Diseño y desarrollo de un prototipo de un videojuego de aventuras con contenido creado procedimentalmente

Grado en Ingeniería Informática

Trabajo de Fin de Grado

Diciembre 2015

Autor: Lucas Roig Pérez

Tutor: Agustín Trujillo Pino

Contenido

1. Objetivos del proyecto

Siempre me han fascinado los videojuegos. Es increíble que una persona pueda pensar en una experiencia, plasmarla en forma de un videojuego, y ver cómo a la gente le atrae esa idea, y se divierte con ella. Para mí, eso era pura magia. Un proceso indescifrable que nunca llegaría a entender del todo.

Entré en la carrera en parte con la esperanza de conseguir entender este universo tan complejo (y muchos otros), así que creo que mi elección de proyecto fue bastante natural.

Con este proyecto quería llevar a cabo por mi cuenta una idea, aunque fuera pequeña, y ver cómo cobraba vida. Quería ver que, de la nada, de repente aparecía un mundo en el que puedes hacer cosas. Y que ese mundo había sido creado por mí.

Además, quería experimentar con algo que me ha llamado la atención desde que descubrí su existencia: la generación procedimental. Cómo hacer que cosas aparezcan de la nada a partir de funciones matemáticas y procedimientos que moldean el mundo en tiempo de ejecución.

A lo largo de este proyecto, trabajaremos tres módulos principales:

- Crearemos un generador de islas usando principalmente Perlin Noise.
- Crearemos un generador de mazmorras usando principalmente un algoritmo BSP.
- Usaremos ambos generadores y crearemos un prototipo de un videojuego en el que controlaremos a unos aventureros que exploran mazmorras en una isla.

A cada paso, haremos referencia a los requisitos planteados por el GDD (Anexo I pág. 55) para solucionar cualquier problema que nos encontremos o para ayudarnos en la toma de decisiones.

También, y de manera introductoria, veremos el estado actual de la industria de los videojuegos, dónde nos podríamos situar nosotros dentro de la misma, y qué podríamos hacer al respecto.

2. Aportaciones

Las empresas más grandes de videojuegos, las que más dinero generan, se basan en franquicias que explotan año tras año, con poca mejora en los juegos en sí, sin contar actualizaciones tecnológicas en gráficos, inteligencia artificial o juego en red.

La mayoría de estos títulos, desgraciadamente, son juegos de lucha, juegos de guerra, juegos de tiros y similares, en los que uno de los atractivos principales, aunque no quieran reconocerlo, es la violencia.

Los juegos indie no pueden competir en capacidades técnicas con juegos de grandes compañías porque, obviamente, no tienen tanto dinero. Esto les lleva a cambiar su estrategia a algo totalmente diferente: la originalidad.

Y es que los desarrolladores indie son muy arriesgados, crean juegos con situaciones que nadie pensaría que podrían ser un juego. Crean juegos "experimentales" en los que no hay ningún botón que se llame "disparar" y muchos de ellos piensan de una manera tan enrevesadamente diferente, que a veces triunfan con sus creaciones y venden cientos de miles de copias.

Yo pienso que el desarrollo de juegos indie y la experimentación es muy importante para la industria de los videojuegos. La mantiene fresca y la renueva cada pocos meses.

Invito a cualquier persona que me esté leyendo y que esté interesada en la industria de los videojuegos, que apoye al movimiento indie o, mejor aún, que se una.

Solo así conseguiremos que el sector de los videojuegos madure, y que los títulos más divertidos no tengan que ser "pulsa este botón para matar".

3. Competencias

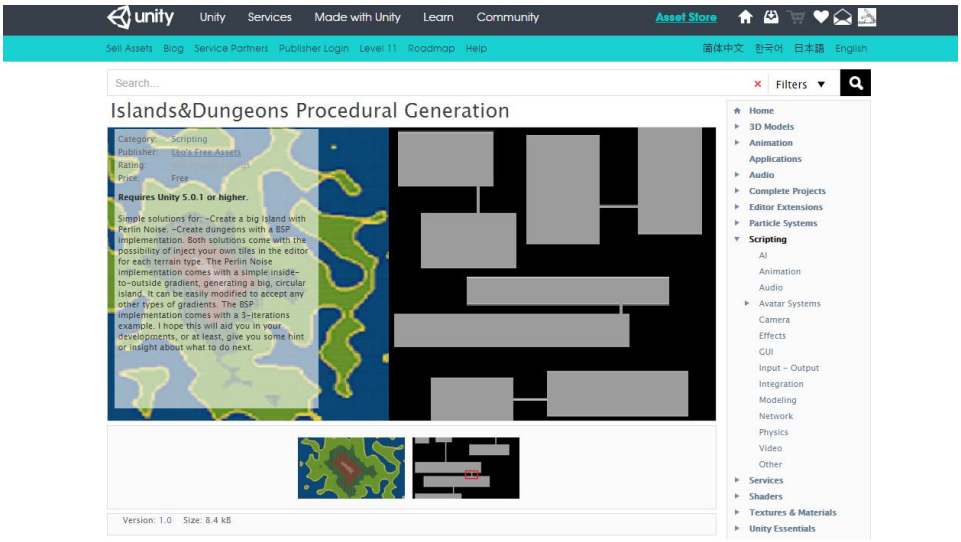
CII01. Capacidad para diseñar, desarrollar, seleccionar y evaluar aplicaciones y sistemas informáticos, asegurando su fiabilidad, seguridad y calidad, conforme a principios éticos y a la legislación y normativa vigente.

Tal como se explica en la fase de desarrollo, y teniendo en cuenta lo dicho en el apartado "Aportaciones", el proyecto recorre una línea previamente planificada. Cada nuevo añadido se reevalúa o se rediseña si es necesario. Todo con el objetivo de sacar adelante un prototipo que refleje los objetivos marcados inicialmente.

CII02. Capacidad para planificar, concebir, desplegar y dirigir proyectos, servicios y sistemas informáticos en todos los ámbitos, liderando su puesta en marcha y su mejora continua y valorando su impacto económico y social.

Antes de comenzar con el trabajo pesado del proyecto, se diseñó un GDD, que actúa tanto como fuente de información para el desarrollo del prototipo, como manual de requisitos y objetivos finales.

Además, se valora el impacto del trabajo realizado en este proyecto en la medida en que partes del mismo han sido subidas de manera gratuita a **assetstore.unity3d.com** para que otros desarrolladores puedan beneficiarse de las horas aquí invertidas.



TFG01. Ejercicio original a realizar individualmente y presentar y defender ante un tribunal universitario, consistente en un proyecto en el ámbito de las tecnologías específicas de la Ingeniería en Informática de naturaleza profesional en el que se sinteticen e integren las competencias adquiridas en las enseñanzas.

Todo lo aprendido a lo largo de la carrera ha sido una gran ayuda y soporte para la realización de este proyecto. El desarrollo, memoria y defensa del proyecto cubren esta competencia.

4. Estado actual de la industria de los videojuegos

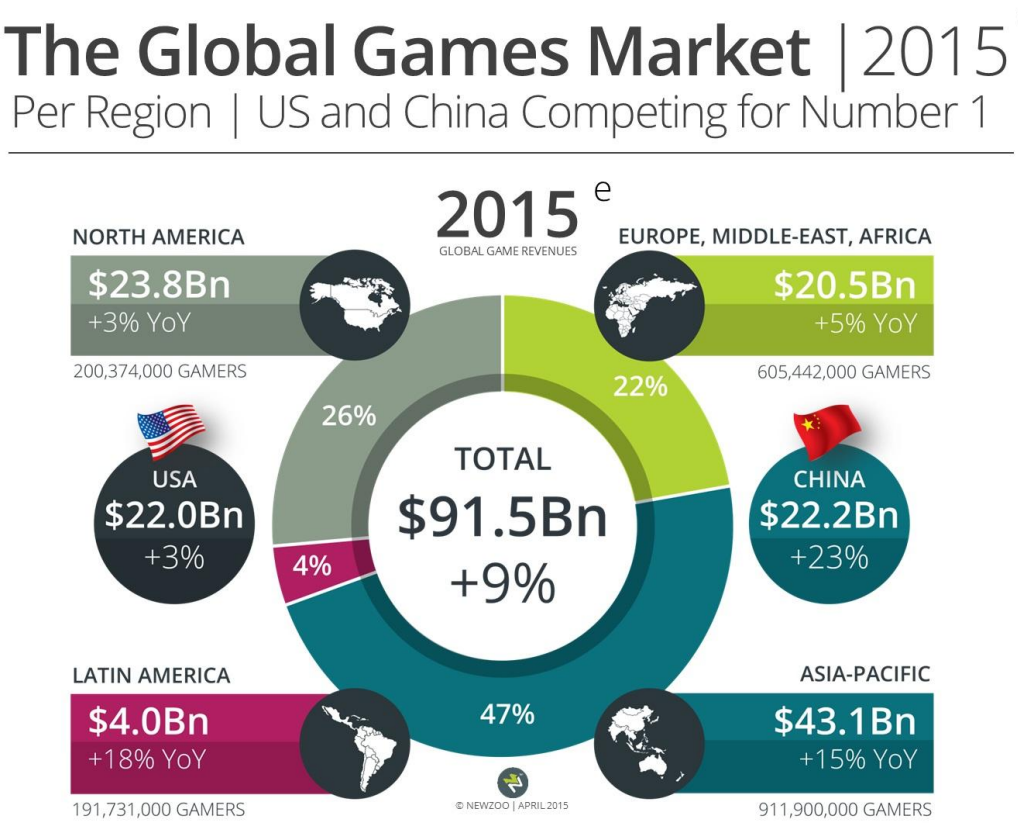
"Todo el mundo juega a algo". Al menos, casi todos. Con la "revolución" de los juegos de móvil ya parece habérsele quitado el miedo a echarse unas partidas hasta a nuestras abuelas, hecho que parecía absolutamente inaudito hace unos años.

Las consolas de toda la vida siguen generación tras generación mejorando sus especificaciones y compitiendo ferozmente por llevarse ese año más compradores que la competencia.

Los avances gráficos en juegos de ordenador empiezan a parecerse seriamente a la vida real.

Y todo ello sin contar con los avances en tecnologías de realidad virtual con aparatos como el Oculus Rift o el Project Morpheus.

Y es que la industria actualmente está ingresando más de 90 billones de dólares al año.

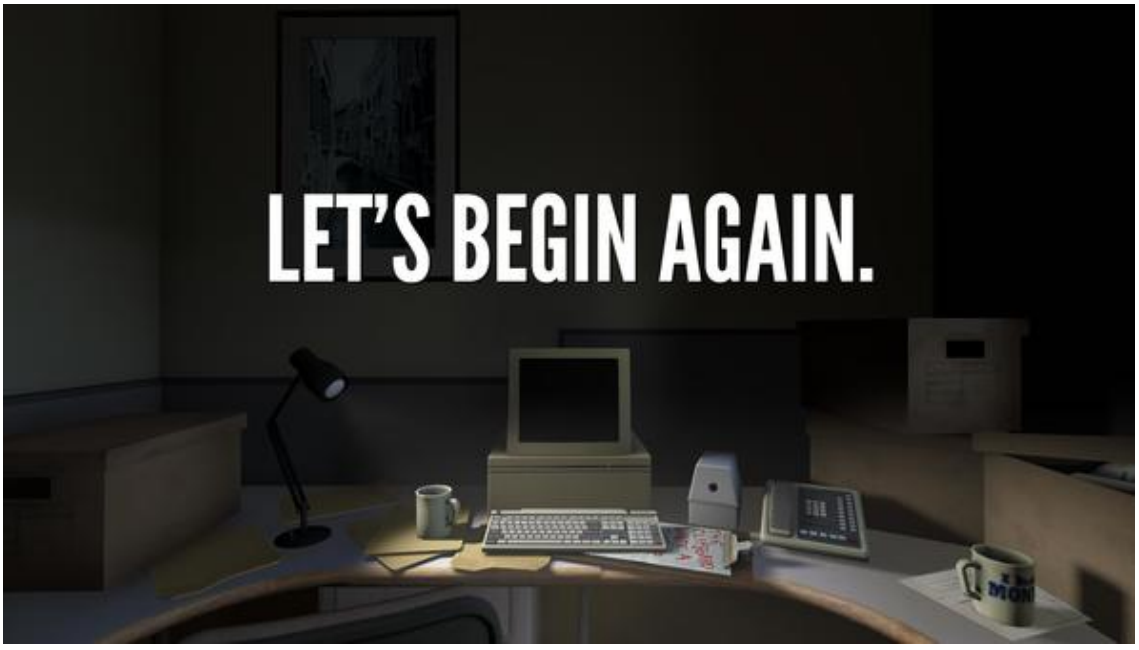


Y, en otro rincón apartado dentro de la industria de los videojuegos, se encuentran los juegos indies y los experimentales que, en muchos casos, se producen como si fueran obras de arte más que productos de mercado.

El Stanley Parabol es un ejemplo excelente. Se basa en algo que podríamos llamar "arriesgado" y que las grandes compañías, debido a la fuerte financiación recibida para hacer sus juegos, no se atreverían a hacer jamás.

El narrador cuenta *exactamente* lo que estás haciendo en todo momento y lo que *harás* a continuación. Y es que aunque te salgas de las pautas que el mismo juego te intenta marcar, el narrador se las ingenia para continuar contando la historia como si nada.

Stanley Parabol rompe las reglas clásicas de "el narrador cuenta lo que vas a hacer y luego lo haces". Más bien parece que el narrador cuenta lo que vas a hacer, y tú, como ser humano que eres, no vas a hacerle caso.



Otros juegos experimentan con las emociones humanas, nos hacen vivir situaciones tan reales que, aún siendo un juego con personajes ficticios, nos hacen pensar acerca de problemas actuales y de la gente que los está sufriendo.

