Generación de Contenido por Procedimientos usando Materiales Prefabricados y Módulos.

Procedural Content Generation using Modules.

Walberto Michel Durán Sierra[1]
Alejandro Martínez Ledesma[1]
[1]Licenciatura en Artes Digitales,
División de Ingenierías del Campus Irapuato-Salamanca,
Universidad de Guanajuato

wm.duransierra@ugto.mx
a.martinezledesma@ugto.mx

Resumen

El objetivo de nuestro proyecto es crear un videojuego con las herramientas de Unity que tenga un escenario generado proceduralmente utilizando materiales prefabricados y módulos. La finalidad del juego es que el jugador encuentre la salida de cada nivel para pasar al próximo, lo que implica que nuestro escenario requiere de un inicio y un final. para la generación de los niveles utilizaremos una cuadricula 8x4, en la que, cada cuadro generara una habitación aleatoria. Para esto, el juego creara una ruta con una sala Inicial y una sala final. Una vez la ruta sea haya generado, el juego completara los cuadros sobrantes de la cuadricula con habitaciones aleatorias. Así, el jugador siempre tendrá una ruta disponible para acabar el nivel, sin embargo, la ruta marcada se verá limitada por las habitaciones que sean creadas por nuestro equipo, por lo que tendrán que ser diseñadas pensando en cómo se verían las salas en conjunto, teniendo en cuenta, la mayoría de las combinaciones posibles.

Palabras clave: Habitaciones; Ruta, Procedural, Plataformas.

Introducción

En los videojuegos, la generación de contenido por procedimientos (PCG por sus siglas en inglés), consiste en el uso de cualquier tipo de algoritmo con el objetivo de generar contenido sin la intervención directa o reducida de un desarrollador. Normalmente, el contenido a generar consiste en: texturas, niveles, personajes, comportamientos o efectos de sonido. Los sistemas y procedimientos que son integrados a los juegos deben de seguir una serie de reglas, diseños y métodos, que permitan que el juego sea funcional de principio a fin

El contenido generado de manera procedural representa una reducción de los costos de producción de un videojuego tradicional, tanto en tiempo como en dinero, lo que representaría para los estudios o cualquier tipo de desarrollador, el poder crear juegos en menor tiempo y más redituables. Según datos recabados en la investigación de Nicolás Barriga (2018), en la que habla sobre algoritmos procedurales. Se estima que un videojuego con presupuesto de AAA que normalmente suele llegar hasta los $150 millones de dólares, gasta cerca del 40% de este en la creación de contenido, lo que supondría un ahorro de este capital y de trabajo al utilizar generación procedural.

Actualmente el desarrollo de juegos procedurales para estudios Indies, pequeños estudios, representa el mayor campo de experimentación de estas herramientas, en parte, debido a que los presupuestos asignados para el desarrollo de estos son limitados, por lo que suelen valerse de la creación procedural para agilizar los tiempos de desarrollo y usarlos en su favor para ofrecer experiencias nuevas, creativas y únicas a las que ofrece el mercado de juegos AAA.

Para la aproximación de este proyecto se ha partido de la idea de hacer un juego de género Rouge-Like de plataformas. El género Rouge se caracteriza por tener niveles generados de manera procedural en forma de

mazmorras o calabozos en los que es tarea del jugador buscar las habitaciones correctas para poder salir y continuar con los niveles, contiene progresión libre y el eliminar enemigos durante el avance en las diferentes cámaras. Las mayores inspiraciones del proyecto son títulos como *Gonner (2016), Spelunky (2008), MegaMan (1987) y Zelda (1986).*

Figura 1. Implementación procedural de plataformas en Gonner (2016). Desarrollado por Art in Heart.

Gonner, desarrollado por Art in Heart, es un juego Rouge-Like de plataformas, los niveles se generan de manera procedural y premian la exploración del jugador con nuevos ítems, armas o modificadores que harán que cada partida sea personalizable por el usuario con los elementos descubiertos, generando diferentes comportamientos a lo largo de las salas, interactuando con objetos del escenario y con los enemigos generados. En el caso particular de Spelunky, desarrollado por Derek Yu se explora un escenario generado proceduralmente, el objetivo del jugador es encontrar la salida del nivel mientras reúne objetos que aumentaran su puntaje. El juego, para la generación de los niveles utiliza una cuadrícula 5x5, en la que, cada cuadro generará una habitación. Para esto, el juego creara una ruta con una sala Inicial y una sala final. Una vez que la ruta sea generada, el juego completará los cuadros sobrantes de la cuadrícula con habitaciones aleatorias. Así, el jugador siempre tendrá una ruta disponible para acabar el nivel.

