Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Generación y Visualización de Laberintos 3D Concéntricos en Unity

Generation and Visualization of Concentric 3D Mazes in Unity

Dario Cerviño Luridiana

La Laguna, 11 de 07 de 2023

D. **Jesús Miguel Torres Jorge**, con N.I.F. 43.826.207-Y profesor Contratado Doctor de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A (N)

Que la presente memoria titulada:

“Generación y Visualización de Laberintos 3D Concéntricos en Unity”

ha sido realizada bajo su dirección por D. **Dario Cerviño Luridiana**,

con N.I.F. 77.143.370-J.

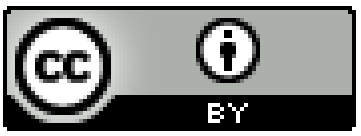
Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 11 de julio de 2023

Agradecimientos

A mi familia, especialmente a mi hermana y mi madre, por su ayuda y apoyo constante.

A mis compañeros, por algo. Ciertamente por algo.

Licencia



Resumen

El objetivo de este trabajo es el diseño y desarrollo de un juego de laberintos tridimensionales por niveles concéntricos.

Se hace uso del motor de videojuegos Unity para generar laberintos volumétricos en formas convexas simples, así como la esfera. Los laberintos se organizan por capas, con el objetivo de llegar al centro. Los controles de movimiento responden a cambios de perspectiva, la cual también está bajo el control del jugador.

La generación de laberintos empieza con la creación de un grafo volumétrico en la forma deseada. El orden de los nodos del grafo permite controlar la generación de un laberinto mediante un algoritmo de búsqueda en profundidad modificado, el cual limita el número de conexiones entre capas.

Utilizando tanto el grafo original como el subgrafo que forma el laberinto, se genera un modelo 3D procedural para cada capa, según lo demande la exploración del jugador.

Se han logrado generar seis tipos de laberintos concéntricos con formas de tetraedro, cubo, octaedro, dodecaedro, icosaedro y esfera. Todos los tipos de laberintos son navegables y usan modelos que permiten distinguir claramente los caminos.

Aunque los sistemas principales del proyecto están completos, aún falta el desarrollo de interfaces y lógica de juego para constituir un juego completo.

Palabras clave: Laberintos, generación procedural, 3D, grafos, Unity 3D, diseño de videojuegos.

Abstract

The objective of this work is the design and development of a game of three-dimensional mazes organized in concentric layers.

The Unity game engine is used to generate volumetric mazes contained in simple convex shapes and the sphere. Each maze is organized by layers with the goal of reaching the center. The movement controls are sensible to perspective changes, which are also controlled by the player.

The maze generation begins with the creation of a volumetric graph in the desired shape. The nodes are ordered so that the maze generation can limit the number of connections between layers. A modified depth-first search algorithm is used for the generation of the maze from the original graph.

Using both the original graph and the subgraph that forms the maze, a procedural 3D mesh is generated for each layer, based on the player's exploration demands.

Six types of concentric mazes have been successfully generated, including the tetrahedron, cube, octahedron, dodecahedron, icosahedron and sphere shapes. All maze types are navigable and able to generate models that clearly distinguish the pathways.

While the main systems of the project are complete, the development of interfaces and game logic is still pending for it to constitute a complete game.

Keywords: Maze, procedural generation, 3D, graphs, Unity3D, game design.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1 Introducción

El proyecto consiste en el desarrollo de un juego de puzles en Unity basado en la generación de laberintos circunnavegables concéntricos. El proyecto abarca desde la generación de estos laberintos hasta la creación de lógica y niveles del juego. Es una idea de propuesta propia, lo que implica más libertad y más responsabilidades a la hora de escoger objetivos.

Un aspecto a destacar es que el proyecto tiene motivación artística más que técnica o de investigación. Esto se manifiesta principalmente en la abundancia de métodos propios y escasez de recursos externos importantes.

Cada nivel se basa en alcanzar el centro de una “esfera” (que puede ser una esfera u otra forma tridimensional convexa, como un cubo). Cada esfera se divide a su vez en capas, con caminos que conectan capas contiguas.

En este sentido, cada esfera podría considerarse un laberinto tridimensional. La idea original es que cada capa deje de ser relevante una vez superada, progresando hacia el centro de la esfera, pero se ha implementado también la opción menos restringida.

Está implementado en el motor de videojuegos Unity, que ofrece herramientas sencillas para la visualización limpia del laberinto. A pesar de ello, la mayor parte de la lógica está desconectada de los sistemas de Unity, siendo implementada directamente en C#.

1.1 Antecedentes

Para la generación de laberintos existe una gran cantidad de algoritmos [1], desde divisiones recursivas hasta algoritmos basados en grafos. La implementación representa los laberintos como grafos, por lo que estos últimos son los que tienen más relación. La mayoría de estos algoritmos son familiares, ya que son variantes de algoritmos de búsqueda y generación de árboles mínimos (DFS, Kruskal, Prim).

Por otro lado, el mapeado de nodos en una esfera tiene varias soluciones. La más básica es la proyección de coordenadas planas, formando un sistema de latitud y longitud. Su inconveniente es la concentración de puntos en los polos. Se pueden proyectar diferentes formas convexas (como un cubo o un icosaedro) para reducir ese problema. La solución más escalable es la esfera de Fibonacci, una proyección de una espiral de Fibonacci a la superficie de la esfera que consigue una distribución muy uniforme.

En el proyecto se hace uso de la esfera de Fibonacci.

No se han encontrado proyectos que hagan específicamente esto, sí existen generadores de laberintos en la superficie de una esfera (aunque fuera por proyección plana). Implementaciones de laberintos tridimensionales, incluso por capas como esta, son raras. El término se suele referir a laberintos planos explorables en un entorno virtual 3D o laberintos superficiales, en lugar de laberintos verdaderamente volumétricos. El proyecto más similar es un juego web [2] que utiliza laberintos pregenerados en la superficie de varios objetos aproximadamente esféricos (por ejemplo, una manzana).

1.2 Objetivos

Para llegar a nuestro objetivo general, se proponen los siguientes objetivos parciales:

- Generación de grafos cúbicos
- Generación de grafos con tetraedros, octaedros e icosaedros
- Generación de grafos con dodecaedros
- Generación de grafos esféricos usando la esfera de Fibonacci
- Extrapolación de los grafos generados a grafos concéntricos
- Generación de laberintos a partir de grafos
- Navegación por laberintos
- Visualización de laberintos (generación de modelo 3D)
- Diseño estético y de audio
- Lógica de juego
- Menú y opciones

Estos objetivos se completaron todos menos los tres últimos, que están apenas empezados.

1.2.1 Desarrollo

Cronológicamente, se completaron los siguientes hitos:

- **Laberinto cúbico**: Se planificó y programó la generación del grafo y el laberinto cúbico superficial. Aún no se generó el modelo 3D.
    - *(en paralelo)* **Visualizador simple**: Un visualizador de depuración de los laberintos. Permite visualizar los nodos, el grafo y el laberinto. Se fue mejorando a medida que avanzó el proyecto.
- **Navegador y cámara**: Se planeó e implementó el esquema de control de la cámara y el jugador. La implementación sufrió pocos cambios, al ser bastante generalista.
- **Modelo 3D de laberinto**: Se generó el primer modelo de un laberinto cúbico. En la refactorización posterior, se alteraría el algoritmo para que pudiera aplicarse al resto de laberintos.
- **Laberinto concéntrico**: Se hicieron varias implementaciones de la profundidad del laberinto, desde la acumulación de varios laberintos superficiales hasta la generación de todos los laberintos en un solo grafo (versión final).
- **Sólidos platónicos**: Se desarrollaron los laberintos con caras triangulares y posteriormente también el dodecaedro.
    - **Refactorización**: Se reorganizó el código en una estructura de clases más sólida. Se eliminó código inútil del laberinto cúbico y el sistema de navegación.
    - **Generalización de generación de modelos 3D**.
- **Laberinto esférico**: Se implementó la generación de grafos esféricos, lo que permitió por extensión la generación de laberintos. La distribución de nodos de acuerdo a la esfera de Fibonacci se hizo mucho antes, pero la triangulación entre ellos resultó ser más desafiante.

Capítulo 2 Diseño

Uno de los objetivos principales es enfocar el proyecto desde el ámbito de diseño de videojuegos, intentando predecir y cubrir necesidades del usuario para que la experiencia sea más natural y cómoda.

2.1 Diseño de niveles

La primera parte de la experiencia de juego viene del diseño del entorno: cómo comunicar las acciones disponibles y transmitir la sensación de progreso.

Las acciones disponibles en un laberinto se limitan al movimiento por los caminos disponibles, que en el caso de laberintos 2D, se suelen ver desde arriba (con “vista de pájaro”). Por otro lado, tener una sensación de progreso clara y fiable está en contra del desafío que debe suponer un laberinto. En ese sentido, el progreso se puede visualizar más como una heurística, según la distancia real al objetivo, pese a la posibilidad de ir por el camino equivocado. La visibilidad del objetivo es un caso más interesante en el caso de laberintos tridimensionales.

En el caso del proyecto, se separa el laberinto volumétrico en capas concéntricas, un enfoque que se aproxima un poco a la completa visibilidad que tienen los laberintos planos, pero sin renunciar al componente tridimensional. Como extensión natural de esto, se puede usar la profundidad de la capa como medida de progresión sin invalidar completamente el laberinto.

En ese caso, cada laberinto sería una colección de laberintos superficiales cada vez más pequeños con el centro como objetivo final. Cada vez que se supera una capa, no hace falta volver a visitarla. Alternativamente, se puede hacer un parámetro de dificultad que afecte la fiabilidad de ese medidor de progreso. Por ejemplo, con la necesidad de volver a capas superiores para avanzar o con la posibilidad de descensos que acaben en caminos sin salida.

El parámetro de dificultad no se ha desarrollado más allá de la opción más fácil y la más difícil.

2.2 Diseño de juego

La otra cara de la moneda es la libertad y herramientas que tiene el jugador. Movimiento, acciones y controles intuitivos.

El movimiento es necesariamente tridimensional, pero se puede separar en movimiento por la superficie del objeto y vertical (que además refuerza la diferencia de importancia entre ambos).

2.2.1 Cámara

Al navegar un espacio tridimensional y no tener todo el laberinto visible a la vez, el control de la perspectiva es muy importante. Se usa una perspectiva flexible que el jugador puede rotar, desde la cual se interpreta el *input* del jugador.

Para remarcar de nuevo la diferencia entre movimientos superficiales y verticales, el punto de vista está orientado directamente contra la superficie por la que pueda moverse, en lugar de orientado hacia el centro del laberinto. Los laberintos van a ser siempre figuras convexas, por lo que no hay obstrucciones a la visión del jugador o el laberinto.