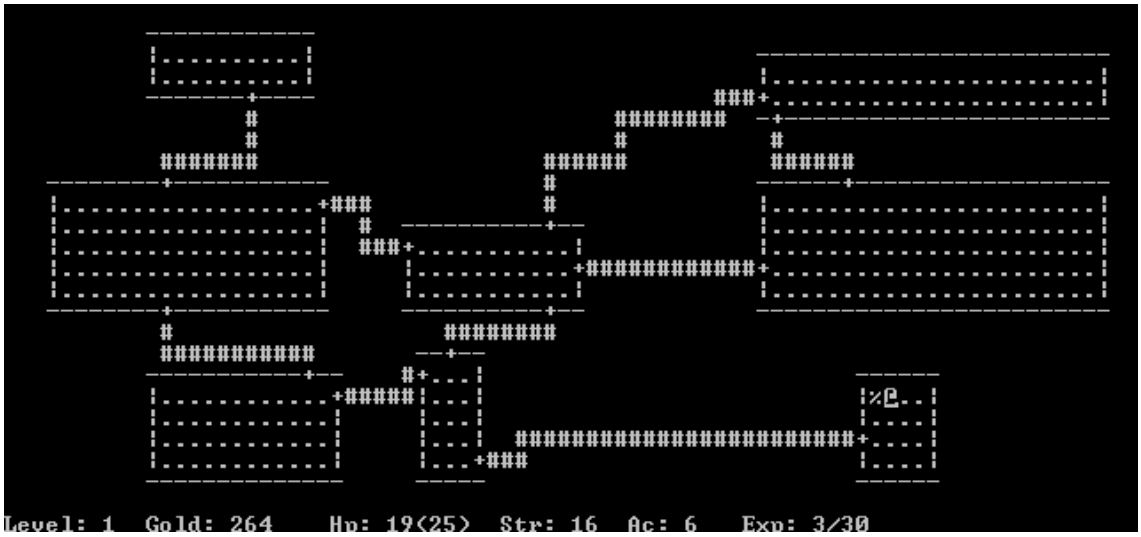
Por ejemplo, el *Papers, Please* nos pone en la piel de un inspector de aduanas en el ficticio estado comunista de Arstotzka. Tenemos que analizar los papeles de las personas que intentan pasar por nuestro puesto de aduana y decidir quién entra o no en el país, con la carga emocional que eso conlleva.

5. Juegos con generación procedimental de contenido

En el desarrollo de un videojuego es importante explotar tus fortalezas profesionales, pero es igualmente importante tener en cuenta tus debilidades.

En el caso de programadores entusiastas que desean crear videojuegos esto es un grave problema: no son capaces de generar arte para sus juegos.

Es un problema serio que no puedas crear la parte "vídeo" de un videojuego. Así que a algunos se les ocurrió explotar esta debilidad. Crearían juegos que tuvieran un gran peso de programación, hasta tal punto que el propio contenido lo crearían en base a líneas de código, algo que se les da muy bien. Luego, dibujarían los elementos del juego con caracteres ASCII:



Así es como nació "Rogue", que popularizó los juegos basados en exploración de mazmorra y con creación de contenido por procedimientos.

Que todo estuviera representado mediante caracteres ASCII no causó tantos problemas como uno podría imaginar. El hecho de que todo estuviera abstraído en forma de caracteres dejaba más hueco para la imaginación y, a las pocas horas, el jugador ya se está imaginando la forma y el color de todo mientras avanza con su personaje por estas pintorescas mazmorras.

La idea detrás de Rogue fue tan brillante, que actualmente existe un género de videojuegos llamado "rogue-like" que se basa en una serie de principios usados en el diseño del videojuego original.

Crear contenido para videojuegos es muy caro. Tienes que crear el arte necesario, escriptar las escenas, colocar los personajes, asegurarte de que todo está correcto... Consume mucho tiempo y dinero, y es algo que los desarrolladores independientes no siempre tienen disponible. Esta es una de las razones de que la generación procedimental de contenido sea tan popular entre este tipo de desarrolladores. Basta con crear los procedimientos adecuados, apretar un botón y ya tienes horas y horas de contenido aleatorio con el que hacer que tus jugadores disfruten.

Por supuesto, la complejidad de la programación de este tipo de desarrollos puede ser mucho mayor a simplemente escribir un guión y crear las escenas del juego a mano, pero también es cierto que los desarrolladores de este tipo de juegos suelen tener buena experiencia en programación como para estar a la altura.

Un ejemplo más moderno y más conocido puede ser Minecraft.

Originalmente concebido como un proyecto personal, el núcleo de Minecraft fue creado por una sola persona, y era capaz de generar infinidad de mapas diferentes, combinando biomas, elevación y zonas de agua, todo formado por cubos o voxels.

La naturaleza aleatoria (aunque controlada) de estos tipos de juegos les añaden una rejugabilidad asombrosa que ningún otro tipo de videojuego ha conseguido jamás.

6. Herramientas utilizadas

6.1. Unity 3D

C#, la enorme comunidad detrás del motor, actualizaciones constantes, versión gratuita muy completa, facilidad de uso de la herramienta... Unity tiene muchas ventajas, y por ello es uno de los motores gratuitos para desarrollo de videojuegos más usados en el mundo.

Gracias a que su lenguaje nativo es C#, muy parecido al que se da en la carrera (Java), la transición de uno a otro es casi imperceptible. Esto permite que en pocas semanas un usuario avanzado de Java trabaje prácticamente al mismo nivel de sutileza con C#.

Como detalle extra, la AssetStore y la cantidad de herramientas que cuenta el motor para hacer juegos en 2D, le dan unos cuantos puntos extra que, personalmente, valoro por encima de muchos otros motores.

6.2. Visual Studio 2013

A pesar de ser una herramienta de pago, como estudiante cuento con una clave para poder usarla. Y es que el Visual Studio es un entorno muy completo en el que implementar soluciones en C# y muchos otros lenguajes.

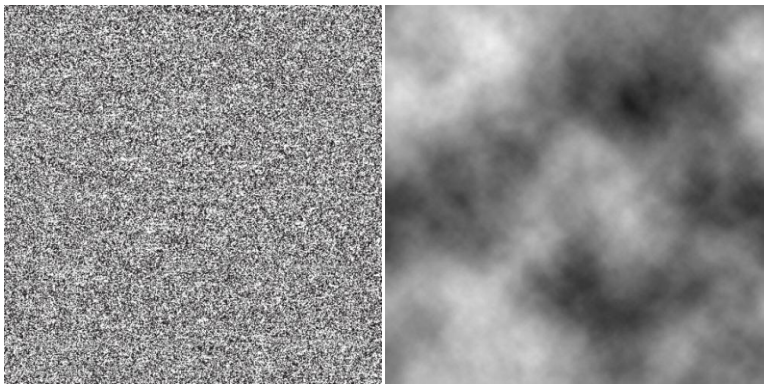
Entre otras ventajas, su estabilidad, ayuda de sintaxis y accesos directos me resultan muy cómodos y completos.

7. Desarrollo - Módulo 1: Perlin Noise

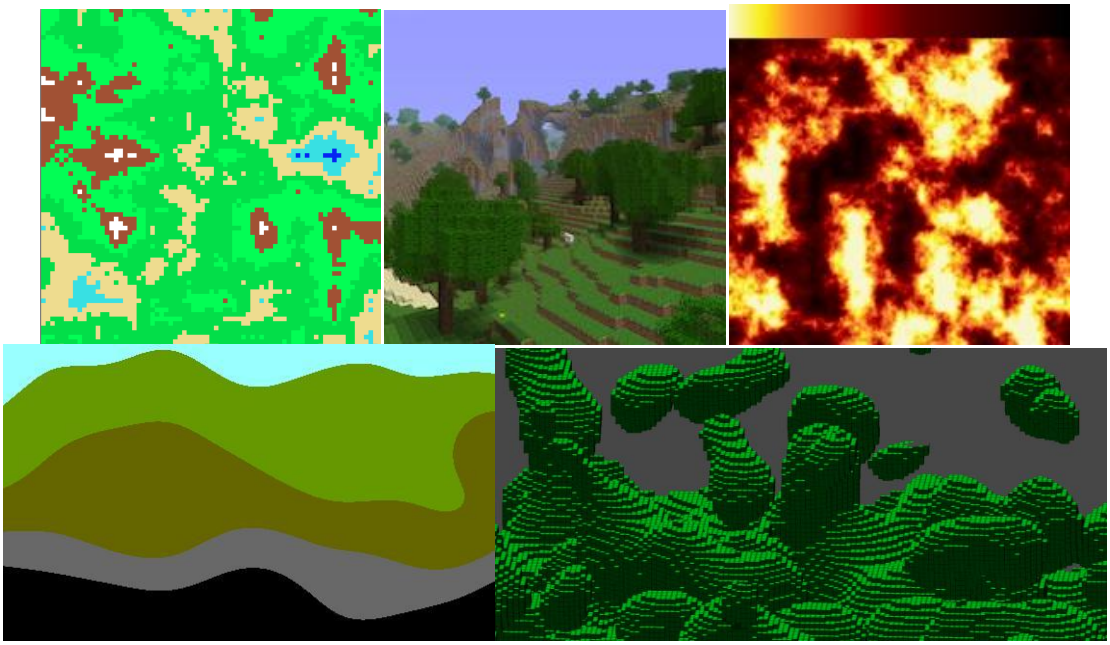
7.1. Qué es Perlin Noise

Perlin Noise es un tipo de ruido desarollado por Ken Perlin en 1963 como resultado de su frustración con el aspecto "antinatural" de las texturas en videojuegos de la época. Se le concedió un premio por el descubrimiento y el desarrollo de este algoritmo.

Perlin Noise es una función matemática que, interpolando entre diferentes gradientes precalculados, genera una textura que recuerda al ruido blanco. El punto fuerte es que se asemeja más a formas naturales como las nubes, o el fuego, que en el caso de ruidos más primitivos.



Perlin Noise se suele usar mucho en generación procedimental para crear texturas de fuego, de desgaste, nubes, mapas, elevaciones, simulación de líneas escritas a mano y muchas otras aplicaciones.



Si tomamos la textura generada como una matriz de valores de 0 a 1, podemos mezclarlo con otras técnicas y funciones, así como modificar los valores de la matriz, haciendo de Perlin Noise una técnica muy flexible.

7.2. Creando la isla

Al comenzar el desarrollo del generador de islas teníamos un par de ideas claras:

- Vamos a usar tiles cuadrados como representación final del terreno.
- Vamos a usar principalmente Perlin Noise para generar la isla completa.

En diseño estaba planteado que la isla sería el espacio de juego, y que ni crecería ni encogería durante toda la partida. Aprovechándome de esto, decidí usar una matriz de tiles como representación lógica de la isla, ya que nos brinda muchas facilidades para acceder a sus elementos.

Lo primero que hice fue crear la clase Tile, que tendría una variable que indicara el terreno al que representa:

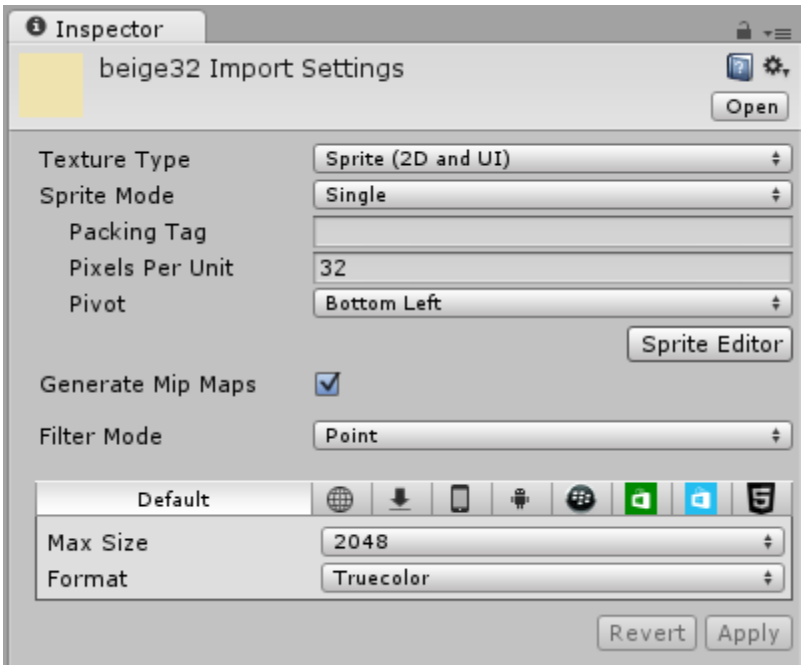
```
public class Tile
{
    public int x;
    public int y;
    public Type terrainType;
```

Únicamente con esto ya podríamos crear mapas rectangulares formados por pequeños tiles cuadrados, cada uno con un tipo concreto de terreno, y por lo tanto, visualmente diferentes.

Para probar todo esto necesitaríamos transformar la información lógica de la isla en imágenes pintadas en la pantalla de juego.

Para ello, nos creamos una serie de cuadrados de 32x32 píxeles con un editor de gráficos (como el MS Paint), cada uno de un color entero.

Los exportamos a Unity, y les ajustamos las propiedades para tratarlos como Sprites.



Para Unity, todas las imágenes son texturas. Una textura se puede usar para muchísimas cosas: puede ser la piel de un personaje en 3D, puede ser un "Mapa de Normales" que crea ilusión de profundidad en un objeto 3D, puede ser un "Mapa de Luces" para "pintar" luces en otras texturas...