Figura 2. Implementación procedural de salas en Spelunky (2008). Desarrollado por Derek Yu.

Otra de nuestras referencias para la creación de un juego de mazmorras con generación de salas, viene de la web de Darious Kazemi, en la que explica la lógica de cómo generar niveles partiendo del uso de rutas para después generar salas extras que completen un nivel, posteriormente agregando elementos como enemigos, ítems, entre otros. También nos apoyamos en el trabajo de investigación de Eduardo Cebollero (2022), de la Universidad San Jorge. En esta investigación se describen el funcionamiento de distintos generadores de mazmorras, incluidas la generación modular haciendo uso de habitaciones prefabricadas, que consideramos, es un punto importante dentro del desarrollo de este proyecto.

Los objetivos de este proyecto consisten en el desarrollo de una demo de juego completamente funcional de plataformas estilo Rouge, en dos dimensiones, con vista ortográfica que se encargue de generar salas de manera procedural haciendo uso de módulos, salas prefabricadas que generen un nivel siguiendo una ruta no definida dentro de un espacio limitado en el cual un personaje podrá navegar a través de las salas para encontrar enemigos y buscar una manera de pasar al siguiente nivel. El motor que será utilizado es Unity[1] en su versión 2022.1.4f1 y el lenguaje de programación será C#.

Metodología

Como se mencionó anteriormente, este proyecto utiliza como punto de partida las ideas propuestas por Darious Kazemi, y Eduardo Cebollero, en el que se propone utilizar un algoritmo que sea capaz de crear niveles para videojuegos, haciendo uso del seguimiento de una ruta de inicio a fin para las salas y del uso de salas prefabricadas o módulos, que puedan ser instanciados dentro de un espacio definido. Así como lo explica Cebollero, esta técnica hace uso de un conjunto de elementos de mazmorra prediseñados, en los que el desarrollador puede intervenir configurando ciertos parámetros como la longitud de las salas y de los componentes, además de los elementos que formarán parte del interior de la sala, como los tiles, objetos y enemigos.

Para el desarrollo del proyecto, se comenzó por crear un área para que las salas creadas por módulos pudieran instanciarse. Se generaron mapas de Tiles, en las que se pudieran organizar diferentes salas, inicial, final, derecha, izquierda, abajo, extras, entre otras. A estas se les agregaron diferentes *prefabs* y estos se subdividen conteniendo *assets* y *scripts* para que pudieran identificarse entre ellas y poder comenzar a inicializar las salas en el espacio. Se consideraron rangos y otros métodos para hacer que la generación del mundo siempre siga una ruta de inicio a fin y se creen salas extras a los alrededores. A continuación, se muestra una descripción de la metodología empleada en la creación del juego.

Los pasos que seguimos en la metodología son los siguientes:

1. Generación de nivel.
2. Generación de la maqueta.
3. Generación de la ruta.
4. Verificación de la ruta.
5. Resolviendo contradicciones.
6. Generación Aleatoria de salas.

1. GENERACIÓN DEL NIVEL

[1] Se recomienda ampliamente que se revise el diccionario al final del texto para un mayor entendimiento de la terminología específica del entorno de Unity, que es nombrada a lo largo de la Metodología.

La generación de niveles en nuestro proyecto comienza con la creación de una ruta, esta ruta, sigue una serie de condiciones que le indican cuándo ir a la izquierda o a la derecha y cuando bajar, contemplando, de antemano, un margen que ponga un límite para la creación del terreno y contando con la distancia que tendrá una habitación de otra.