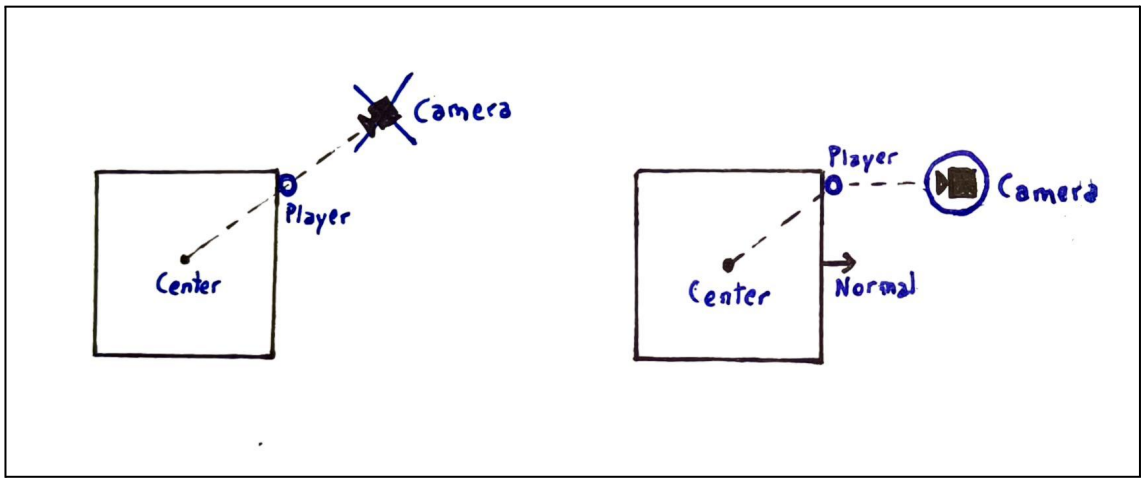


Figura 1: Posición de la cámara
La cámara se orienta según el vector normal del plano sobre el que se mueve el jugador.

Para permitir la mayor flexibilidad posible, el jugador tiene control sobre la rotación, zoom e inclinación de la cámara.

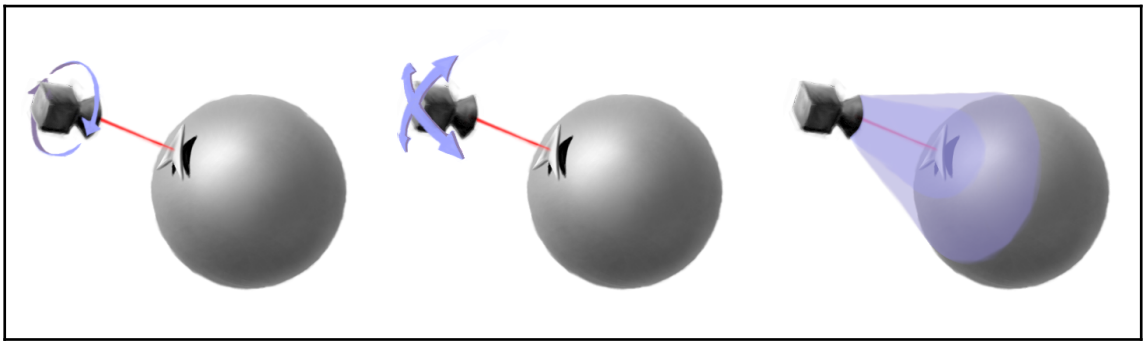


Figura 2: Controles de la cámara (rotación, inclinación y zoom)

2.3 Diseño de controles

Se utiliza un mando de XBox, aunque los sistemas que se utilizan están presentes en la mayoría de mandos modernos. Hacen falta tres ejes de movimiento y 4 de control de cámara (rotación, inclinación en 2D y zoom).

Se utiliza el joystick izquierdo para el control de movimiento superficial y los gatillos superiores como eje vertical (siendo el gatillo izquierdo el positivo).

La inclinación tiene dos ejes de libertad, por lo que encaja perfectamente en el joystick restante. La rotación de la cámara sobre el eje de visión se controla con los gatillos inferiores y el zoom con el eje vertical del D-Pad.

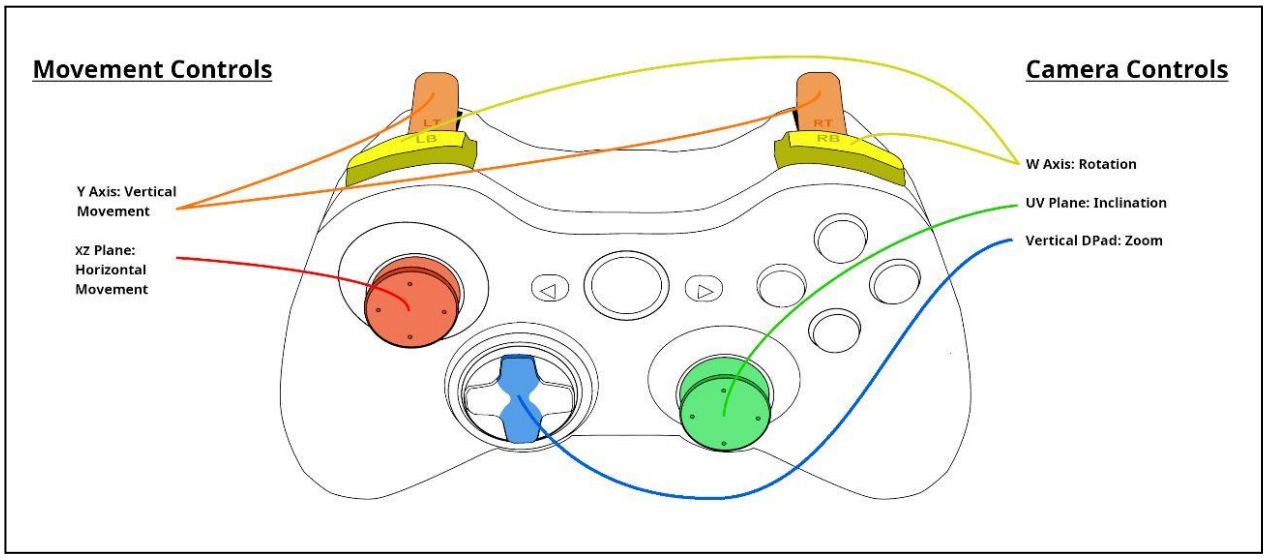


Figura 3: Esquema de controles
Imagen editada a partir de [3].

2.3.1 Otras opciones de control

Una opción que siempre viene bien incluir es la opción de modificar los controles para que le resulten más cómodos al jugador. Aunque la idea no se ha implementado, hay una capa de abstracción que facilitaría su posterior inclusión.

Actualmente, la clase responsable de los controles tiene alternativas de teclado para los ejes, pero sería interesante añadir controles con giroscopio de un móvil o un mando.

Capítulo 3 Aspectos técnicos

3.1 Estructura de clases

Las clases siguen un diseño modular, compacto y flexible, con el principal objetivo de permitir expansiones fácilmente.

3.1.1 Laberintos

Los laberintos forman un árbol de clases que dependen de las características que tienen en común. La base es una clase abstracta con las propiedades comunes a todos los laberintos, que también actúa como interfaz de los laberintos para el acceso de otras clases.

La primera separación diferencia los laberintos basados en sólidos platónicos de los esféricos, ya que los primeros se organizan por caras. Dentro de los sólidos, se diferencian por la forma de la cara (triángulo, cuadrado y pentágono), que tienen indexaciones diferentes. Finalmente, las formas con caras triangulares forman su propio subárbol con sus tres casos (tetraedro, octaedro, icosaedro).

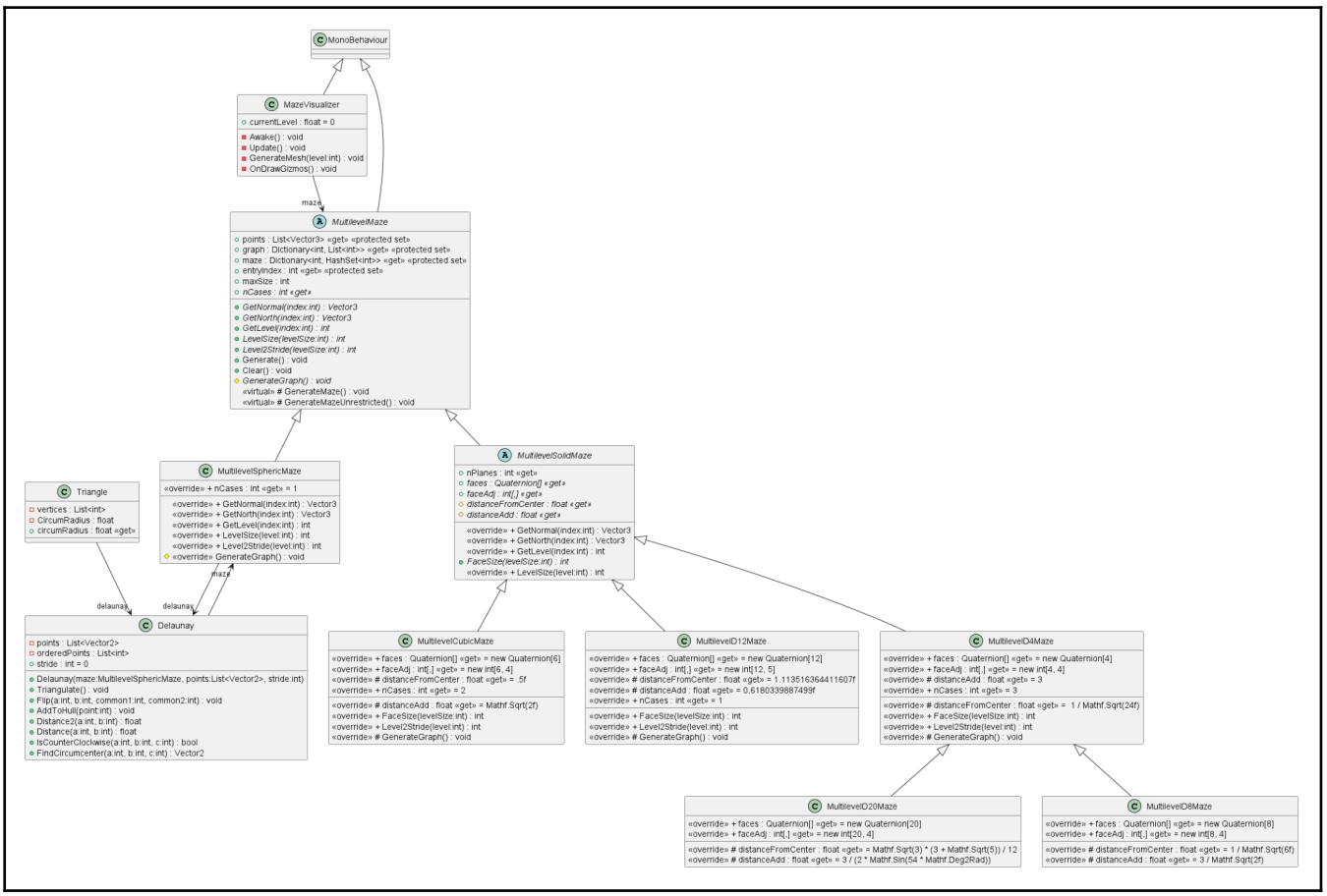


Figura 4: Árbol de clases de los laberintos

3.1.2 Navegación

La navegación por el laberinto está dividida en varias clases independientes. La primera (*MazeNavigator*) proporciona una interfaz sencilla para el desplazamiento restringido al laberinto. La segunda (*NavigatorController*) conecta esta interfaz con los controles. La separación entre navegador y controles permitiría implementar otros tipos de navegadores (automáticos, basados en físicas, etc.).