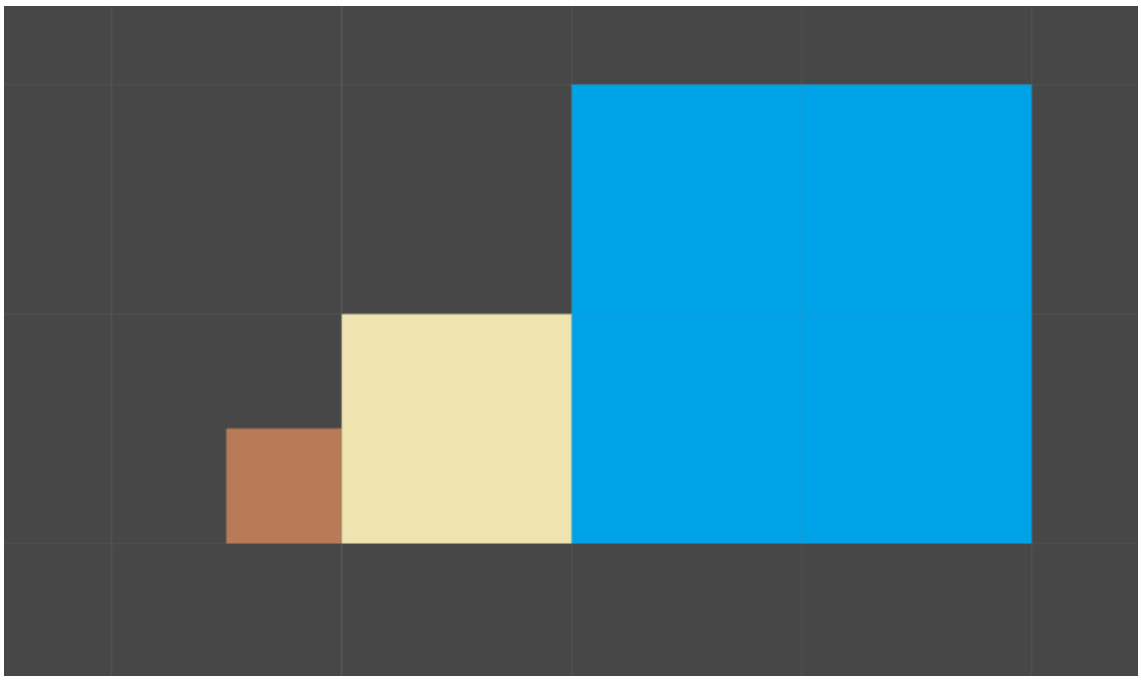
Nosotros queremos usar nuestras imágenes tal cual, como si fueran un dibujo, así que seleccionamos "Sprite" en la propiedad "Texture Type".

Un Sprite es un fotograma concreto que representa algo en el espacio de juego. Como los personajes, entornos y demás de un videojuego no suelen tener un tamaño en potencia de 2, se suelen crear hojas de sprites o Spritesheets en las que se colocan muchos sprites en una sola hoja transparente con un tamaño en potencia de dos, ahorrando mucho espacio de disco, ya que se puede comprimir mucho más fácilmente.

Nosotros no vamos a usar spritesheets para los terrenos de una isla, así que seleccionamos "Single" en la propiedad "Sprite Mode".

Vamos a usar imágenes de 32x32 píxeles como tamaño mínimo del juego. Es decir, la pieza más pequeña de nuestro juego medirá como mínimo 32 píxeles de ancho y de largo. Para facilitar nuestro trabajo en el editor (y en el código), vamos a fijar los Pixels Per Unit de cada tile a 32. De este modo, las posiciones de nuestras piezas de terrenos siempre van a ser números enteros.

En el siguiente ejemplo, el sprite azul está a 16 pixel por unidad de unity, el beige a 32 pixel y el marrón a 64:

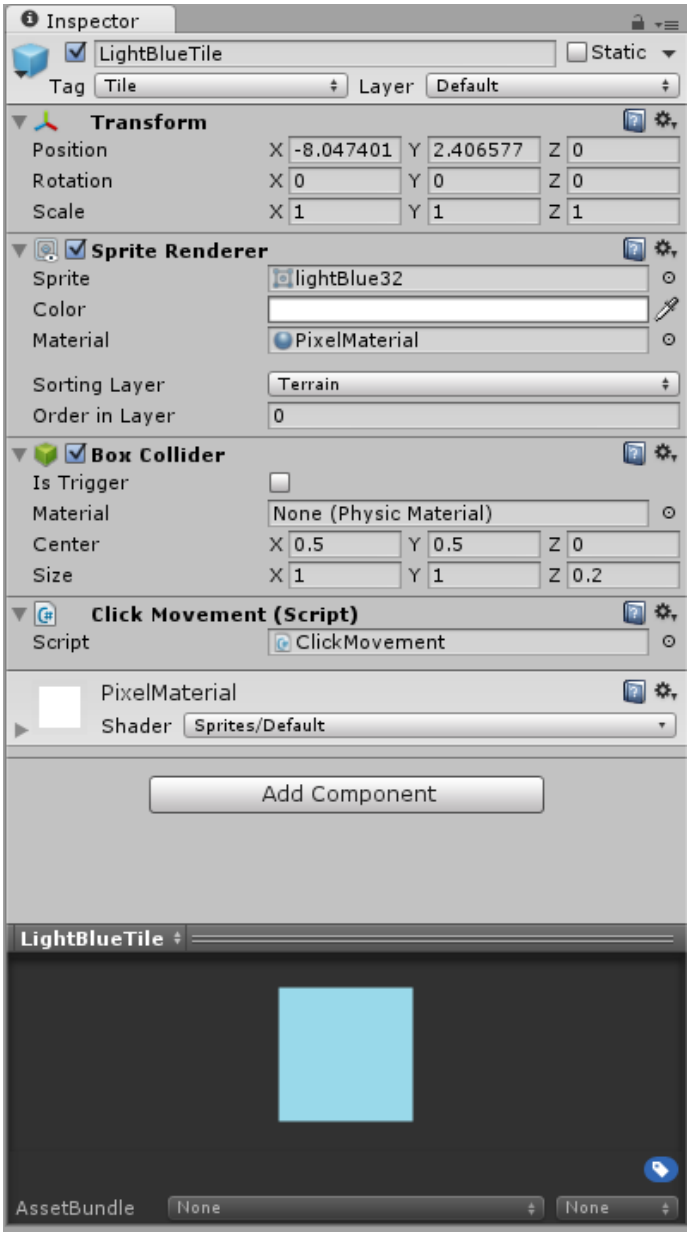


Ya tenemos todos los sprites importados en Unity. Podríamos usarlos tal cual, si arrastráramos un sprite dentro del editor, Unity le daría un componente transform, y eso es todo lo que necesitaríamos.

Pero como sabemos que vamos a trastear bastante con el terreno, lo mejor será crear primero un prefab para cada uno de los tipos de terrenos. De esta manera, si necesitáramos añadir funcionalidad extra al terreno, bastaría con añadírsela al prefab para que el cambio se propagase a todas las instancias de ese prefab.

Ejemplo de un prefab con los siguientes componentes:

- Transform, para situarlo en el espacio de juego.
- Sprite Renderer, donde podemos indicar la imagen asociada a ese tile.
- Box Collider, para poder usar colisiones y raycasting.
- ClickMovement, un script que añade la funcionalidad de poder ser clickado con el botón izquierdo del ratón.



Ya tenemos la representación lógica de la isla, y una colección de prefabs que representan los tiles que se verán en el juego, en este caso, con imágenes de colores.

Nos falta un intermediario, que coja toda la información de la isla y luego instancie el objeto pertinente para cada tile de la matriz.

Por suerte, y gracias a que hemos creado los prefabs anteriormente, la tarea es sencilla. Basta con recorrer la matriz y, por cada elemento, instanciar el prefab correspondiente al tipo de terreno que representa ese elemento.

```
for (int x = 0; x < width; x++)
{
    for (int y = 0; y < height; y++)
    {

        Terrain Type Switch

        tile = (GameObject)Instantiate(tileColor, terrainTiles.transform.position + new Vector3(x, y, 0), Quaternion.identity);
        tile.transform.parent = terrainTiles.transform;
    }
}
```

Por supuesto, necesitaremos un "traductor" que reciba un tipo de terreno y nos devuelva qué prefab tiene que instanciarse:

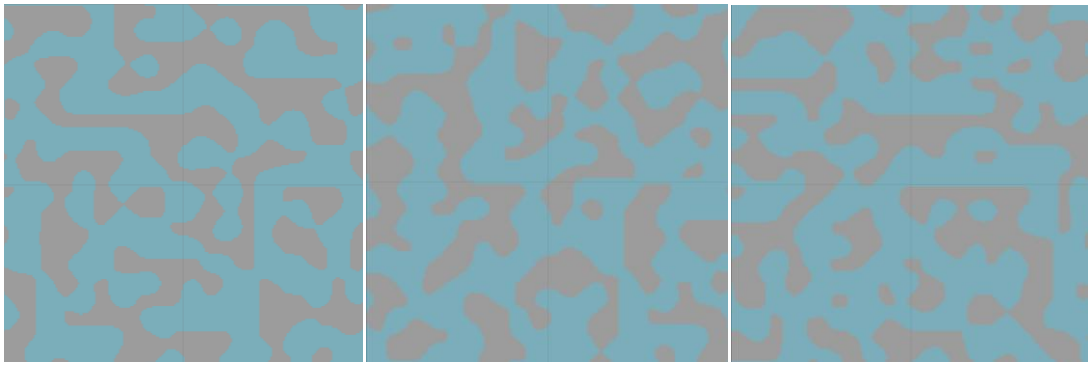
```
#region Terrain Type Switch
switch (map[x, y].terrainType)
{
    case Tile.Type.water:
        if (map[x, y].ocean)
            tileColor = deepWater;
        else
            tileColor = water;
        break;
    case Tile.Type.mountain:
        tileColor = mountain;
        break;
    case Tile.Type.grassland:
        tileColor = grassland;
        break;
    case Tile.Type.sand:
        tileColor = desert;
        break;
    case Tile.Type.snow:
        tileColor = snow;
        break;
    case Tile.Type.forest:
        tileColor = forest;
        break;
    case Tile.Type.greyTerrain:
        tileColor = grayTerrain;
        break;
    case Tile.Type.black:
        tileColor = blackTile;
        break;
}
#endregion
```

Tenemos todas las herramientas necesarias para ponernos a trabajar con la generación procedimental de la isla. Lo único que tenemos que hacer es, en vez de generar la matriz del terreno a mano, crear una serie de procedimientos que la vayan modificando hasta que quede un resultado que nos complace.

7.2.1. Primera parte: la forma

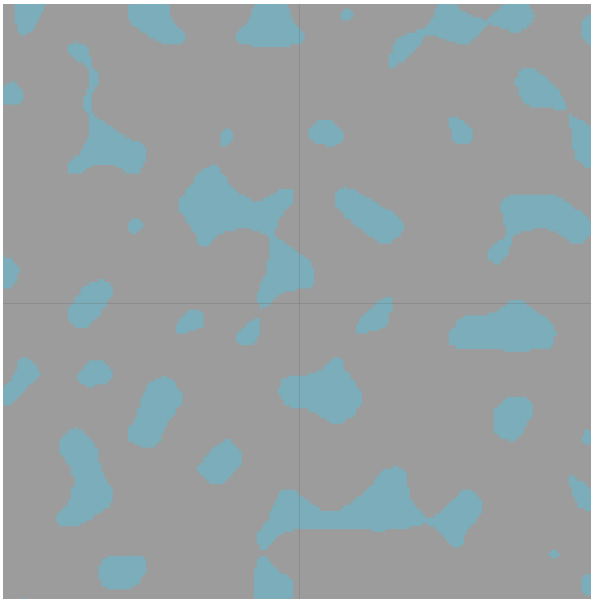
Una isla se compone principalmente de dos cosas, "tierra" y agua. Vamos a seleccionar uno de nuestros prefabs (el azul) para que sea el agua y otro (el gris) para que sea la "tierra". Esto simplifica mucho la generación de la isla. Primero nos preocuparemos de darle una forma ideal y ya luego veremos cómo pintamos el terreno dentro de la isla.

Recurrimos a Perlin Noise para hacer la primera prueba. Perlin Noise genera zonas de escala de grises, desde el blanco (1) hasta el negro (0), con formas muy naturales. Lo que vamos a hacer es, para cada elemento que nos devuelve Perlin, comprobamos su escala de blanco-negro. Si está por debajo de 0.5, entonces ese tile será de agua. Si es 0.5 o está por encima, el tile será de tierra.

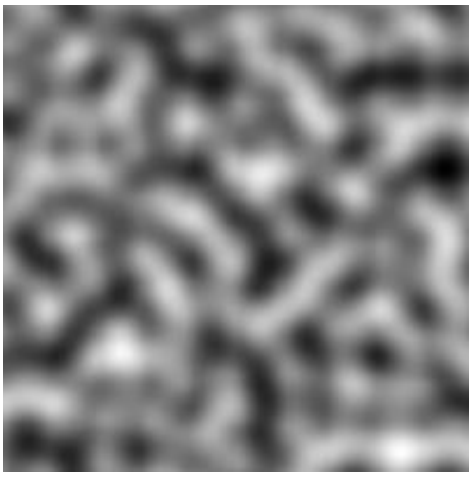


No está mal, pero tiene mucha agua. Además de que hay demasiadas islas separadas unas de otras, y nosotros queremos más bien una isla grande y alrededor algunas más pequeñas.

Vamos a intentarlo de nuevo, esta vez con agua < 0.3 y tierra >= 0.3 :



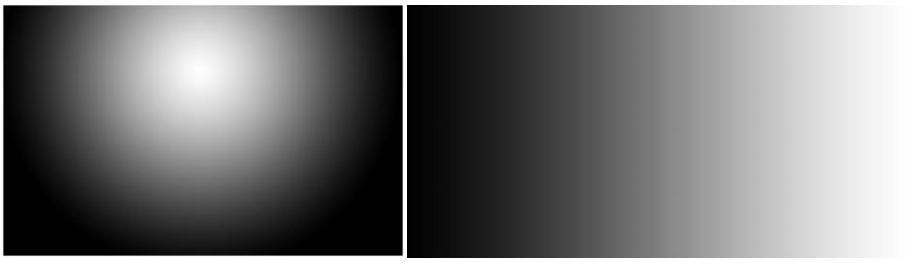
Claramente hay mucho más terreno que antes. Pero no se parece ni por asomo a una isla. Y es que tenemos un problema. Perlin sólo puede generar texturas de este tipo:



Por mucho que le demos vueltas a la historia, es imposible generar una isla grande y varias pequeñas alrededor de esta manera.

