlein, S. Schlauderer, and S. Overhage, “Unraveling Agile Software Development Business Value: A Qualitative Systematic Review of Agile Practices and Benefits,” *Pacific Asia Journal of the Association for Information Systems*, vol. 17, no. 5, pp. 48–93, 2025, doi: 10.17705/1pais.17503.

[20] C. Politowski, F. Petrillo, J. E. Montandon, M. T. Valente, and Y.-G. Guéhéneuc, “Are Game Engines Software Frameworks? A Three-perspective Study,” Sep. 2020, doi: 10.1016/j.jss.2020.110846.

[21] K. Maher, “Game Engines Power Content Creation,” 2019.

[22] Rabin S., “Game Programming Gems 1,” 2010.

[23] J. Haas, “A History of the Unity Game Engine An Interactive Qualifying Project Submitted to the Faculty of WORCESTER POLYTECHNIC INSTITUTE in partial fulfillment of the requirements for graduation,” 2014.

[24] M. Foxman, “United We Stand: Platforms, Tools and Innovation With the Unity Game Engine,” *Social Media and Society*, vol. 5, no. 4, Oct. 2019, doi: 10.1177/2056305119880177.

[25] J. Skeet, “M A N N I N G IN DEPTH FOURTH EDITION,” 2019.

[26] Bjarne. Stroustrup, *The C++ programming language : language, library and design*. Addison-Wesley, 1997.

[27] R. Nystrom, “Game Programming Patterns,” 2011.

[28] Z. Shen, “Procedural Generation in Games: Focusing on Dungeons,” *SHS Web of Conferences*, vol. 144, p. 02005, 2022, doi: 10.1051/shsconf/202214402005.

[29] F. Shariatpour, A. Shakibamanesh, and M. Rahbar, “Procedural generation as an approach for digital representation of a city,” *Computational Urban Science*, vol. 6, no. 1, Dec. 2026, doi: 10.1007/s43762-026-00263-8.

[30] Ben. Garney and Eric. Preisz, *Video game optimization*. Course Technology Cengage Learning, 2011.

[31] D. Chądzyńska and D. Gotlib, “Maps in video games – range of applications,” *Polish Cartographical Review*, vol. 47, no. 3, pp. 137–145, Sep. 2015, doi: 10.1515/pcr-2015-0011.

[32] R. Kaźmierczak, R. Skowroński, C. Kowalczyk, and G. Grunwald, “Creating Interactive Scenes in 3D Educational Games: Using Narrative and Technology to Explore History and Culture,” *Applied Sciences (Switzerland)*, vol. 14, no. 11, Jun. 2024, doi: 10.3390/app14114795.

[33] C. H. Lai and P. Y. Hu, “The Gaming Revolution in History Education: The Practice and Challenges of Integrating Game-Based Learning into Formal Education,” Jun. 01, 2025, *Multidisciplinary Digital Publishing Institute (MDPI)*. doi: 10.3390/info16060490.