3.1.3 Observador y Controles

El controlador por *input* depende de una clase que haga de punto de referencia, que además también implementa los controles de la cámara. *ObserverController* se encarga de posicionar el punto de referencia y la cámara y de recibir los controles de cámara.

Ambos además necesitan una referencia al laberinto y hacen uso de una clase estática para tener control sencillo de las opciones de *input* (*InputMapper*).

3.2 Generación de laberintos

3.2.1 Propiedades

Las propiedades básicas que debe tener un laberinto son un conjunto de nodos y las conexiones entre ellos. Para un juego, los nodos tendrán además una posición y las conexiones posibles estarán restringidas por esa posición.

En este caso, los nodos guardan su posición en una lista de posiciones. Los nodos en sí mismos están más representados por el índice dentro de esa lista. La mayoría de métodos aprovechan y dependen del orden de los nodos.

Para la matriz de adyacencia (posibles conexiones), se utiliza un diccionario o mapa de listas. Cada índice corresponde a un nodo y su entrada es la lista ordenada de adyacencias que posee. Las últimas dos entradas en la lista representan adyacencias verticales, es decir, las que cambian de nivel del laberinto. Para el resto no tiene relevancia por cuál se empiece siempre que estén ordenadas en sentido horario mirando desde el exterior.

Las conexiones que forman el laberinto son un diccionario de conjuntos (HashSet), ya que generalmente se usará como filtro de la matriz de adyacencia que tiene una forma más regular.

Además de esto, los laberintos tienen métodos útiles para cálculos con índices (tamaño de capa, profundidad de índice, etc.) y métodos para obtener el vector normal y el vector norte (que es un vector tangente sin más requisitos) de cada nodo.

3.2.2 Laberintos sólidos

Los laberintos con forma de sólidos platónicos tienen otros atributos en común que no están presentes en la esfera. Sus caras están representadas por una lista de rotaciones (cuaterniones de Unity) y una matriz de adyacencia que describe las conexiones que tienen entre bordes.

Además, también tienen valores de interés para la generación, como la distancia que tiene cada cara al centro en proporción al lado.

Todas estas variables dependen del sólido que se represente, por lo que son constantes dentro de cada clase.

3.2.3 Generación de laberintos cúbicos

Los laberintos cúbicos tienen caras cuadradas, que son las más sencillas de organizar. Como se ve en el diagrama (Figura 5), las caras cuadradas se rellenan de oeste a este, de norte a sur.

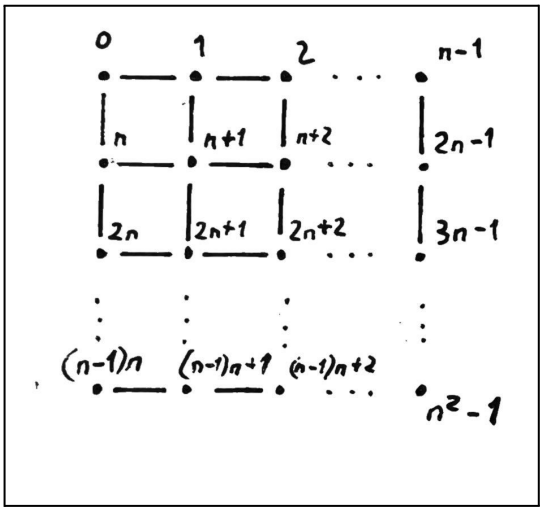


Figura 5: Cara cuadrada
Índices y orden de nodos de una cara cuadrada con *n* nodos por lado.

Debido a que las caras están organizadas para facilitar las conexiones entre caras, usar oeste-este en lugar de izquierda-derecha es más apropiado. Las adyacencias entre caras contemplan la orientación de las caras (una cara puede conectar su lado norte con el lado oeste de otra).

Todas las caras se rellenan a la vez, por lo que cada cara contiene un índice de cada seis distribuidos uniformemente. Esto es la regla para todos los sólidos, ya que permite localizar la cara a la que pertenece cada nodo con una operación al índice.

La generación del grafo base se hace un bucle de cuatro niveles (cubo o nivel, fila, columna y cara). Compactado en el segundo nivel, también se exploran simultáneamente las vecindades entre caras en todas las direcciones. El cubo está organizado de tal forma que los índices que se deben conectar entre caras se exploran en la misma dirección (Figuras 6 y 7), pudiendo usar una sola plantilla de índice para cada dirección.

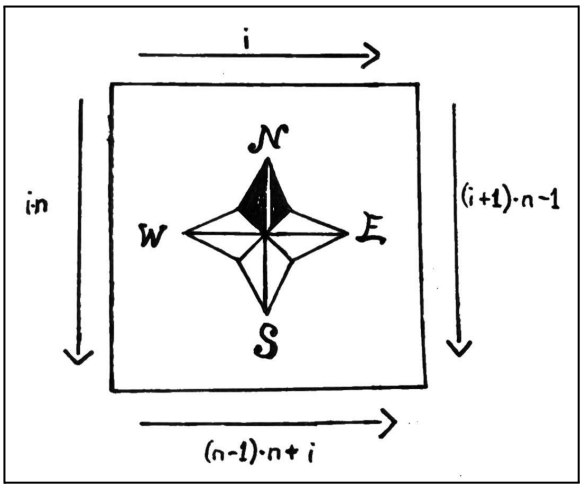


Figura 6: Bordes del cuadrado
Tamaño de lado en nodos *n* e iterador *i*.

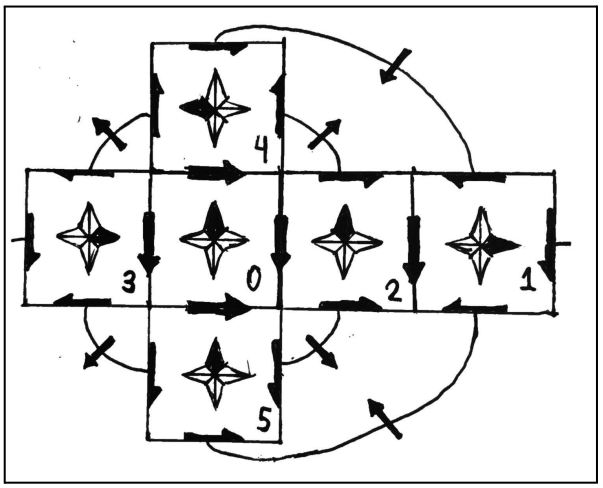


Figura 7: Organización de caras del cubo

La conexión entre niveles utiliza como regla general que la cara del nivel inferior está contenida y rodeada por nuevos nodos en el nivel superior. En el caso del cubo, esto implica que cada nivel tendrá dos nodos más por lado que su inferior, lo que a su vez separa los laberintos cúbicos concéntricos en dos casos según si los niveles tienen un número par o impar de nodos por lado.

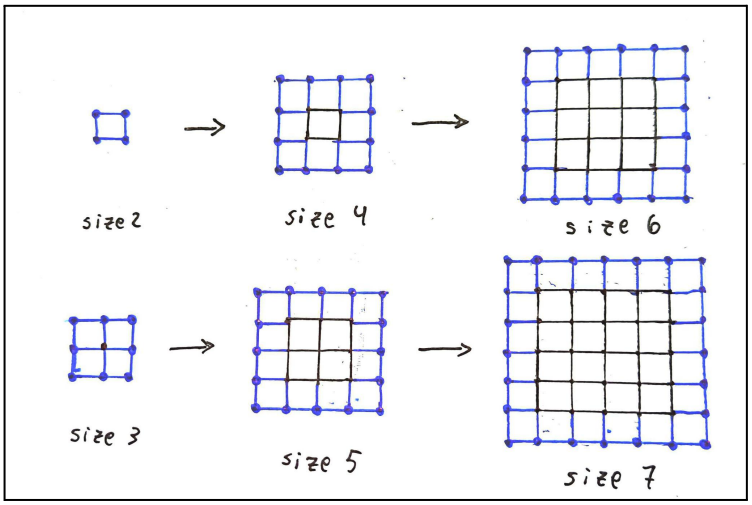


Figura 8: Progresión del cuadrado
Las caras cuadradas separan laberintos con niveles de lado par de los de lado impar (en azul nodos sin conexión con la cara anterior).

Cada nodo tiene como máximo 6 vecinos: 4 en el mismo cubo y dos en los cubos superior e inferior. En versiones anteriores se aprovechaba que el número de vecinos superficiales era constante para organizar también los vértices de la malla, pero como el resto de laberintos tienen un número variable de vecinos superficiales por nodo, se debió cambiar a un método más general.

3.2.4 Generación de laberintos con caras triangulares

Los sólidos con caras triangulares son el tetraedro (4 caras), octaedro (8 caras) e icosaedro (20 caras). Utilizan un método generalizado, por lo que forman un árbol de clases con el tetraedro como padre y los otros dos sobreescribiendo solo las variables constantes que definen las caras y su organización.

Aunque es más complicado que el cuadrado, el triángulo se puede subdividir en filas y columnas distribuidas uniformemente con relativa facilidad. Cada fila empieza más lejos que la anterior y tiene un elemento más. Como los triángulos equiláteros pueden rellenar una superficie plana con exactitud, esto resulta en una subdivisión satisfactoria y uniforme de cada cara.

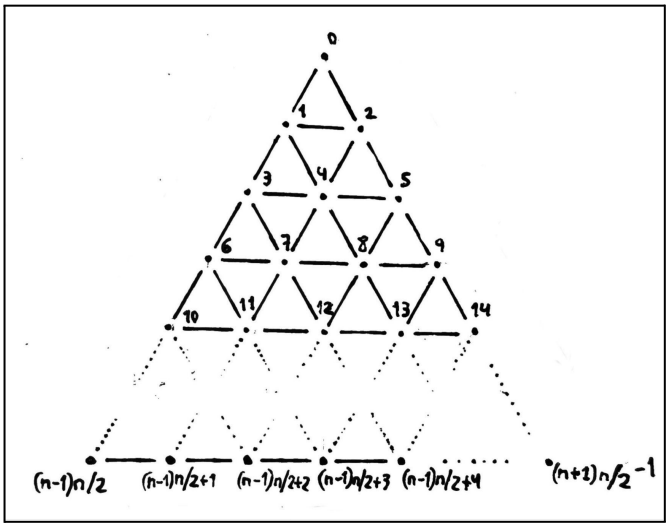


Figura 9: Cara Triangular
Índices de cara triangular de *n* nodos por lado.