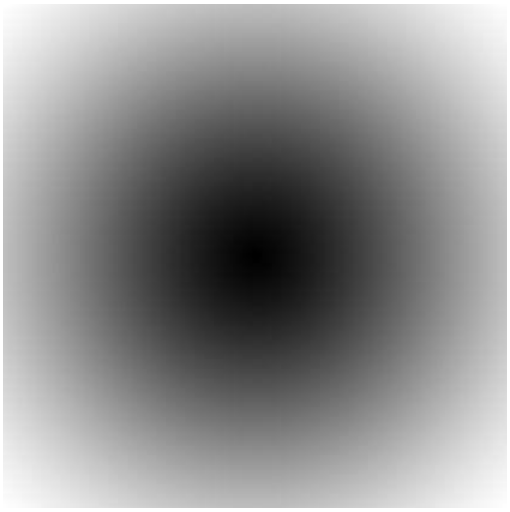
Nos gusta las formas y curvas que genera Perlin Noise y la sensación de "naturalidad" que queda al final, pero necesitamos mezclarlo con algo más para poder conseguir las formas que buscamos.

Después de mucho investigar, di con una posible solución, que además se presentaba como algo simple pero potente, algo que sería muy fácil de ampliar: usar degradados:

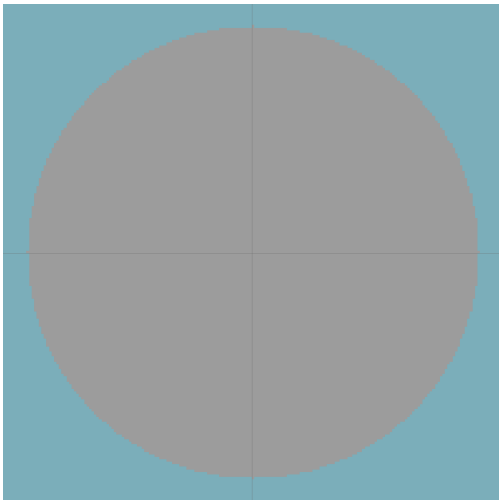


Perlin usa valores del 0 al 1 en escala de grises. Si usamos un degradado de blanco a negro, representándolo con valores float de 0 a 1 y lo mezclamos con los valores de Perlin, podríamos darle un toque que guíe esas curvas tan naturales que da Perlin para que siguieran una forma concreta.

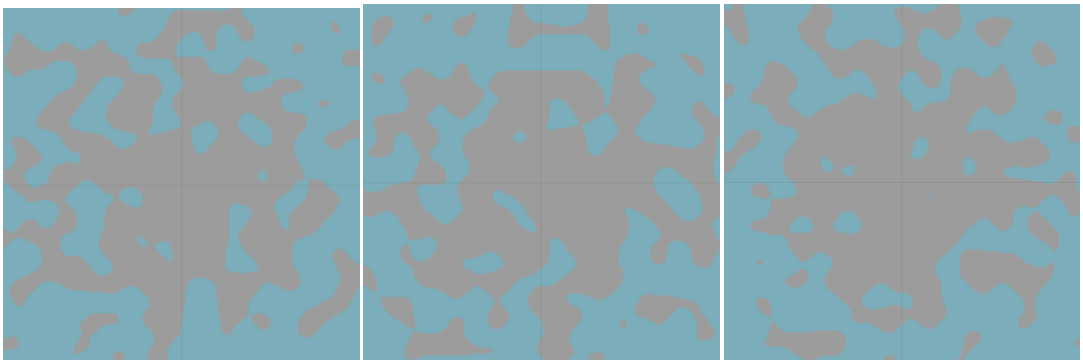
Ya que queremos una isla grande y centrada, vamos a usar un degradado que vaya del centro hacia afuera, del estilo del siguiente:



```
int centerX = (int)width / 2;
int centerY = (int)height / 2;

for (int x = 0; x < width; x++)
{
    for (int y = 0; y < height; y++)
    {

        map[x, y] = new Tile(x, y);

        float distanceX = (centerX - x) * (centerX - x);
        float distanceY = (centerY - y) * (centerY - y);

        float distanceToCenter = Mathf.Sqrt(distanceX + distanceY);
```



Ahora que tenemos el degradado y la función de Perlin, vamos a mezclarlos, a ver qué obtenemos:



Esto ya tiene mucho mejor aspecto. Al mezclar el degradado en forma de círculo con los valores que nos ofrece Perlin, conseguimos un gran pedazo de tierra con bordes irregulares, pero con forma ligeramente circular. Además, conseguimos pedacitos de tierra extra alrededor, que forman pequeñas islas. Justo lo que necesitábamos.

Esta generación es fácilmente ajustable. Si queremos más agua, podemos aumentar el umbral por el que se elige tierra. También podríamos cambiar el degradado para darle formas totalmente diferentes a la isla. Incluso podríamos darle más peso al valor del degradado que al de Perlin antes de mezclarlos, haciendo que las islas sean más regulares y menos aleatorias.

7.2.2. Perlin y el zoom

Perlin Noise se puede usar a varios niveles de detalle, haciendo más zoom o menos zoom una vez hayamos obtenido la matriz de valores.

En el caso de un juego que necesite tener en pantalla grandes mapas (como en la saga Total War):

Si el terreno es generado de manera procedimental, puede que queramos trabajar el terreno a diferentes tipos de detalle, cuanto más nos acerquemos, queremos que se vean más detalles, y cuanto más nos alejemos, tenemos el riesgo de que el costo computacional de generar y renderizar se dispare y produzca una caída en el número de fotogramas por segundo.

En ese caso, se puede modificar la función de perlin para que funcione a diferentes frecuencias. De esta manera, podríamos reproducir el mismo patrón a un nivel de detalle mayor o menor, según se necesite.

En el caso del prototipo del proyecto, no se usa LOD (nivel de detalle) ni hace falta modificar perlin para que genere a frecuencias diferentes, ya que la isla se renderiza desde bastante cerca y tampoco ofrece muchos detalles que puedan consumir nuestro tiempo de CPU.

7.2.3. Segunda parte: la elevación del terreno

Ya hemos conseguido generar islas con una forma que nos complace. Todos los tiles de la matriz tienen ahora mismo uno de dos valores, azul(agua) y gris(tierra). Lo siguiente que tenemos que pensar es en cómo crear diferentes tipos de terreno dentro de la isla (y en las islas pequeñitas).

Después de mucho investigar, llegué a la conclusión de que hay muchísimas formas de generar elevación del terreno, pero que todas empiezan por el mismo paso: distinguir el agua profunda (océano) de la que no lo es.

Distinguiendo tipos de agua

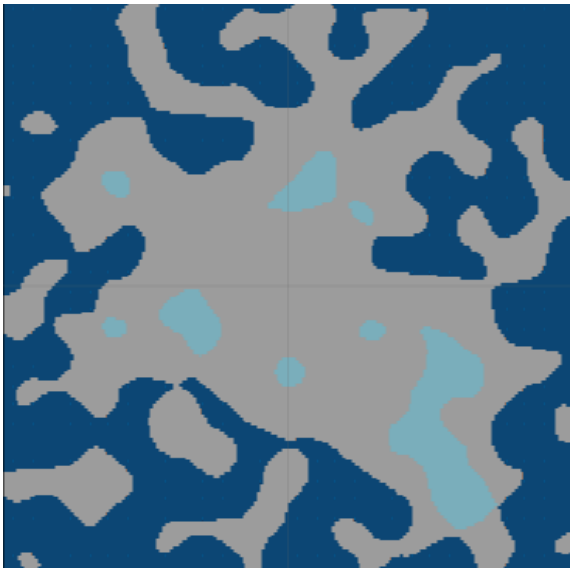
Viendo los mapas generados, distinguimos dos tipos de agua, interior (lagos) y océano. Para de verdad empezar a trabajar con las elevaciones, tendremos primero que distinguir estos dos tipos de agua de manera apropiada.

Para facilitar el procedimiento, creo un marco de tiles negros alrededor del mapa. Luego, añado a la clase Tile la propiedad "Ocean", un bool que nos indica si ese tile es de océano o no. Entonces, marco todos los tiles negros como océanos, y ejecuto lo siguiente para cada tile del mapa:

- Si el tile es un tile de agua y además está junto a un tile de océano, entonces este tile de agua debe de ser océano también.

Una sola pasada no es suficiente, claro. Debido a que el terreno es bastante irregular por algunos lados, muchos tiles que deberían ser océano no lo son, ya que sus vecinos aún no han sido pintados como tal.

Para arreglarlo, ejecuto el mismo procedimiento empezando en cada una de las cuatro esquinas, y cambiando el orden de los bucles de arriba-abajo y de izquierda-derecha.



Observaciones acerca de los lagos

En el prototipo final no se usan lagos y ríos en la generación de la isla. Éstos son tapados antes de comenzar con el proceso de creación de elevaciones. La razón fue que estaban complicando demasiado los cálculos, y presentaban algunos problemas respecto al diseño original del juego. Muchas veces, los lagos y los ríos encerraban y complicaban el acceso a diferentes puntos clave del mapa. Además, los lagos hacían bastante más complejo la asignación de elevaciones.

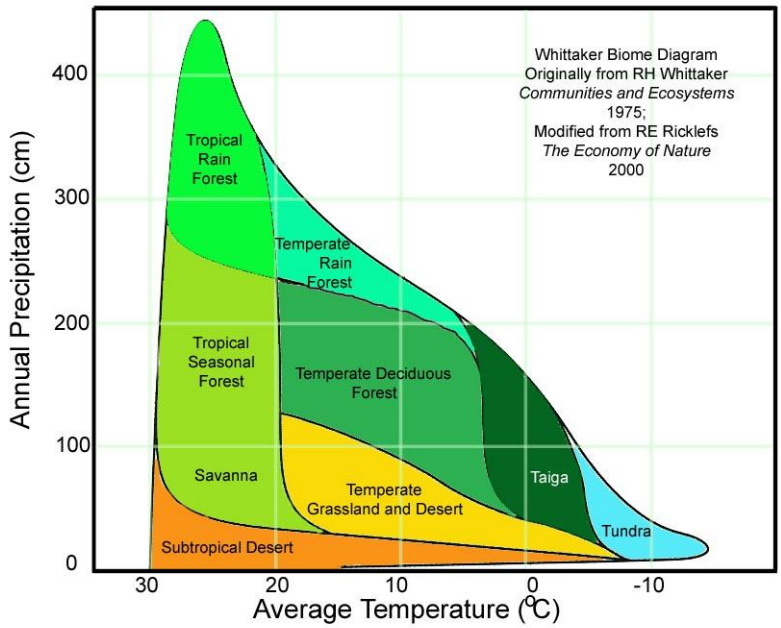
Al terminar el proyecto, una vez estudiado y analizado mis decisiones, llegué a la conclusión de que la mejor manera de crear estos accidentes geográficos era después y no antes de la asignación de elevaciones.

Creando las elevaciones

Intento 1: Usando tablas de humedad/altitud

Esta forma de crear elevaciones/tipos de terreno es muy común en videojuegos con generación procedimental.

Se basa en usar una versión personalizada de un diagrama de Whittaker:



Por ejemplo, en el artículo sobre generación de terreno con mapas poligonales, su autor calcula la humedad y elevación de cada polígono según su distancia a la costa y a los ríos.

Luego, crea su propia diagrama de Whittaker, con valores enteros:

Elevation Zone	Moisture Zone					
	6 (wet)	5	4	3	2	1 (dry)
4 (high)	SNOW			TUNDRA	BARE	SCORCHED
3	TAIGA		SHRUBLAND		TEMPERATE DESERT	
2	TEMPERATE RAIN FOREST	TEMPERATE DECIDUOUS FOREST		GRASSLAND		TEMPERATE DESERT
1 (low)	TROPICAL RAIN FOREST		TROPICAL SEASONAL FOREST		GRASSLAND	SUBTROPICAL DESERT

Finalmente, hace una pasada por todo el mapa, y por cada polígono, obtiene su nivel de humedad y elevación, y le asigna el terreno correspondiente según la tabla.

Para el proyecto, me creé mi propia tabla de humedad/elevación, luego asigné valores a cada uno de mis tiles:

- La elevación la extraía de una nueva muestra de Perlin Noise.
- La humedad venía dictada por la posición Y del tile, en relación con el mapa completo.

Altura\Latitud(humedad)	6	5	4	3	2	1
4	Nieve	Nieve	Nieve	Montaña	Montaña	Montaña
3	Nieve	Montaña	Montaña	Bosque	Llanura	Desierto
2	Montaña	Bosque	Bosque	Llanura	Desierto	Desierto
1	Bosque	Bosque	Llanura	Llanura	Desierto	Desierto

Cambié algunos números aquí y allí, pero el resultado no terminaba de satisfacerme. No salía natural, y las muestras de Perlin Noise eran demasiado aleatorias.

Intento 2: Usando la distancia a la costa

He de decir que quedé un poco decepcionado con las tablas de humedad/altitud. Estaba bastante confiado de que las tablas iban a funcionar a la perfección, pero estaba claro que, de la forma en que lo estaba intentando, no iba a funcionar. No me quedaba más remedio que pensar en otra forma de asignar los distintos tipos de terreno.

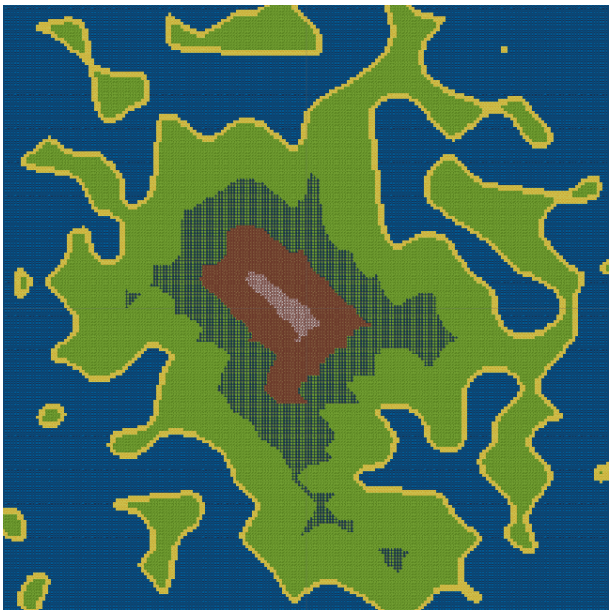
Entonces me hice la pregunta que me resolvió el problema:

Cuanto más te alejas de la costa, ¿cómo se va volviendo el terreno?

Por lo general, cerca de la costa el terreno suele ser llano, o de poca altitud, y seco. A medida que te vas alejando, el terreno suele aumentar en altitud y se va volviendo cada vez más húmedo.

Entonces, empezando desde la costa, fui apuntando diferentes tipos de terreno:

- Justo en la costa, pegados a los tiles de océano, hay playas. Es decir, arena.
- Un poco más adentro, necesitamos un terreno un poco más húmedo pero que no suba mucho en elevación. Decidimos usar un terreno genérico de llanura.
- La siguiente elección debería ser un terreno bastante más húmedo. Nos estamos acercando al centro, a la cumbre que corona la isla. Un bosque es perfecto, es un terreno húmedo y queda muy bien junto a la montaña.
- El resto de tiles que quedan se cubren con montaña, adornando unos cuantos de éstos con nieve.



Al final, el enfoque más simple fue el más efectivo. Gracias a la generación de la forma de la isla con perlin, hay diferentes zonas con más o menos altitud, ya que la distancia a la costa cambia muy a menudo, gracias a los bordes irregulares. Y tenemos lo que queríamos, incluida una montaña con nieve en el centro de la isla.

7.3. Clases generadas

- **PerlinTerrain**: clase principal que genera la isla completa paso a paso. Toma un tamaño para la isla y tiles para cada tipo de terreno y crea el mapa entero. Cada paso está implementado en una función independiente con diferentes variables como

  ```
  void SetTerrainTypeByElevation(Tile.Type terrainType, int elevation)...
  ```

  que permite asignar un tipo de terreno a partir de una elevación (distancia a la costa) dada.

Crearemos un prefab llamado "TerrainGenerator" al que le añadiremos como componente la clase "PerlinTerrain" para poder poner las variables de entrada y objetos utilizados (tiles) y guardarlo para instanciarlo en cualquier momento.

7.4. Observaciones finales

La generación procedimental es como una pieza de arte, es difícil darla por terminada. En este caso, puedes seguir ajustando números, mejorando las funciones y probar, probar y probar durante semanas. El aspecto general mejoraría mucho, pero aún así, seguro que no estarías completamente satisfecho, y habría más cosas que podrían mejorar u optimizarse.

Me encantaría seguir trabajando aún más la generación de la isla pero, tal como está ahora, el resultado es bastante cercano al objetivo que me había marcado, y me permite continuar con el resto de cosas a las que me dedicaré en este proyecto.

De todas maneras, en el capítulo 9 "Conclusiones y trabajos futuros" se indican algunas mejoras que podrían hacerse a la generación procedimental de la isla, que perfeccionarían aún más la forma, elevación y aspecto general, así como reducir la posibilidad de que la isla generada no sea usable por el juego.

8. Desarrollo - Módulo 2: BSP

Hojeando roguebasin.com, me encontré con un artículo muy interesante que hablaba sobre la generación de mazmorras aleatorias usando un algoritmo llamado BSP (binary space partition). A simple vista, parecía un algoritmo sencillo pero flexible, así que nos lanzamos a ello.

8.1. Qué es BSP

El BSP (binary space partition) es un método para dividir recursivamente un espacio, pudiendo representar la estructura del mismo en forma de árbol.

Los árboles BSP se suelen usar en videojuegos para optimizar el renderizado, trazado de rayos y detección de colisiones.

Curiosamente, en el caso de este proyecto, vamos a usar el BSP para algo completamente distinto: crear mapas de mazmorras. La simplicidad del algoritmo junto con su facilidad para controlarlo en medio de la ejecución lo transforman en el candidato perfecto para esta tarea.

8.2. Primera parte: creando el árbol

Nota: En el primer intento de crear el árbol BSP, escribí un algoritmo que se quedaba sólo con los nodos hoja, que eran los que me interesaban para pintar luego las habitaciones de las mazmorras. Evidentemente, eso fue un craso error, ya que necesito el árbol al completo para poder emparentar las diferentes habitaciones y asegurarme de que se puede llegar a todas y cada una de ellas.

La raíz del árbol sería una habitación del tamaño del que queremos hacer la mazmorra. Así, al aplicar luego el BSP, las diferentes habitaciones que nos van saliendo, quedan enmarcadas dentro de esa "habitación madre", haciéndonos muy fácil colocarlas en el espacio de juego.

Creamos una clase "Room" con las siguientes variables:

```
public class Room
{
    public Vector3 position;
    public int width;
    public int height;
    public int id;

    public List<Room> children;
```

En principio, eso es todo lo que necesitamos para crear el árbol.

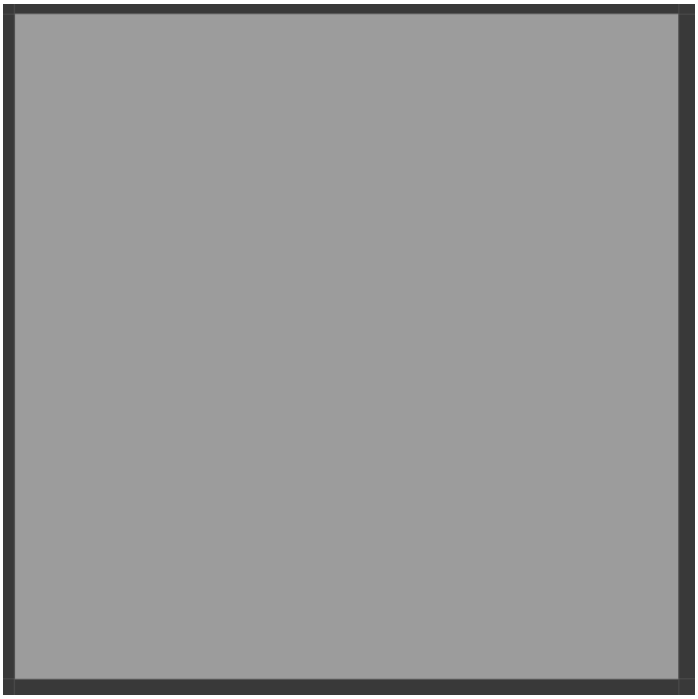
Creamos el algoritmo BSP y le permitimos indicar cuántas veces queremos partir en dos las habitaciones de la mazmorra y qué porcentajes mínimo y máximo usaremos para el corte.

Como dato, en el prototipo final usé tres iteraciones (8 habitaciones) y un corte del 40%-70% de la habitación inicial.

8.3. Segunda parte: instanciando las habitaciones

Para instanciar los tiles de las habitaciones usaremos el mismo método de los prefabs que usamos en la instanciación de la isla. Los tiles de las habitaciones serían tiles grises y el resto, tile negros.

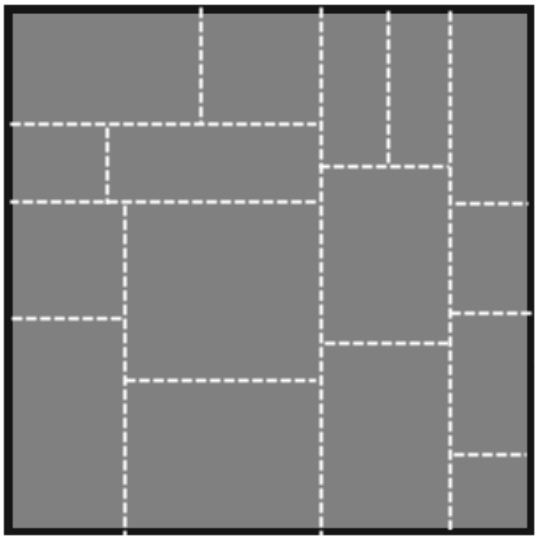
La primera vez que instanciamos la mazmorra nos quedó algo así:



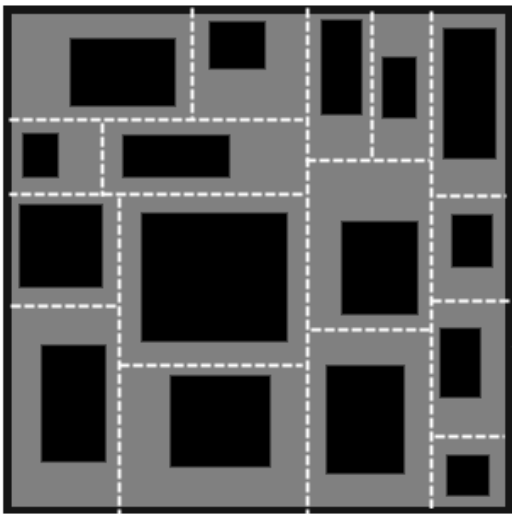
Evidentemente, algo estaba fallando. Repasé el código del BSP varias veces, comprobé la instanciación de los tiles, cambié la configuración inicial de la habitación raíz... Llegué a pintar cada habitación de un color diferente y ahí fue donde me di cuenta: las habitaciones estaban llenando el 100% de su tamaño con tiles, por lo que no hay nada de espacio entre una habitación y otra.

Además de ser un inconveniente, las habitaciones generadas son todas muy parecidas y la mazmorra queda muy aburrida. Volví a visitar roguebasin.com y efectivamente, me salté el paso final.

Así es como deberían quedar las habitaciones después de pasar todas las iteraciones del BSP:



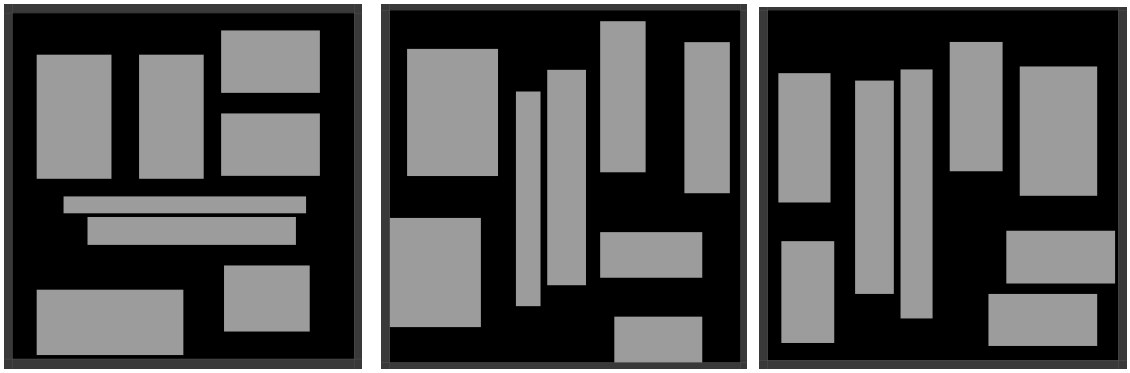
Y así, si haces que cada habitación pinte sus tiles en un espacio más pequeño dentro de su tamaño total:



Esto añade aún más posibilidades a la generación de la mazmorra. Puedes modificar los porcentajes mínimo y máximo de dónde se empieza y dónde se acaba de pintar la habitación, tanto para su anchura como para su altura.

Nota: en el prototipo final, reduzco las habitaciones al 60%-80% tanto de su anchura como de su altura.

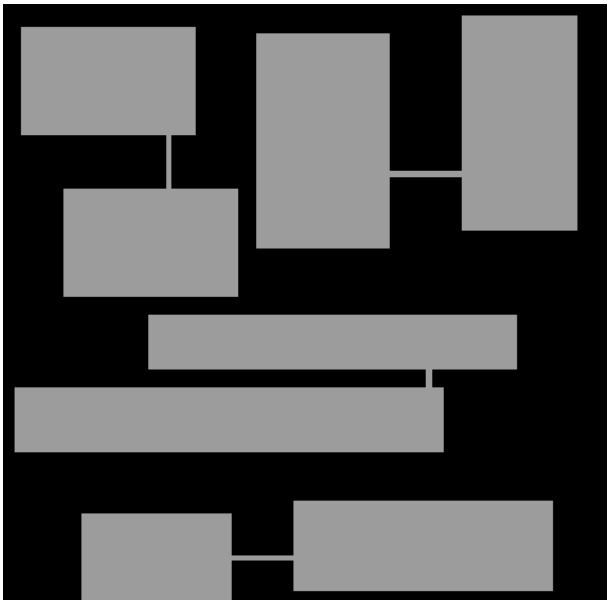
Una vez escrito el algoritmo para empequeñecer las habitaciones, así es como quedó finalmente la mazmorra:



8.4. Tercera parte: creando los pasillos

Después de varios intentos fallidos (unir las dos caras más cercanas de cada habitación, elegir dos puntos de manera aleatoria y unirlos con una línea recta) decidí que la mejor manera era coger el punto medio de cada habitación, y unirlos mediante A*. Así me evito problemas, siempre van a encontrar el camino y no se va a solapar con nada.