La complicación viene con que cada cara tiene tres vecinos en lugar de cuatro. No se puede seguir sin modificaciones el método del cubo para conectar caras. Si dos lados se exploran en un sentido y otro en el opuesto, habrá más lados que se recorren en una de las dos direcciones, lo que causaría conexiones cruzadas.

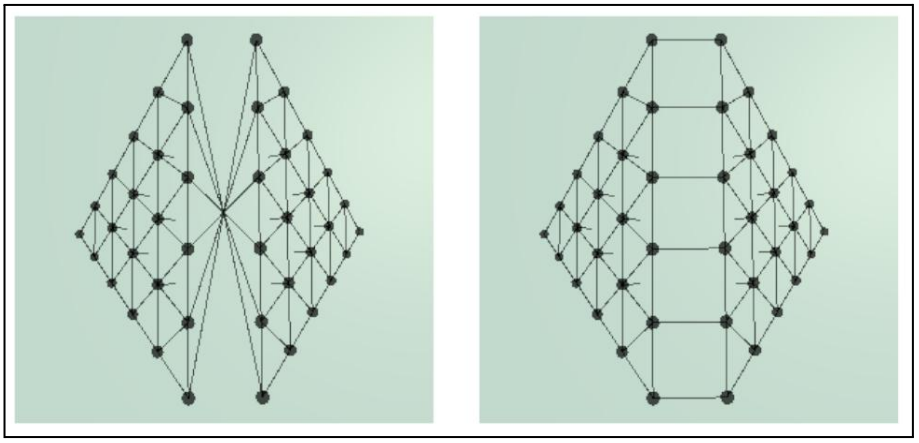


Figura 10: Conexiones cruzadas
Conexiones cruzadas (izquierda) y correctas (derecha) en el borde entre dos caras.

Para solucionar este problema, se llegó al compromiso de que dos lados de cada cara se recorrerían en direcciones opuestas, mientras que el restante cambiaría de dirección dependiendo de si la cara es par o impar dentro del orden de las caras. El resultado de esto es que la generación contempla cuatro patrones o plantillas de vecinos para cada cara y dos de ellos los aplica dependiendo de si la cara es par o impar.

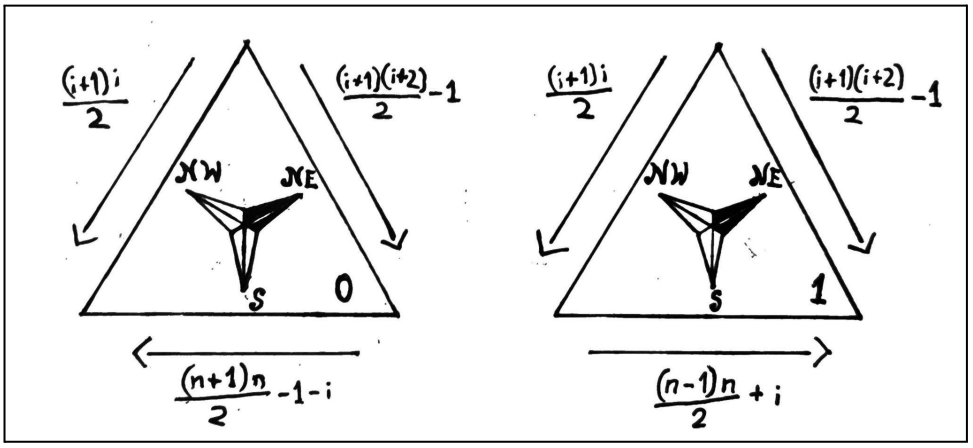


Figura 11: Bordes del triángulo
Según la cara sea par e impar se utiliza un esquema u otro (con *n* nodos por lado e iterador *i*).

Otra dificultad con respecto al cubo es que los nodos no tienen un número constante de vecinos superficiales. Si un nodo está en medio de una cara, tiene 6 vecinos, mientras que si está en un borde tiene 5 y en una esquina solo 4.

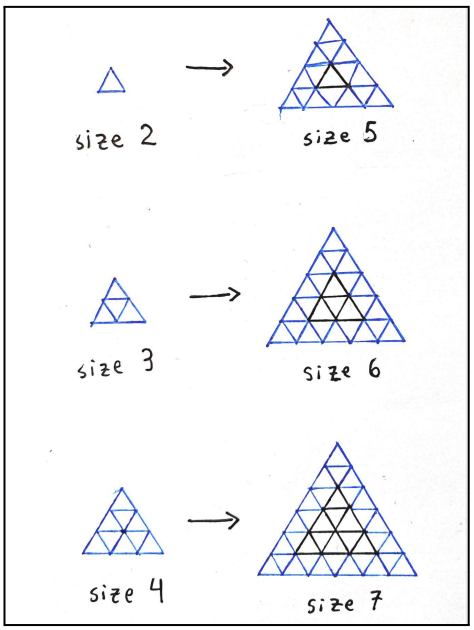


Figura 12: Progresión del triángulo
Las caras triangulares se agrupan en tres casos dependiendo del número de nodos por lado.

3.2.5 Generación de laberintos en forma de dodecaedro

El dodecaedro es la figura más problemática de las propuestas. Tanto los cuadrados como los triángulos tienen subdivisiones elegantes y exactas, pero el pentágono no.

Se optó por triangular pentágonos concéntricos, ya que esto además facilitaba el cambio de niveles. Como consecuencia de esto, los nodos ya no se organizan por filas y columnas, sino por distancia al centro (o más bien radio externo del pentágono del que forman parte) y en sentido horario, resultando más o menos en una generación en espiral.

La generación de dodecaedros es más complicada, siendo 5 bucles anidados (dodecaedro o nivel, pentágono, lado y nodos por lado) con excepciones, Tanto el primer nodo de cada cara como el primer nodo de cada lado (la esquina) se generan antes de su bucle, ya que tienen un comportamiento diferente (el “pentágono” número 0 tiene un solo vértice, en lugar de 0 o un múltiplo de 5, por ejemplo).

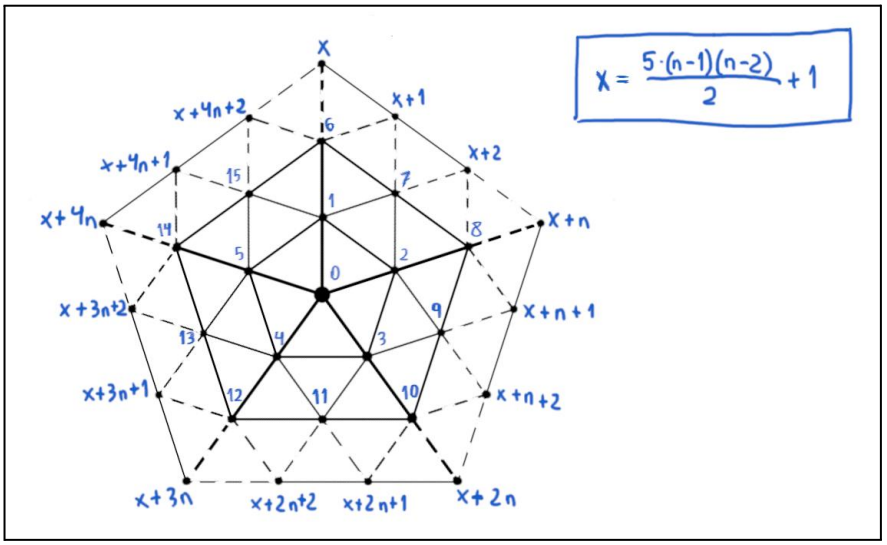


Figura 13: Cara pentagonal
Las caras pentagonales ordenan sus índices en espiral, lo que permite que las adyacencias entre caras sean fáciles de conectar durante el bucle principal.

Una facilidad que ofrece esta generación es que los extremos del pentágono están formados por los últimos nodos de este y están siempre en sentido horario. Para conectar las caras, se puede hacer directamente en los bucles principales, ya que todas las conexiones siguen la misma plantilla. Más que eso, esto también simplifica la organización de las caras, que, mientras la tabla de adyacencia esté correcta, no depende de que sus caras tengan orientaciones específicas.

Con todo, el pentágono no distribuye uniformemente sus nodos, además de que para el cálculo de las coordenadas de cada lado, solo se han obtenido resultados inexactos (las imperfecciones son casi imperceptibles, pero existentes).

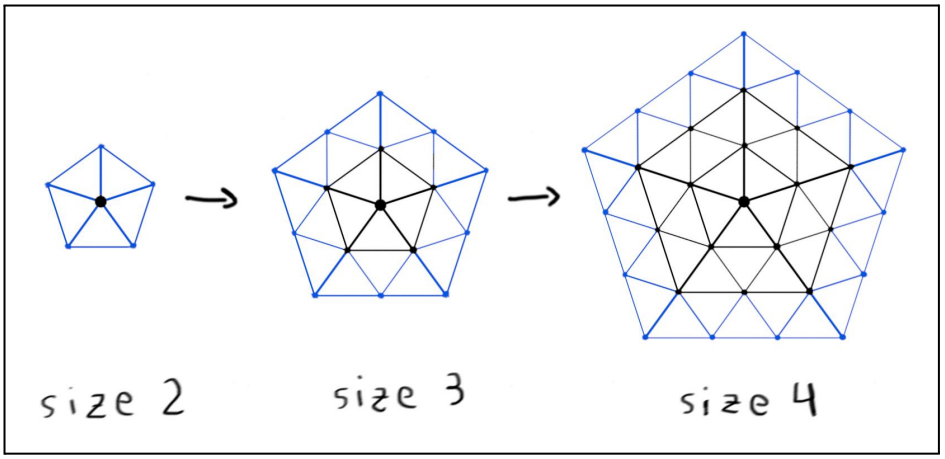


Figura 14: Progresión del pentágono
Las caras pentagonales no se agrupan en casos. Un laberinto en forma de dodecaedro recorre todas los tamaños hasta su máximo.

3.2.6 Laberintos esféricos

El laberinto esférico es a la vez más complicado y más simple. Por un lado, hace falta una manera de generar nodos equidistantes en la superficie de una esfera, además de que esta generación sea escalable a diferentes radios. Para esto se usa la esfera de Fibonacci [4], que es una proyección de la espiral de Fibonacci en espacio esférico. La implementación es muy sencilla, aunque carece del orden estricto de los nodos que se ha dado en los anteriores casos.