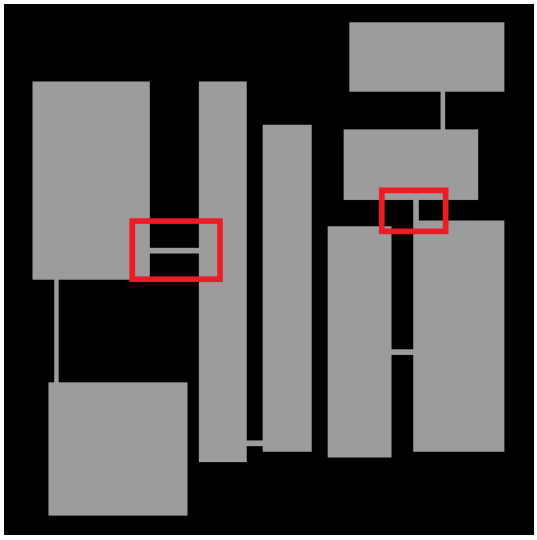
Para elegir las habitaciones que se juntan mediante pasillos usé un pequeño truco: recojo todas las habitaciones hoja y las introduzco en una nueva lista. Como esa lista está ordenada, las habitaciones en números pares tienen a su hermana en la siguiente posición (hab[0] tiene su hermana en hab[1]). Así que recorriendo esa lista de dos en dos, puedo unir sin problema a todas las habitaciones hermanas.



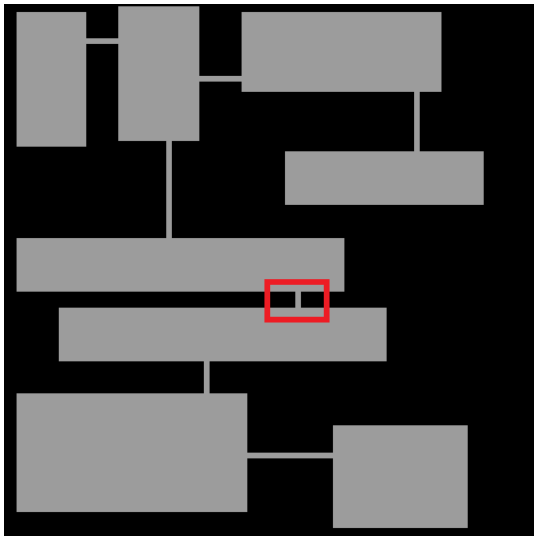
Hacer el resto de pasillos entre grupos de habitaciones es un poco más complejo. Primero identificas el grupo de donde sales y el grupo hacia donde vas siguiendo el siguiente proceso:

- Identificas el grupo de donde vas a salir, y te quedas con la primera habitación de ese grupo en la lista.
- El tamaño de cada grupo es 2^n donde n es la iteración en el procedimiento de crear pasillos (empezando por 0).
- Los grupos están adyacentes en la lista ordenada. A partir de la primera habitación, y contándose a sí misma, cuentas 2^n y tienes el primer grupo. Luego cuentas 2^n otra vez, y ya tienes el segundo grupo.
- Introduces las habitaciones en dos listas nuevas. Luego comparas ambas listas y te quedas con las dos habitaciones que estén más cercana la una de la otra.
- Creas un pasillo entre esas dos habitaciones.

Segunda iteración:



Tercera iteración:



Notas acerca de los pasillos:

Para los pasillos escribí una nueva clase, "Corridor" que guarda la información de las habitaciones que une y de los tiles de que se compone. Luego al crear la lógica del juego, hará falta que la mazmorra se exploren poco a poco, y tener los pasillos y las habitaciones ordenadas y cada cual sabiendo a quién está unida, me facilitará mucho las cosas.

8.5. Clases Generadas

- **BSPDungeonGenerator**: crea una mazmorra paso a paso. Cada paso está separado en una función independiente. Guarda toda la información de la misma en un árbol BSP, en el que contamos desde la habitación raíz hasta las hojas, que son las que realmente se ven luego en la escena. También guarda por separado las habitaciones hoja, para poder hacer cálculos y usar funciones con ellas rápidamente, tales como buscar habitaciones contiguas, hacer pasillos o reducir sus tamaños finales.
- **AStar**: solución personalizada de A*. Como también la usaremos para ajustes en el mapa y para el movimiento del jugador, hemos decidido crearnos una desde cero.

Tal como hicimos con el generador de la isla, crearemos un prefab llamado "DungeonGenerator" al que le añadiremos como componente la clase "BSPDungeonGenerator" para poder poner los objetos utilizados (tiles) y guardarlo e instanciarlo en cualquier momento.

9. Desarrollo - Módulo 3: El prototipo del juego

Tenemos un generador de islas que nos brinda un mapa extenso y que nos asegura que va a haber una serie de tipos de terreno, agua y algunas islas. Tenemos también un generador de mazmorras que nos asegura que son resolubles. Es hora de empezar con la lógica del juego. Vamos a unir todo lo que tenemos en el prototipo de lo que sería un juego de aventuras en el que todo está generado procedimentalmente, no habiendo casi nada "colocado a mano".

La perspectiva de nuestro juego es top-down, es decir, 90º de inclinación en la cámara respecto a la superficie del terreno. Todo es representado mediante iconos cuadrados, los tiles.

Según diseño, el juego debería poder jugarse únicamente con un ratón. Esto es un punto interesante, porque significa que en caso de que el juego quisiera portarse a alguna plataforma móvil, lo podría hacer sin demasiados problemas. Todas las acciones serían toques de dedo, arrastrar o cosas parecidas que haríamos con el ratón.

9.1. El mapa

El flujo de juego es el siguiente: el jugador accede a la primera ciudad y se le revela la situación de la primera mazmorra. Completa la primera mazmorra y gana un objeto que le permite desbloquear la siguiente ciudad y la siguiente mazmorra. A medida que va avanzando, debe ir comprando objetos que le permitan desplazarse por zonas del mapa que normalmente no podría (como son las montañas o el agua) y así poder llegar hasta la última mazmorra y completarla.

En principio, el juego tiene un total de 4 ciudades y 4 mazmorras, un par en las llanuras, otro en el bosque, el tercero en las montañas y el cuarto en una de las pequeñas islas que rodean la isla principal donde se encuentra el jugador.

Al cargar un nuevo juego, generamos una mapa con nuestro generador de islas, luego creamos 4 ciudades y las colocamos en un punto aleatorio de entre todos los tiles del terreno donde deberían estar. Luego usamos nuestro generador de mazmorras para crear 4 mazmorras y las guardamos. Para cada una de ellas, creamos una entrada que colocaremos en el mapa de manera similar a como hicimos con las ciudades.

Para facilitar el acceso y uso de estas entidades, las ciudades y mazmorras se llaman de la siguiente manera:

- Grassland Town/Dungeon
- Forest Town/Dungeon
- Mountain Town/Dungeon
- Island Town/Dungeon

Además, las ciudades se colocan como hijas de un GameObject vacío llamado "Towns", y las mazmorras en otro llamado "Dungeons".

Sobra decir que el controlador principal del juego, el GameDirector guarda la información de todas estas entidades, para poder cargar niveles e instanciarlas a placer.

9.2. El jugador

Según diseño, el jugador controla un grupo de tres aventureros que viajan juntos. Llevan una mochila donde guardan los objetos y el oro que se van encontrando y cada uno de ellos tiene una puntuación concreta en cada una de sus estadísticas.

Además, pueden desplazarse por la isla y entrar en ciudades y mazmorras. En el caso de las mazmorras, también pueden explorarlas tal y como se desplazan por la isla.

El control más interesante para este tipo de requisitos es moverse a la posición donde se hace click con el ratón. Gracias a nuestra implementación de A*, podemos inyectar el/los tipos de terreno que el jugador no puede atravesar y ya tenemos un movimiento suave y automático hacia donde el jugador haga click.

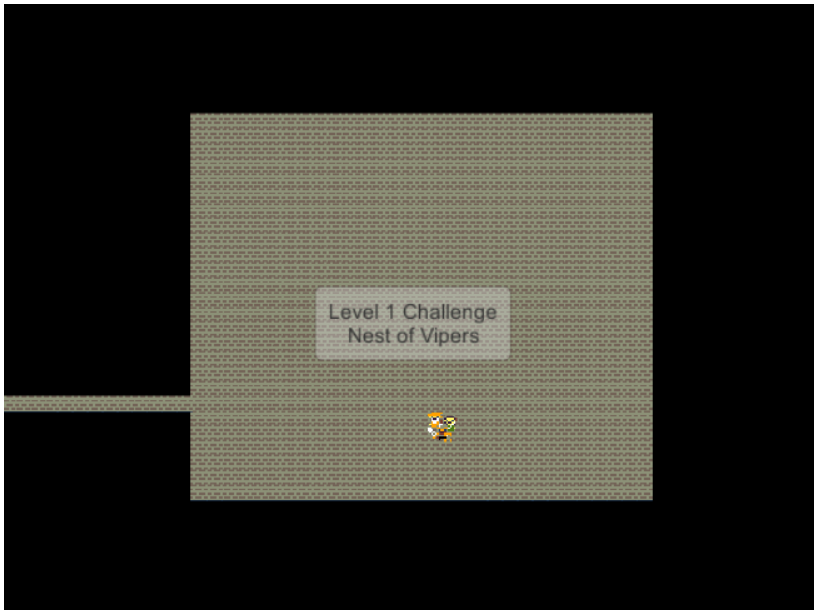
A parte del movimiento, el jugador puede interactuar con el juego y realizar el resto de acciones mediante las interfaces que se le van presentando.

9.3. Las mazmorras y los retos

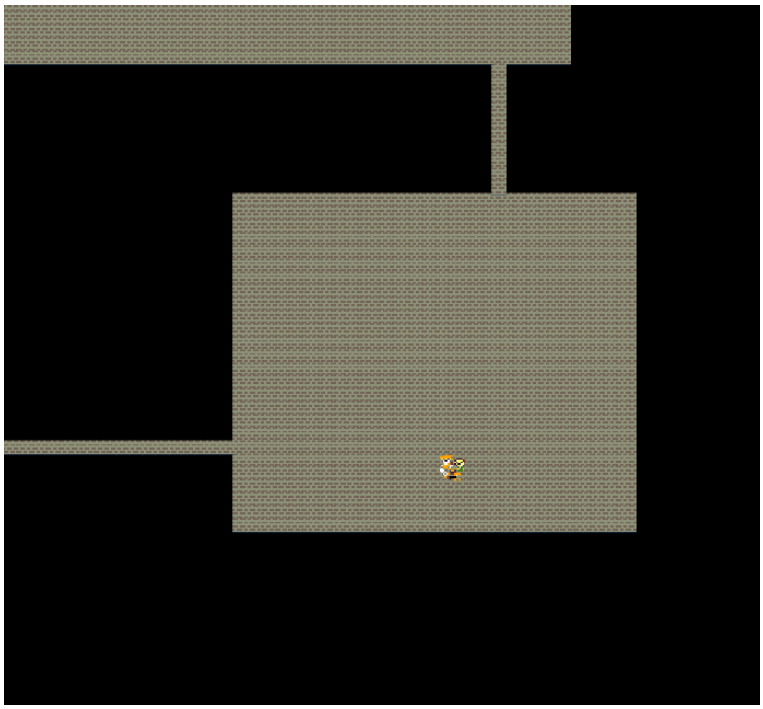
Habíamos dicho que para cada mazmorra, hemos creado una entrada y las hemos colocado alrededor del mapa. Cuando el jugador se mueve y se para en la entrada de una mazmorra, entra dentro de ella. El juego carga un nuevo nivel vacío, y el GameDirector instancia entonces la mazmorra que corresponda. El jugador entonces aparece en la primera habitación de esa mazmorra, junto a la puerta de entrada/salida.

La idea es que en cada habitación (salvo la primera) haya un reto que el grupo de aventureros tenga que completar para poder abrir las puertas de esa habitación y poder avanzar por la mazmorra.

Sala con un reto aún no completado:



La misma sala, ya con el reto completado y el camino abierto:



Creamos la clase "Challenge", que llevará la siguiente información:

```
public class Challenge
{
    public string name;
    public Attribute attribute;
    public int difficulty;

    public List<Item> loot;

    public bool challengeCompleted;

    public enum Attribute
    {
        Combat, Mechanics, Deduction, Wisdom, Mercantile
    }
```

Debido a la naturaleza compleja de los retos (nivel de dificultad, tipo, recompensa), usaremos una clase para instanciar diferentes tipos de retos y guardarlos en una colección a la que podremos acceder cuando necesitamos asignar un nuevo reto a una habitación. Los guardaremos ordenados por niveles de dificultad.

Gracias a esto, podemos asegurar un buen balance entre dificultad y recompensa obtenida. Además, podemos hacer pequeñas variaciones en la cantidad/tipo de recompensa que se obtiene de cada reto usando selecciones aleatorias de números entre un rango concreto. De esta manera, cuando un jugador completa un reto, siempre recibirá una recompensa aleatoria, pero adecuada a la dificultad del mismo.

El jugador puede abandonar en cualquier momento la mazmorra saliendo por la puerta de entrada. El juego guardará el estado de la misma, qué habitaciones se han descubierto ya, y qué retos aún no han sido completados.

9.4. Interfaces

9.4.1. Interfaz del grupo

En la interfaz del grupo o *party* queremos ver información acerca de los aventureros que lo conforman. Además, queremos ver el contenido de nuestra mochila, para saber cuánto oro tenemos y qué objetos estamos llevando actualmente.

9.4.2. Interfaces de una mazmorra

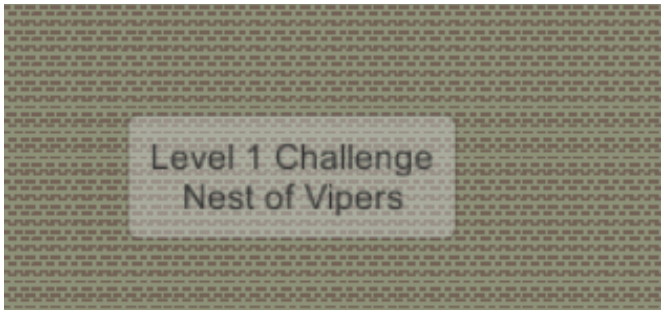
En cada habitación de una mazmorra (exceptuando la primera) hay un reto. La idea es que el jugador pueda moverse libremente por las habitaciones, pero hasta que no complete el reto de una habitación, ésta no se abrirá. Esto requiere, por un lado, instanciar paneles en el espacio de juego (en el centro de cada habitación), y por otro lado, una serie de paneles que se muestran y se ocultan según lo que vaya haciendo el jugador.

La manera más eficaz de hacer esto, es crear prefabs de cada uno de los tipos de interfaz que se van a usar y, para cada uno de ellos, crear un controlador al que se le inyecta los datos necesarios. La ventaja principal de esta manera de crear interfaces es que cada uno de esos paneles son independientes, sólo hay que instanciarlos y ellos solos se rellenan según la información que se le haya inyectado automáticamente.

Panel pequeño de habitación

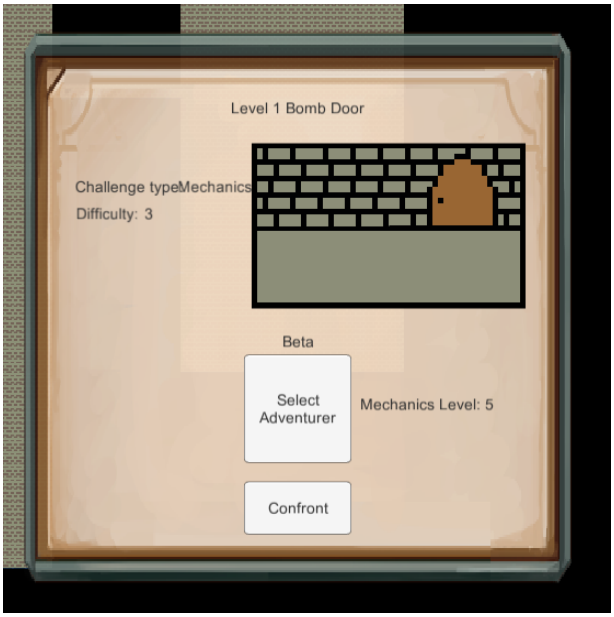
A diferencia del resto de interfaces del juego, ésta es una interfaz que se instancia dentro del espacio de juego. La razón es simplemente que éstos paneles tienen que encontrarse en el centro de la habitación al que pertenecen.

Muestran suficiente información como para que el jugador sepa qué es lo que se encuentra dentro de ese reto, y más o menos el nivel de dificultad del mismo.



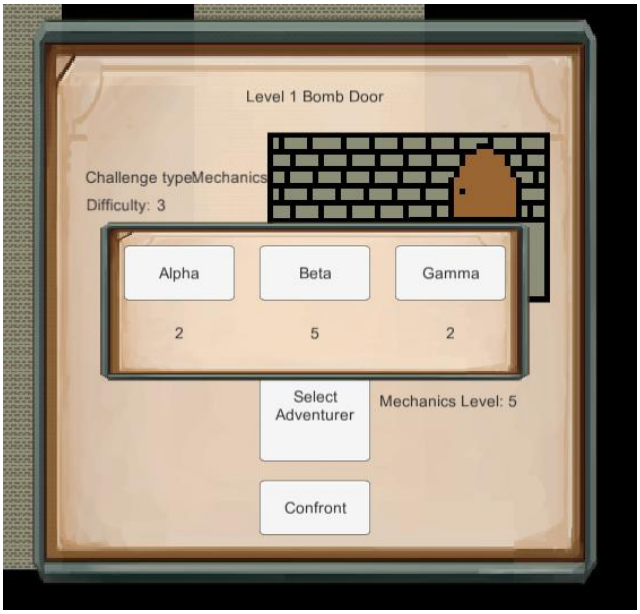
Panel grande de habitación

Al pasar el ratón por encima de un panel pequeño de habitación, aparece su versión completa, el panel grande de habitación. Este panel muestra la información completa acerca del reto al que pertenece. Además, desde aquí puede seleccionarse el aventurero que intentará superar el reto.



El jugador leerá detenidamente la información que se le mostrará y tendrá dos opciones:

- Elegir un aventurero que se enfrentará al reto.
- Evitar el reto por el momento.



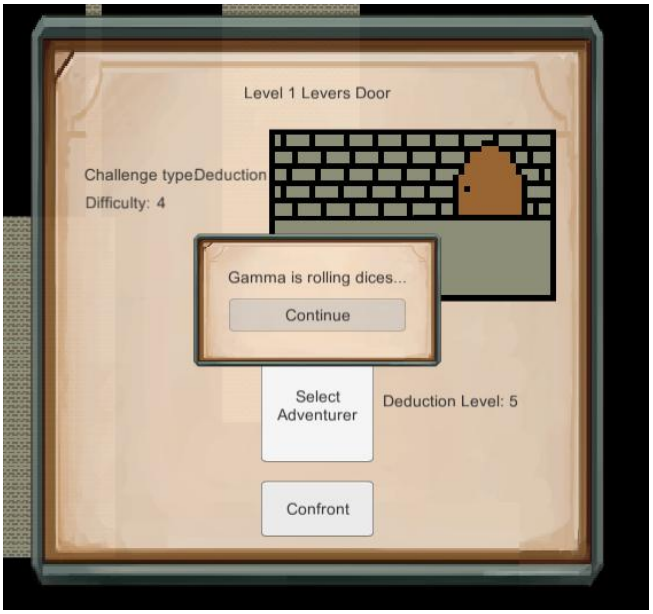
Si el jugador saca el ratón del panel grande, éste se cerrará, y se mostrará de nuevo el panel pequeño de la habitación. Luego, el jugador podrá irse a otro lado si lo desea.

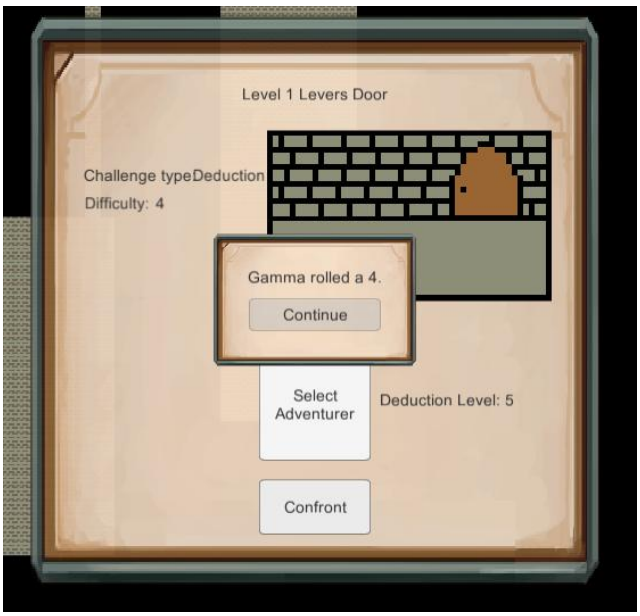
Si el jugador quiere enfrentarse al reto, sólo tiene que darle a botón "Confront" una vez esté satisfecho con su selección de aventurero.

Panel de enfrentamiento

Cuando el jugador hace click en el botón Confront, éste panel se pone en funcionamiento.

Básicamente, aquí el jugador podrá ver como su aventurero tira un dado, suma su atributo al resultado de ese dado, y luego se compara con la dificultad del reto.





Si su resultado es mayor que el del reto, lo habrá superado. El grupo recibirá una recompensa, y la habitación se abrirá, pudiendo mostrar el camino hacia nuevas habitaciones.

Panel de nueva reliquia

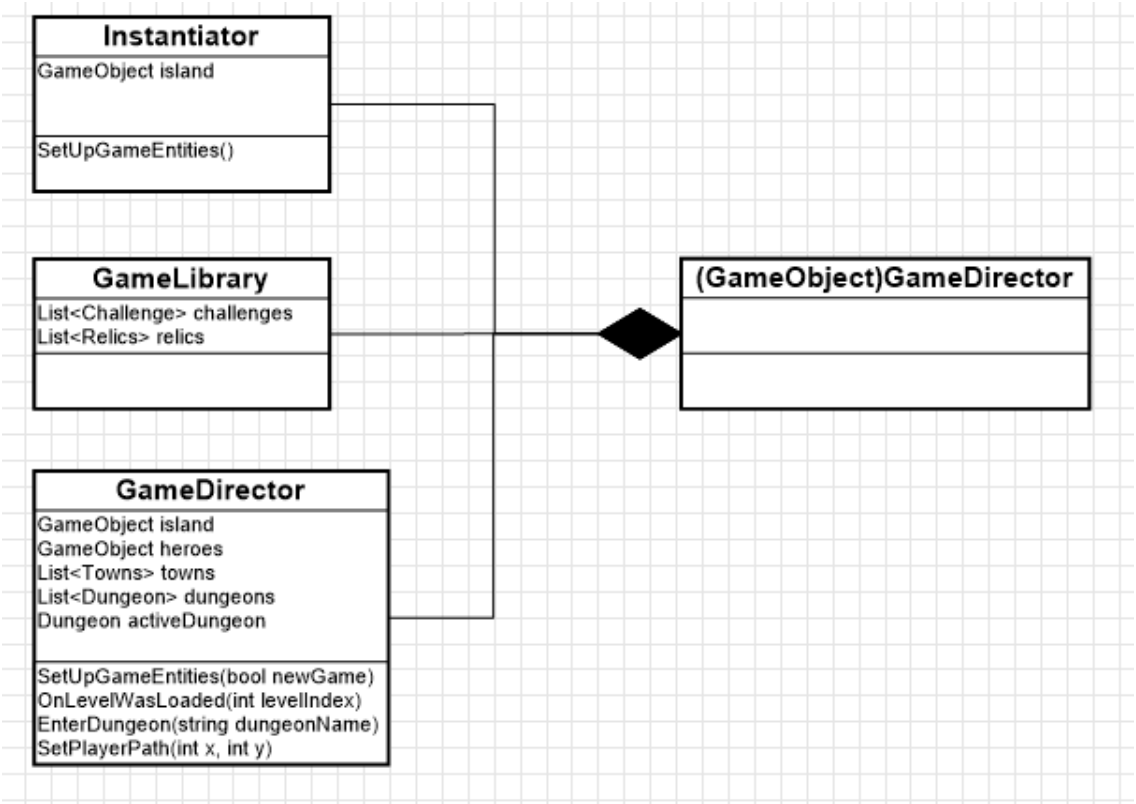
Si entre la recompensa de un reto se encuentra una reliquia, el juego avisa al jugador de que efectivamente ha conseguido una nueva reliquia. Además, el juego indica que una nueva mazmorra ha sido desbloqueada.

9.5. Estructura de clases

9.5.1. Game Director

El controlador principal del juego, encargado de conectar todas las cosas y de mandar a instanciar, cargar y generar todos los demás objetos del juego.

- Manda a generar la isla y todas las mazmorras, luego las guarda para mandar a instanciarlas cuando sea necesario.
- Tiene acceso al instanciador del motor del juego (Instantiator class).
- Manda instanciar al jugador y a la interfaz principal.
- Tiene acceso a la librería de objetos del juego (GameLibrary).
- Gestiona los clicks del ratón para mandarle los caminos de A* al jugador para su movimiento.
- Carga el nivel principal (la isla) cuando empieza la partida. También carga los niveles de mazmorra cuando se lo pidan.



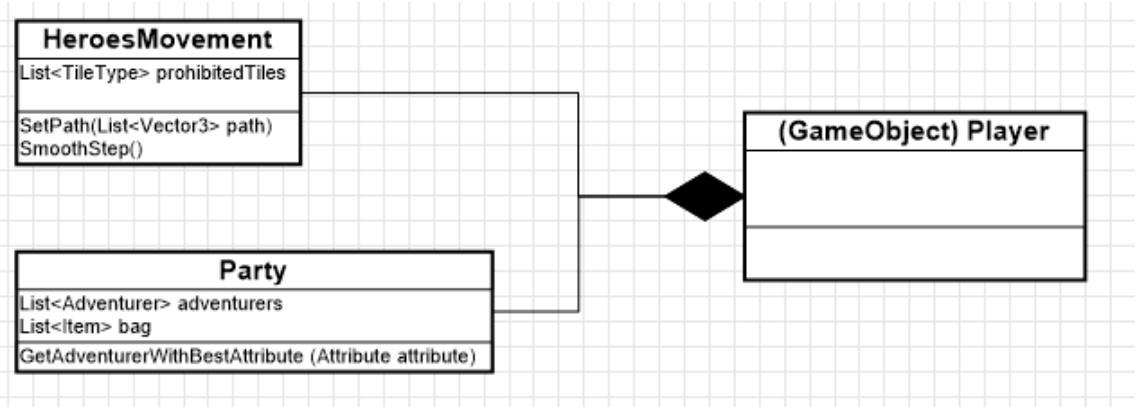
El GameObject GameDirector está instanciado desde tiempo de editor, y pone en marcha todo lo demás en su método Start al ejecutar una nueva partida.

Tiene tres componentes, a los que tiene acceso en todo momento, la clase GameDirector que se encarga de lo mencionado anteriormente, la librería de objetos y el instanciador.

9.5.2. Player

El jugador se forma de dos componentes:

- El componente de movimiento se encarga de recibir trayectorias del GameDirector y de aplicarlas al avatar.
- El componente del grupo (party) tiene toda la información lógica del propio jugador, los aventureros que lo componen y los objetos que llevan.



El resto de objetos, una vez instanciados, funcionan por su cuenta.