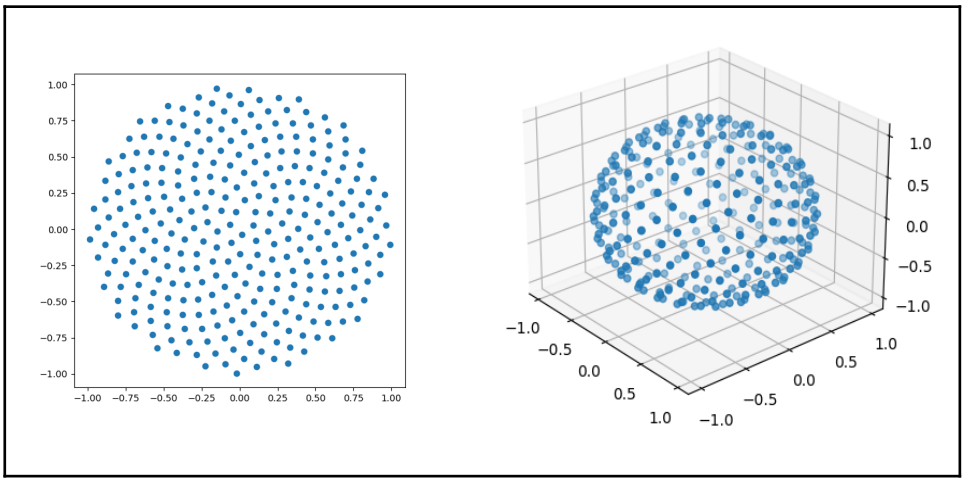


Figura 15: Espiral y esfera de Fibonacci
Ejecución con 300 puntos, usando un programa simple en python [5].

Lo que supone un desafío más complicado es la generación de las conexiones. La esfera de Fibonacci solo genera los puntos, no incluye conexiones entre ellos. Además, las conexiones entre nodos vecinos cambian conforme aumenta el número de nodos (que depende del radio de la esfera).

Siguiendo el ejemplo de varios otros proyectos [6][7], para las conexiones superficiales se implementa la triangulación de Delaunay, que consiste en una triangulación plana donde el

círculo circunscrito de cada triángulo no engloba otro punto del grafo. La triangulación de Delaunay resulta en una triangulación con triángulos compactos que, en combinación con puntos bien distribuidos, genera una malla satisfactoria.

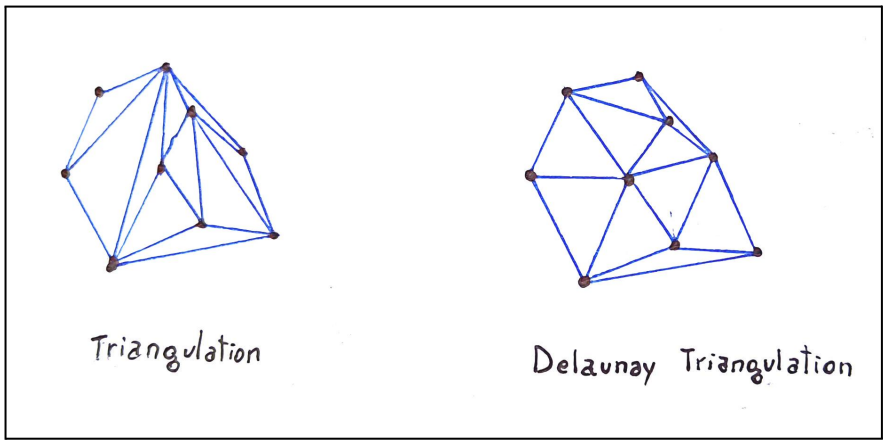


Figura 16: Triangulación de Delaunay
Comparación entre una triangulación válida cualquiera con una triangulación de Delaunay.

Con el método escogido, se usaría la proyección en el plano de los puntos generados por la esfera (proyección estereográfica), ya que el algoritmo está orientado a puntos restringidos al plano. La proyección causa que los puntos del hemisferio sur se proyecten dentro de una circunferencia centrada en el origen, y los del hemisferio norte al exterior de esa circunferencia. Triangular esa proyección debería generar una triangulación válida para el grafo tridimensional. El inconveniente de esta proyección es que el punto del polo norte se proyecta al infinito, lo que limita o complica los algoritmos de triangulación que se pueden usar.

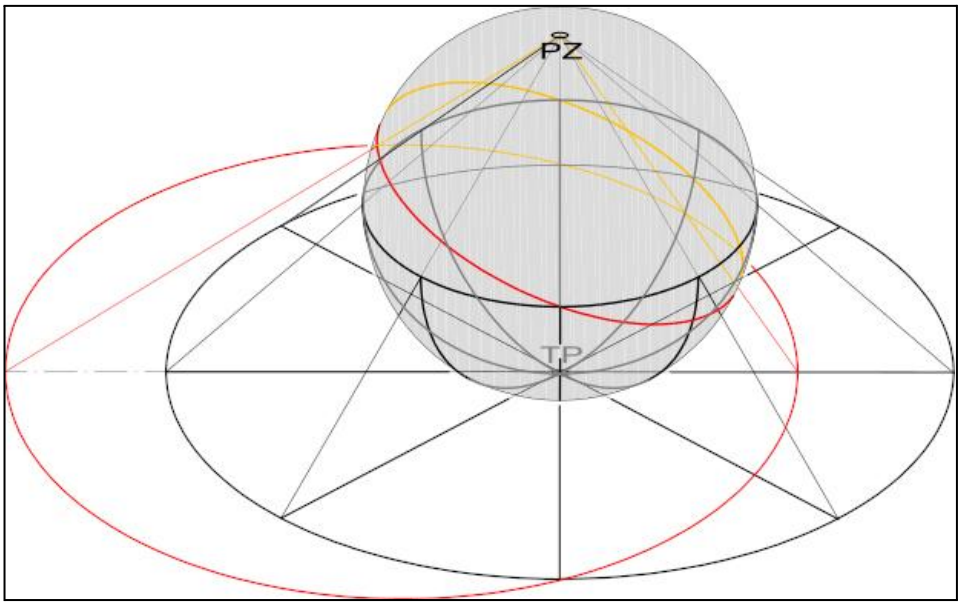


Figura 17: Proyección estereográfica [8]

El algoritmo implementado (*s-hull* o *sweep-hull*), según el artículo que lo propone [9], genera un conjunto de triángulos no superpuestos y luego usa un algoritmo no especificado para

convertir esa malla en una triangulación de Delaunay. En el proyecto este último paso se implementó con una solución directa que resulta en errores frustrantes aunque muy poco frecuentes. Los vecinos se ordenan posteriormente en sentido horario para la generación de modelo.

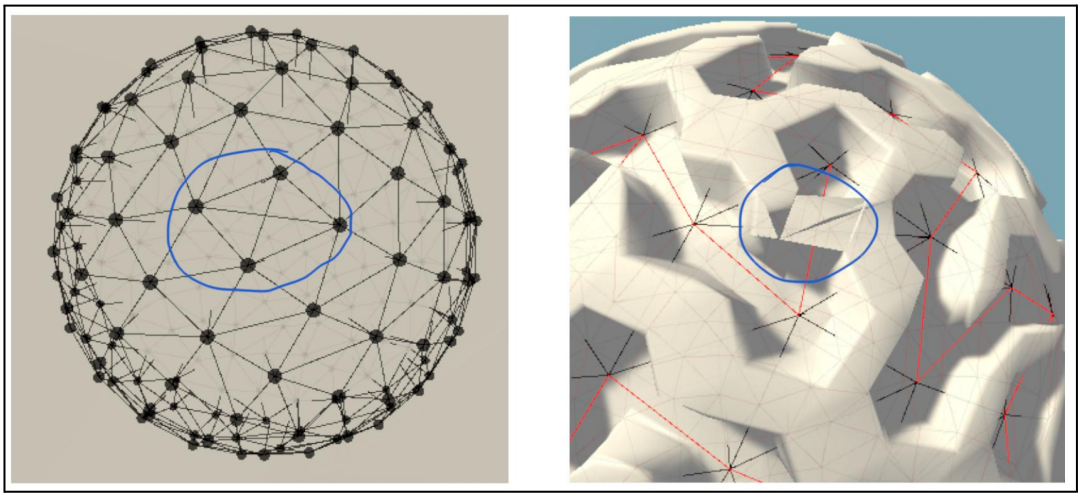


Figura 18: Error de triangulación
Se genera una conexión cruzada en el grafo. En el modelo, la misma región genera un “túnel” ya que el laberinto conserva una de las conexiones.

Para las conexiones entre capas se utiliza una implementación aún más burda donde cada punto de un nivel se proyecta al nivel superior y busca el punto más cercano en este. Esto resulta en conexiones que no son completamente verticales.

Por último, como no hay caras que organicen la orientación, se utiliza el vector del origen a cada nodo normalizado como normal del nodo y un vector cualquiera perpendicular a la normal como vector norte (lo relevante es que los vecinos estén ordenados).

3.2.7 De grafo a laberinto

Partiendo del grafo generado por los métodos anteriores, se puede generar un laberinto de muchas maneras, generalmente sencillas. El método implementado es una modificación de la exploración en profundidad del grafo que explora un vecino aleatorio de cada nodo hasta que se hayan explorado todos.

La versión más simple de este algoritmo no distingue entre vecinos superficiales y vecinos de capas superiores o inferiores, por lo que genera comúnmente laberintos con una cantidad absurda de cambios de nivel.

Modificando el algoritmo, se puede limitar la exploración a cada capa y ejecutarlo una vez en cada una. Las conexiones verticales se añadirían en antes de cada ejecución, determinando el punto de entrada de cada nivel.

3.3 Generación de modelos

La generación de modelos se hace proceduralmente y por niveles. Solo es práctico ver un nivel del laberinto a la vez, por lo que los niveles generan modelos diferentes que se ocultan o revelan a voluntad.

Originalmente, se usaba una plantilla de vértices relevantes que se copiaba con el desplazamiento y orientación adecuados en cada nodo y se conectaban entre sí siguiendo el grafo del laberinto. Esta implementación era rígida, ya que solo permitía laberintos cuyos nodos tenían el mismo número de conexiones.

3.3.1 Celdas procedurales

Los vértices del modelo se generan en relación con los nodos del grafo, marcando los puntos donde puede haber esquinas. Para ello, se convierte la lista ordenada de vecinos en una lista de direcciones. Cada dirección apunta a un vecino del nodo, pero lo que se necesita es la dirección de las esquinas. Aquí entra en juego que la lista esté ordenada, ya que se puede generar una nueva lista de vectores que contiene las direcciones intermedias entre cada par de vecinos contiguos (último y primero de la lista serían contiguos).