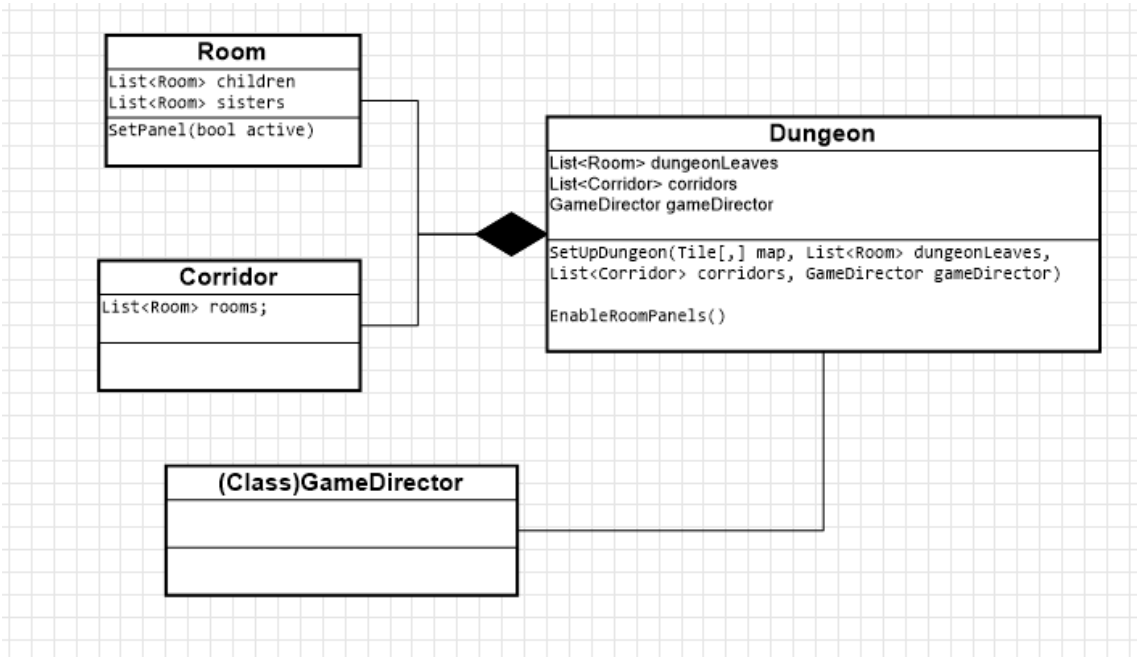
En el caso de las habitaciones de una mazmorra (cuando estamos dentro de una), se encargan de mandar a instanciar todas las interfaces correspondientes. Estas interfaces están preparadas como prefabs, y actualizan la información que muestran automáticamente después de ser instanciadas.

9.5.3. Dungeon

Una mazmorra se compone de una serie de habitaciones, las habitaciones hoja de un árbol de BSPDungeonGenerator. También tiene una lista de pasillos que unen todas esas habitaciones.

El GameDirector es quien crea las mazmorras, pasándole por parámetro tanto el mapa de tiles, como la lista de habitaciones y de pasillos.

En el momento que sea necesario, la mazmorra activará los paneles de sus habitaciones. Una vez instanciados, estos paneles funcionan autónomamente.



9.5.4. El generador de la isla y las mazmorras

Tal como vimos en sus apartados correspondientes, tanto el generador de la isla como el de las mazmorras tienen sus propios prefabs.

Éstos prefabs son GameObjects que el GameDirector instancia cuando son necesarios.

Al comienzo del juego se instancia el de la isla, para crearla y guardarla. Luego la isla se instancia cuando se carga el nivel correspondiente.

Para cada mazmorra, usamos una instancia del generador de mazmorras, que nos devuelve todos los datos que necesitamos para crearlas. Esta información la usamos para crear instancias de Dungeon, como vimos en el apartado anterior.

Cuando se carga un nivel de mazmorra, se accede a la mazmorra indicada y se instancia su mapa.

10. Conclusiones y trabajos futuros

Ha sido una experiencia increíble trabajar con técnicas de generación procedimental.

Crear un videojuego es un reto muy duro en el que se aprenden un montón de cosas. Nunca tienes muy claro cuál es el siguiente paso que tienes que dar, y eso hace que tengas que renovar continuamente tu forma de pensar o de resolver los problemas.

Pero crear un juego usando técnicas y métodos de generación procedimental es mucho más duro de lo que podría parecer en un principio. Meses atrás no me habría creído capaz de sacar adelante un proyecto en el que tantas cosas se crean sobre la marcha, y se conectan unas con otras para entregar una experiencia de juego.

Una de las cosas más interesantes de éste tipo de métodos es que, casi siempre, puedes mezclar unos con otros. Puedes usar diferentes tipos de ruido para obtener una serie de valores, que luego puedes modificar haciendo medias ponderadas personalizadas con otros tipos de ruidos, o con texturas pre-generadas por ti. Puedes usar técnicas propias de IA para trazar caminos, comprobar trayectorias y muchas otras cosas, como puede ser crear pasillos, ríos y elementos parecidos.

10.1. Mejoras para la generación de la isla

- Limpiar restos de terreno en medio de otros tipos. Si por ejemplo se queda un tile de tierra en medio de muchos de agua, con esta pasada lo eliminaríamos. Este procedimiento también elimina algunos dientes de sierra en los bordes entre terrenos, haciendo los bordes más "suaves".
- Normalizar la matriz de perlin antes de empezar a trabajar. Si normalizamos la matriz de perlin podemos asegurar que encontraremos valores del 0 al 1. Esto posiblemente evitaría que en una isla no se genere nieve, por ejemplo, ya que puede ser que el pedazo de matriz que hemos usado no tiene valores más altos que 0.8 por ejemplo.

10.2. Cosas que se quedaron atrás

10.2.1. Las ciudades

- Estaban planteadas en diseño como lugares donde comprar y vender.
- Principalmente se usan para avanzar en el juego, comprando objetos (como el kit de montaña o la barca) que permiten al jugador caminar por terrenos que no podría de otra manera.

10.2.2. Los retos grupales

- En diseño se planteó la posibilidad de hacer retos grupales, en el que todos los aventureros juntos intentaran completar el reto planteado.

10.2.3. Los objetos equipables

- También se había pensado en implementar una serie de objetos que se podrían equipar a los aventureros, para añadirle una serie de mejoras a sus estadísticas, que

añadieran dados mejores a sus tiradas (todos los aventureros empieza con un dado de cuatro caras) o que mejoraran otro aspecto del grupo.

10.3. Añadidos para el juego

10.3.1. Extensión del mapa

Con el motor actual del juego, sería relativamente sencillo ampliar la generación procedimental de la isla para transformarla en una generación de terreno "infinito", en el que, cuando el jugador se acerca a un borde del mapa, se autogenera más mapa en esa dirección. Esto da sensación de inmensidad al juego, un poco la misma experiencia que la exploración en Minecraft.

10.3.2. Ciudades explorables

Aunque las ciudades se habían pensado en forma de interfaz (ya que realmente no se necesitaba más), se podría plantear crear ciudades de forma procedimental, usando una versión modificada del BSP para mazmorras. De esta manera, el jugador podría visitar personalmente las tiendas y callejones de una ciudad, y añadiría inmersión al juego.

11. Anexo I - Documento de Diseño de Juego (GDD)

11.1. Introducción

Juego de aventuras en el que manejas a un grupo de buscadores de reliquias dispuestos a recorrer y limpiar hasta la última ruina, mazmorra o castillo que se encuentren por delante.

11.2. Ritmo de juego

Los aventureros tendrán que ir desplazándose por pueblos, para conseguir información acerca de mazmorras cercanas y abastecerse antes de cada expedición.

El jugador se moverá por un mapa atravesando bosques, llanuras, montañas, desiertos, nieve e incluso agua para llegar a el emplazamiento de las mazmorras. Una vez dentro de la mazmorra, el grupo de aventureros debe llegar hasta el final de la misma para saquear todos los tesoros que haya dentro.

Luego, volverán a la ciudad con todo el botín, con el que financiarán la siguiente expedición.

11.3. Elementos de juego

Mapa del mundo

Por donde se desplazará el jugador para ir a un pueblo o en busca de una nueva mazmorra.

Pueblos

Aquí el jugador puede buscar información acerca de nuevas mazmorras y reabastecerse antes de cada expedición.

Mazmorras

El objetivo es llegar hasta la última sala y saquearlo todo.

Objetos

Equipo

Cada uno de los aventureros va equipado con una serie de objetos que aumentan sus estadísticas y le permite avanzar por el mapa y las mazmorras.

Los diferentes objetos de equipo se crearán a partir de plantillas y se generarán de manera procedimental pudiendo, a partir de unos pocos objetos y características, sacar cientos de combinaciones y por lo tanto, cientos de objetos únicos.

Botín

Baratijas que se pueden vender en los pueblos para ganar unas monedas y así financiar más expediciones.

Reliquias

Piezas únicas que el jugador puede coleccionar. Hay una en cada mazmorra.

11.4. Héroes

El grupo se conforma de 3 aventureros. Cada aventurero tiene una puntuación determinada en cada atributo:

- Destreza en combate
- Pericia mecánica
- Capacidad de deducción
- Sabiduría de mundo
- Mercadeo

Cada atributo añade su puntuación a la tirada correspondiente, siempre que ese aventurero en cuestión la haga.

Mercadeo mejora los precios de los objetos en las ciudades.

Además, cada héroe puede llevar 3 objetos equipables.

11.5. Desafíos

Criaturas

No sólo hay polvo y piedras en una mazmorra. Las salas castillos y ruinas están repletas de criaturas dispuestas a merendarse a cualquiera que entre en su territorio.

Trampas

A veces, las salas, corredores y puertas de una mazmorra tienen artefactos mecánicos posiblemente dañinos que deben ser desactivados para poder avanzar.

Cómo superar desafíos

Cada desafío que se encuentre el jugador, ya sea en forma de criaturitas rabiosas o de trampas probablemente mortales, debe ser superado con una tirada de dados por parte del grupo de aventureros.

La suma de todos los dados de la tirada debe ser igual o superior a la dificultad del desafío. Si el grupo falla la tirada, sufrirá las consecuencias (según contra lo que se estuviera haciendo la tirada).

Las estadísticas del aventurero, así como los objetos que lleva equipados pueden ayudar a superar estas tiradas.

Hay veces que las tiradas pueden hacerse en grupo, como lo sería combatir a un nido de ratas gigantes. Pero muchas otras, las tiradas las debe hacer únicamente un miembro del grupo (como desactivar una trampa en una puerta), por lo que conviene llevar un grupo bien heterogéneo de aventureros para ser capaces de superar cualquier reto con facilidad.

12. Anexo II - Fuentes de Información

12.1. Unity y programación en general

docs.unity3d.com - API y manual de Unity fusionados en una sola página.

answers.unity3d.com - Página muy importante tanto para novatos y veteranos. La comunidad de Unity es muy vasta, siempre encontrarás respuesta a tus preguntas acerca de la programación en Unity (tanto en c# como en javascript).

stackoverflow.com - Página de preguntas y respuestas sobre programación.

http://forum.unity3d.com/

http://gamedevelopment.tutsplus.com/ - Página dedicada a videojuegos que de vez en cuando suben un artículo-tutorial acerca de un aspecto concreto en el desarrollo de videojuegos.

wikipedia.org

gamasutra.com - Página para profesionales del sector de los videojuegos. Artículos semanales de muchos desarrolladores alrededor del mundo, discutiendo sobre diseño de videojuegos, creado post-mortem o simplemente enseñando soluciones técnicas que han usado anteriormente.

12.2. A*:

theory.stanford.edu - Página dedicada al análisis y estudio teórico de algoritmos y técnicas de programación.

blog.two-cats.com - Blog dedicado a la ingeniería del software.

12.3. Perlin Noise:

http://flafla2.github.io/2014/08/09/perlinnoise.html - Conceptos básicos acerca de Perlin Noise.

http://www.java-gaming.org/index.php?topic=36254.0 - Ejemplo práctico de uso de Perlin Noise para terrenos en 2D clásico.

http://netbook-game.blogspot.com.es/search/label/Terrain%20Generation - Infinidad de artículos y tutoriales acerca de la generación procedimental de terreno.

http://mattdietz.net/posts/perlin-noise-for-terrain-generation.html - Ejemplo práctico de uso de Perlin Noise para terrenos en top-down 2D.

http://freespace.virgin.net/hugo.elias/models/m_perlin.htm - Explicación extendida acerca de Perlin y otros ruidos.

http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/ N - Artículo muy completo acerca de la generación de una isla usando un mapa de polígonos. Usa Perlin Noise para controlar la forma.

12.4. BSP:

www.roguebasin.com - Página número 1 para los desarrolladores de juegos "rogue-like". Dónde empezar, generación procedimental de contenido y un sinfín de ejemplos y juegos reales que estudiar.

12.5. Arte del juego

http://forums.tigsource.com/ - Foro de discusión sobre desarrollo de videojuegos.

http://forums.flixel.org/ - Foro de Flixel, una librería open-source para desarrollo de videojuegos. Escrita completamente en AS3.

http://opengameart.org/ - Página donde artistas comparten sus creaciones con la comunidad. La mayor parte del arte en esta página es completamente gratis de usar, salvo mención de sus creadores en los créditos del juego.

12.6. Otros:

http://www.gamesindustry.biz/ - Noticias acerca de la industria del videojuego en general.

http://www.pcgamesn.com/ - Noticias acerca de la industria del videojuego en general.

http://w3.marietta.edu/~biol/biomes/biome_main.htm - Información acerca de los biomas del mundo real.

http://www.esrb.org/ - Página de evaluación de contenido de juegos, principalmente como guía para padres.

http://www.theesa.com/ - Página/servicio dedicada a conectar profesionales y empresas con publishers.

http://papersplea.se/ - Página oficial del juego Papers, Please.