Con combinaciones entre las direcciones entre vecinos y el vector normal correspondiente al nodo, se puede generar una “celda” de puntos que lo rodean. Cada vecino del nodo tiene así cuatro vértices en su dirección.

Por el momento, los únicos vértices que se conectan son los que forman el suelo de la celda.

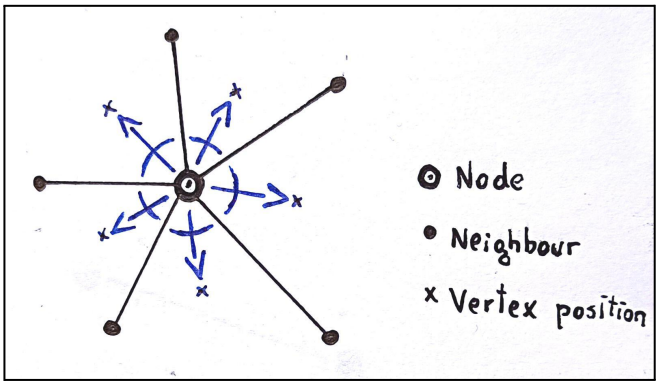


Figura 19: Posición de los vértices
Dirección de vértices y esquinas (azul) a partir de vecindad del grafo (negro).

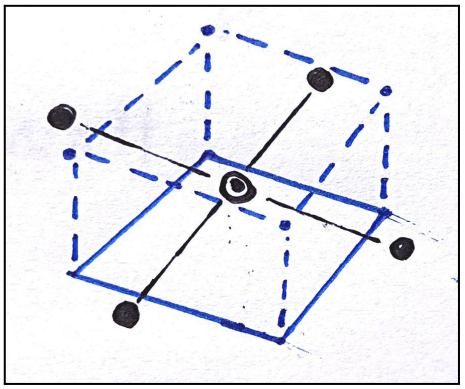


Figura 20: Celda de vértices
Modelo en azul y grafo en negro. Hay 4 vértices en la dirección a cada vecino.

3.3.2 Caminos procedurales

A partir de los vértices generados para cada nodo se tienen que formar los caminos y muros del laberinto. El primer paso es localizar los vértices relevantes, para lo que se tiene una lista que detalla cuál es el primer vértice correspondiente a la celda de cada nodo. El número total de vértices y posición exacta de los vértices correspondientes a un vecino de ese nodo dependen de la lista de vecinos, que impone orden en su generación.

Ambas posibilidades tienen sorprendentemente pocas diferencias en su formación. Como se ve en el diagrama (Figura 21): para un camino se genera un suelo conectando los vértices

inferiores de ambos nodos y dos paredes conectando estos con los superiores de cada lado; mientras que para un muro se genera un techo que conecta los vértices superiores de ambos nodos y dos paredes que conectan los inferiores y superiores de cada nodo independientemente.

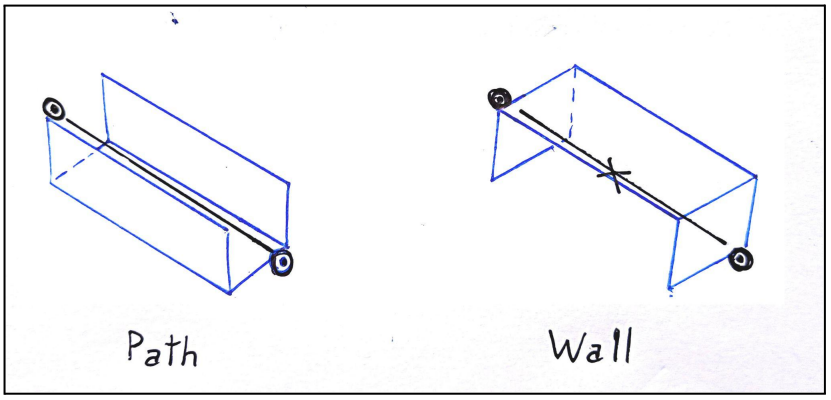


Figura 21: Caminos y muros
Modelo para un camino y para un muro entre dos nodos, conectando los 8 vértices correspondientes.

Una vez terminadas las conexiones, solo faltan los huecos intermedios, que existen en el espacio entre más de dos nodos, que se consigue conectado solo nodos superiores.

3.3.3 Subir y bajar

Para representar un nodo con conexión hacia el nivel inferior, se puede cambiar el suelo de su celda a un agujero. Eliminar directamente el suelo es una opción válida pero estéticamente cuestionable, por lo que se genera una copia de los vértices del suelo a menor distancia del centro de la celda y se conectan los dos conjuntos en un pequeño borde.

Para representar caminos hacia fuera se añade un foco de luz. Este método es más sensible a la dirección estética con la que pueda acabar el proyecto, pero aún así es bastante efectivo.

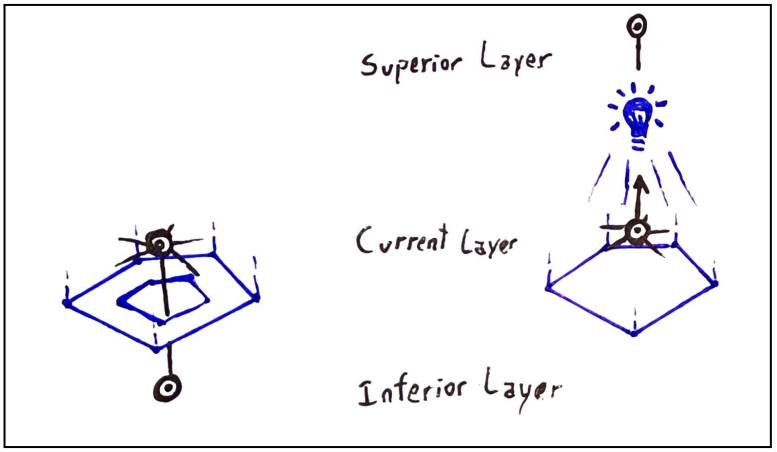


Figura 22: Pozos y luces
Cambios en el objeto (azul) según las conexiones verticales del laberinto (negro).

3.3.4 Detalles de la generación de modelos

Lo expuesto anteriormente es una versión simplificada de la generación de modelos. Por razones técnicas y estéticas, la implementación requiere ciertas complicaciones.

El sistema de modelos de Unity utiliza vectores asociados a cada vértice para interpolar el vector normal en cualquier punto de un triángulo. Por ello, muchos modelos tienen vértices duplicados para cada cara de la que forma parte (el modelo del cubo de Unity tiene 24 vértices en lugar de 8).

Con respecto al laberinto, duplicar los vértices no es una solución tan simple ni efectiva. Por un lado, es generación procedural, por lo que se tiene que predecir qué copia del vértice usar. Además de eso, las caras de las esquinas, sean “techos” o “suelos” tienen diferente orientación que las de las caras que conectan. Con todo esto, hacen falta al menos 3 copias de cada vértice (que no son precisamente escasos).

La solución implementada renuncia mayormente a tener caras completamente planas. Se usan dos copias de los vértices, una para las caras que forman paredes (independientemente de dirección) y otra para suelos y techos. El efecto es que los bordes dentro de esos dos grupos se suavizan y se mantienen los bordes “duros” entre los dos grupos, que son más importantes para la sensación de relieve.

Como último detalle estético, se separan ligeramente las dos copias y se generan las caras entre ellas. Aunque esto suaviza esos bordes y afecta al cálculo de normales de Unity, no es tan pronunciado.

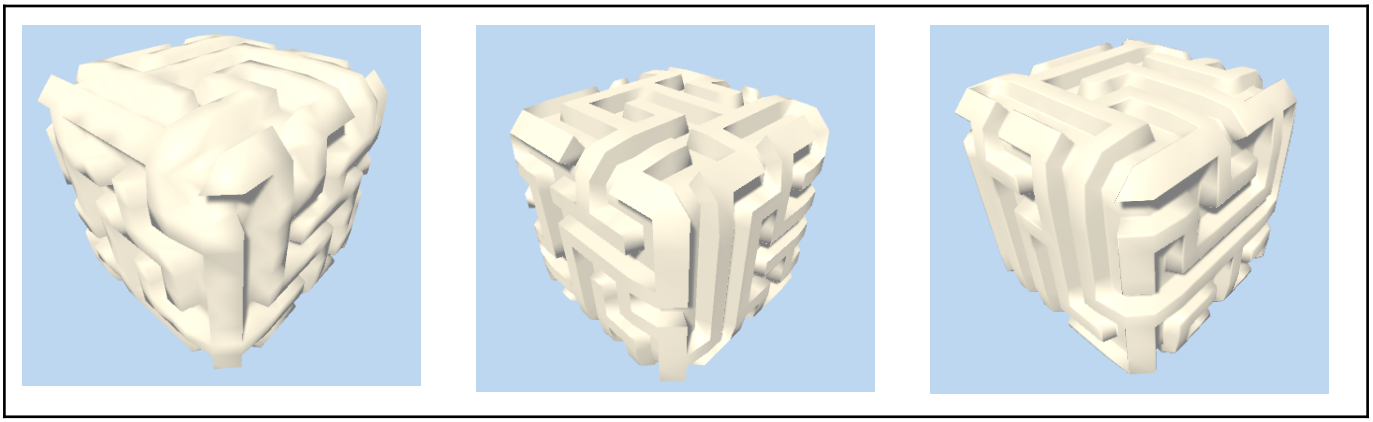


Figura 23: Versiones del modelo 3D
Comparación entre versiones. El modelo sin copias de vértices (izquierda) hace difícil distinguir esquinas. El modelo que duplica vértices (medio) remarca más las esquinas entre suelos, muros y techos. El último modelo (derecha) suaviza las esquinas sin hacerlas irreconocibles.

3.4 Navegación y cámara

3.4.1 Navegación Modular

El movimiento está dividido en un navegador y un controlador.

MazeNavigator es responsable de limitar el movimiento del objeto al laberinto. Internamente, solo conoce los dos nodos entre los que está y la proporción de proximidad entre uno y otro. Para el movimiento se usan principalmente dos métodos: uno para identificar un vecino que esté en una dirección dada y otro para moverse en dirección a un nodo. El navegador considera diferentes nodos según la posición del navegador entre los dos nodos de referencia.

MazePlayerController conecta el *input* del jugador al navegador. Como se ha mencionado en el apartado de diseño, el control del jugador depende de un punto de referencia. El controlador transforma una dirección de *input* de acuerdo al sistema de referencia y busca el nodo que puede haber en esa dirección. Si lo encuentra, avanza en esa dirección de manera proporcional a la magnitud del movimiento y a la alineación entre la dirección de *input* y la dirección del nodo.

La separación del sistema de movimiento en dos subsistemas está motivada por la posibilidad de añadir otro tipo de controladores, como un explorador automático o uno con inercia y aceleración.

3.4.2 Cámara

El control de cámara controla el punto de observación ideal de un navegador (no necesariamente controlado por el jugador) y la posición de la cámara respecto a este.

Actualmente, el *ObserverController* es un singleton, pero esto sería sencillo de cambiar en caso necesario.

El punto de observación no es necesariamente el punto en el que está la cámara. El observador permite inclinar la cámara respecto al navegador observado. El controlador del jugador usa el punto de observación y no la cámara como punto de referencia, lo que asegura que la inclinación de esta no convierta movimientos por la superficie de un laberinto a movimientos entre niveles.

3.5 Controles

Los controles son responsabilidad de una clase estática que los agrupa en ejes y vectores. Los ejes son números decimales del -1 al 1. Los vectores son conjuntos de ejes relacionados. Así se puede obtener cada eje en grupo y por separado.

El vector XYZ es responsable del movimiento, siendo el plano XZ el movimiento superficial.

El vector UVW son los controles flotantes de cámara, que incluyen UV como control de inclinación y W como control de rotación.

El vector DPad agrupa los dos ejes del DPad. Actualmente, se usa solo el eje vertical del DPad para controlar el zoom.

Capítulo 4 Conclusiones y líneas de expansión

4.1 Conclusiones

Los objetivos y sistemas principales del proyecto se han completado satisfactoriamente. El juego tiene un sistema para generar laberintos que es flexible a ampliaciones o modificaciones. Hay seis variantes funcionales, que usan las formas de los cinco sólidos platónicos y la esfera. Los laberintos se exploran usando modelos generados proceduralmente para cada capa.

Los controles de movimiento son simples y responden intuitivamente a cambios de perspectiva. La cámara se adapta a la posición del jugador y permite cambios de perspectiva en una gran variedad de formas.

El juego no está completo, principalmente por la falta de interfaces y lógica de juego, pero también por el escaso diseño estético. Esto último destaca en la ausencia total de audio, que es un recurso necesario para que un juego se sienta completo.

Aun así, el diseño y el trabajo realizado forman una fundación sólida sobre la que se puede expandir cómodamente.

4.2 Líneas de expansión

4.2.1 Correcciones

Los primeros cambios que mejoran el proyecto serían las correcciones de funcionalidad. El principal error está en la triangulación de Delaunay que se usa para generar el grafo de la esfera. Rara, pero inevitablemente, comete un error que causa una conexión cruzada, que el modelo a menudo interpreta como túneles.

Con correcciones funcionales solucionadas, el formato del código puede repasarse. Los laberintos de sólidos usan listas de cuaterniones indescriptibles para definir correctamente sus caras. Buscar una representación más legible que no pierda exactitud estaría entre los siguientes objetivos.

4.2.2 Mejoras

Aparte de correcciones al trabajo ya hecho, se puede completar el proyecto para que sea realmente jugable. Esto principalmente se refiere a sistemas simples que conecten y den sentido a los laberintos generados. Entre otras cosas, haría falta añadir:

- Un objetivo o condición de victoria del laberinto
- Un menú para seleccionar niveles y dificultades
- Opciones de controles y cámara libre

También sería conveniente explorar y comparar otros algoritmos de generación de laberintos, ya que el único implementado tiende a generar menos bifurcaciones dependiendo de la cantidad de vecinos que hay por nodo.

4.2.3 Expansiones

Con esos cambios, el proyecto sería por fin un “juego”. A partir de ahí se podría expandir para ser un “buen juego”. Para esto haría falta pensar bien en la experiencia del jugador para tener dirección. Por ejemplo, se podría trabajar en:

- **La experiencia de nivel**: Trabajar en los objetivos y herramientas de una partida. Permitir que el jugador marque regiones visitadas o deje balizas, por ejemplo.
- **Estética**: Trabajar en la dirección estética del juego. Esto incluiría añadir efectos de audio por movimiento y cambios de nivel, alterar las luces y modelos para diferenciar estéticamente las capas de un mismo laberinto y añadir música o alterar el menú para que sea atractivo, representativo y fácil de navegar, entre otras cosas.
- **La experiencia de juego**: Trabajar en la relación entre el jugador y el juego. Una opción muy atractiva de este tipo sería añadir movimiento basado en aceleración e implementar controles con giroscopio de móvil.

Capítulo 5 Conclusions and future work

5.1 Conclusions

The main objectives and systems of the project have been successfully completed. The game has a maze generation system that is flexible for expansions and modifications. There are six functional variants that use the shapes of the five Platonic solids and the sphere. The mazes are explored using procedurally generated models for each layer.

The movement controls are simple and intuitively respond to perspective changes. The camera adapts to the player's position and allows for perspective changes in a variety of ways.

The game is not complete, mainly due to the lack of interfaces and game logic, as well as the limited aesthetic design. In this regard, the absence of audio stands out, as it is a necessary resource to feel that the game is truly complete.

Nevertheless, the design and work done form a solid foundation upon which further expansion can take place.

5.2 Expanding the project

5.3 Corrections

The first changes that would improve the project involve functionality corrections. The most notable issue lies in the Delaunay triangulation used to generate the sphere's graph, as it occasionally produces a crossed connection, which the 3D model often interprets as tunnels.

Once functional corrections are addressed, the code structure should be revised. The maze solids utilize unintelligible quaternion lists to properly define their faces. The goal in that case would be to find a more readable representation that does not sacrifice accuracy.

5.4 Enhancements

In addition to corrections to the existing work, the project can be completed to make it truly playable. This mainly refers to implementing simple systems that connect and give meaning to the generated mazes. Among other things, the following would need to be added:

- An objective or victory condition inside each maze
- A menu for level and difficulty selection
- Control options and free camera mode

It would also be beneficial to explore and compare other maze generation algorithms, as the one currently implemented tends to generate fewer branches depending on the number of neighbors per node.

5.5 Expansions

With the previous changes, the project would finally become a "game". From there, it could be expanded to become a "good game". This would require careful consideration of the player's experience to provide direction. These are some ideas to consider:

- **Level design**: Developing objectives and tools for a game session. Allowing the player to leave markers or paint the paths already explored, for instance.
- **Aesthetics**: Working on the visual direction of the game. This would include features like adding audio effects for movement and level changes; modifying lights and models to add aesthetic differences between layers of a maze; adding music; and improving the menu to make it attractive, representative, and easy to navigate, among other things.
- **Gameplay experience**: Enhancing the gameplay experience for the player so that it’s more satisfying to play. One appealing option would be to add acceleration-based movement and implement gyro controls to make a build for smartphones.

Capítulo 6 Presupuesto

6.1 Desglose de costes

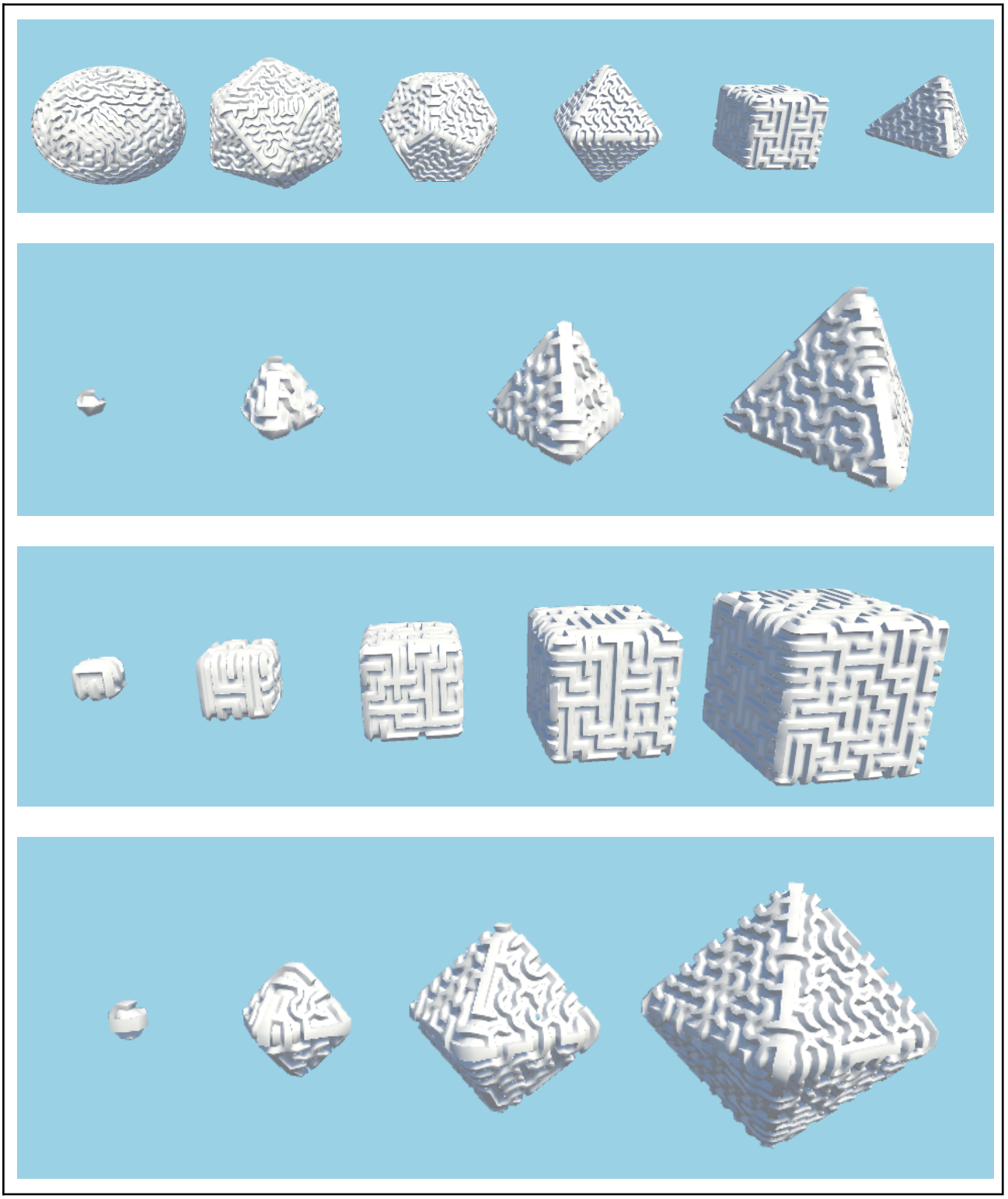
Para calcular el precio del trabajo se usa una tasa de 20 € por hora, que se aproxima al salario medio de un desarrollador de videojuegos en España.

RECURSOS HUMANOS		
Descripción	**Horas**	**Coste**
Investigación	20	400 €
Diseño y prototipado	55	1100 €
Desarrollo	225	4500 €
MATERIALES		
Descripción		**Coste**
Ordenador de sobremesa		1050 €
TOTAL		
Descripción		**Coste**
Recursos Humanos		6000 €
Materiales		1050 €
	Total	**7050 €**

Tabla 1: Presupuesto

Apéndice 1: Resultados

A1.1 Laberintos



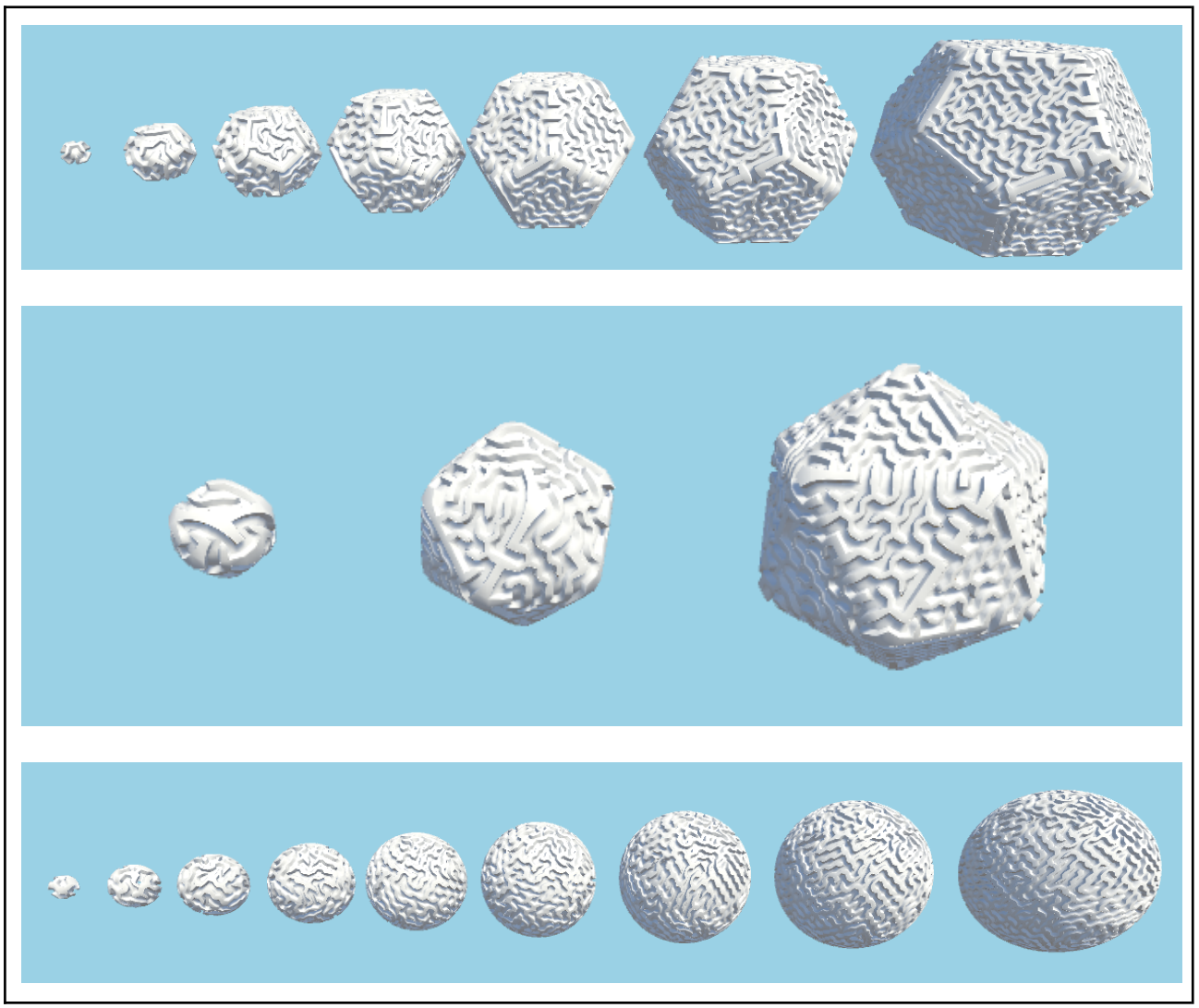


Figura 24: Ejemplo de cada tipo de laberinto

A1.2 Capturas de juego

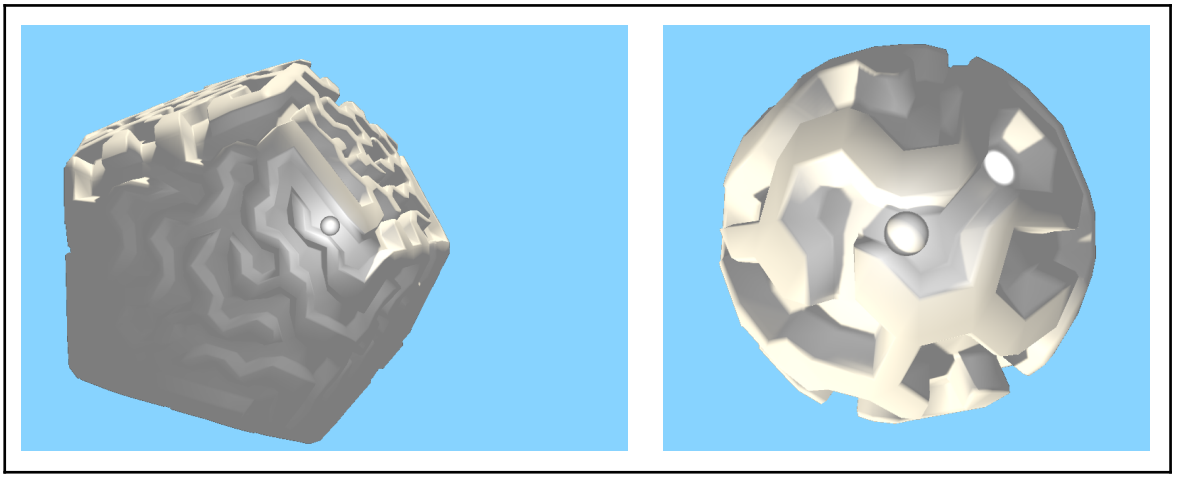


Figura 25: Exploración por el laberinto I
Se puede ver una salida a la capa superior en la imagen de la derecha

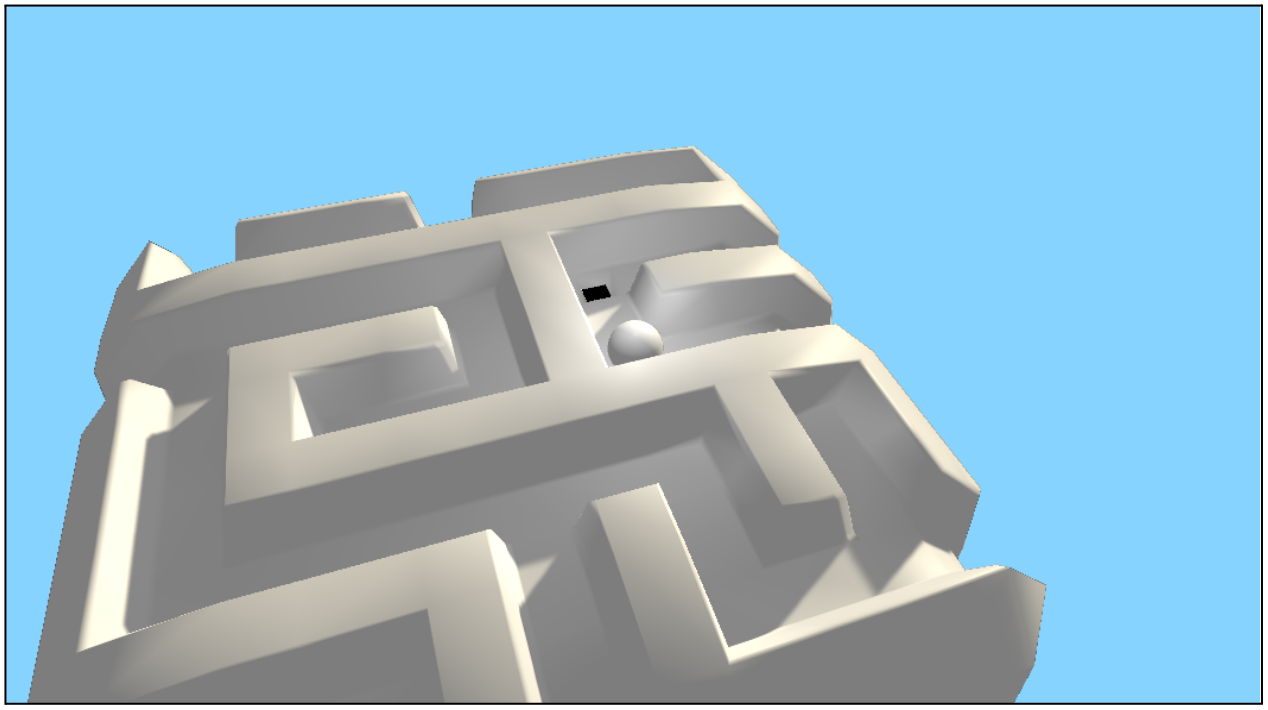


Figura 26: Exploración por el laberinto II
Se aprecia el uso de los controles de inclinación. Se puede ver una entrada a la capa inferior.

Bibliografía

[0] Cerviño Luridiana, D. (2023, July 14). TFG Concentric Labyrinths. https://github.com/alu0101315058/TFG_ConcentricLabyrinths

[1] Wikimedia Foundation. (2023, July 2). *Maze Generation algorithm*. Wikipedia. https://en.wikipedia.org/wiki/Maze_generation_algorithm

[2] DK Games. (n.d.). *Maze Planet 3D 🕹️ play on CrazyGames*. 🕹️ Play on CrazyGames. https://www.crazygames.com/game/maze-planet-3d

[3] Jishenaz. (2013). Xbox Controller. File: Xbox Controller.svg. Retrieved July 13, 2023, from https://commons.wikimedia.org/wiki/File:Xbox_Controller.svg.

[4] Reusser, R. (n.d.). Fibonacci sphere. ricky reusser. https://rreusser.github.io/fibonacci-sphere/

[5] CR Drost. (2020, July 27). Evenly distributing n points on a sphere. Stack Overflow. https://stackoverflow.com/questions/9600801/evenly-distributing-n-points-on-a-sphere/44164075#44164075

[6] Delaunay+Voronoi on a sphere. (2018, October 19). https://www.redblobgames.com/x/1842-delaunay-voronoi-sphere/

[7] Roberts, M. (2020, October 5). Evenly distributing points on a sphere. Extreme Learning. https://extremelearning.com.au/evenly-distributing-points-on-a-sphere/

[8] Wikimedia Foundation. (2023a, May 20). Stereographic projection. Wikipedia. https://en.wikipedia.org/wiki/Stereographic_projection

[9] Sinclair, D. (2010, July 15). S Hull: A fast sweep-hull routine for Delaunay triangulation. S Hull. http://s-hull.org/