2. GENERACION DE LA MAQUETA

Para esto, es necesario antes comenzar a maquetar nuestra escena, así que se necesaria la creación de dos tipos de objetos, objetos que ahora llamaremos GameObjects, El primer GameObject tendrá el propósito de contener el Script que inicia la generación del nivel y el segundo tendrá la función de ser un punto de aparición para nuestra habitación, que. De ahora en adelante llamaremos Punto de Spawn, El punto de Spawn, o, mejor dicho, los Puntos de Spawn, son GameObjects que no contienen absolutamente nada, porque su función parte de utilizar sus coordenadas para colocar las habitaciones, por ende, es necesario colocar los Puntos de Spawn en posiciones específicas para mantener la estructura del nivel, eso quiere decir, que la escena consta de filas de Puntos de Spawn. En nuestro proyecto, por ejemplo, colocamos nuestros Puntos de Spawn en filas de 8x4 con una separación de 10 unidades cada uno, cabe aclarar que en esta separación se toma en cuenta en todas las direcciones, tanto arriba, abajo y a hacia los lados. Para concluir esta sección, en la Figura 3 podemos ver como se vería visualmente la maqueta finalizada.

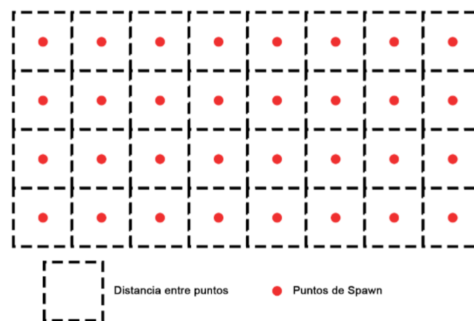

Figura 3. Maqueta finalizada

3. GENERACION DE LA RUTA

3.1 Inicio de la ruta

El código, cuando se inicializa, necesita que el usuario coloque una "Posición inicial" en la que comience la ruta del nivel. La Posición Inicial se coloca en el inspector de Unity, y en este caso, el usuario, puede colocar una cantidad ilimitada de Posiciones iniciales debido a que el código utiliza un arreglo dinámico, que ahora, para aclarar, cuando hablemos de un ArrayList, estaremos refiriéndonos a un arreglo dinámico. En este ArrayList será donde se colocarán los Punto de Spawn que el usuario considere indicados para comenzar la ruta.

Siendo más técnicos, cuando el código reconoce que existe un ArrayList de Posicion Incial, en un método start se inicializa una variable que llamaremos "Posición Inicial Aleatoria" que básicamente creará un número aleatorio entre cero y la cantidad de Posiciones Iniciales que haya en el ArrayList, esto es para tomar la posición de transformación que tenga el Punto de Spawn seleccionado por la variable. Por ejemplo, si "Posición Inicial Aleatoria" es igual a cero, el Punto de Spawn que se encuentre en el ArrayList como cero será el seleccionado para tomar sus coordenadas como punto de partida. Una vez ya exista ese punto de partida se hará un proceso similar al anterior, sin embargo, este necesita su propia explicación, pues, el código repite la misma lógica en varias partes.

3.2 Asignación aleatoria de salas

En el código, existe un método que en términos simples se encarga de inicializar una sala aleatoria, en este caso, utiliza la misma lógica que hablamos con anterioridad. El código le pide al usuario crear un ArrayList que contenga GameObjects en el inspector, el código al reconocer que existe un ArrayList, tomará una variable global que creará un número aleatorio entre 0 y la cantidad de GameObjects que haya en el ArrayList, y dependiendo del número que tome se mostrará una de las salas que hay en el ArrayList.

Para esto es necesario tener creadas las salas como un “Prefab” y colocarlas en el inspector previamente. Cabe aclarar que existe un ArrayList para cada tipo de sala y en nuestro caso, nosotros tenemos los siguientes ArrayLists: Sala Inicial”, “Sala Final”, “Sala con salidas a la izquierda y a la derecha”, “Sala con salidas a la izquierda, a la derecha y hacia abajo”, “Sala con salidas a la izquierda, a la derecha y hacia arriba”, “Sala con salidas a la izquierda, a la derecha, hacia abajo y hacia arriba” y “Habitaciones Extra”.

Es importante decir que cada ArrayList es independiente y tiene su propia línea de código para obtener una sala aleatoria e inicializarla. Particularmente, en el método que mencionamos al inicio de esta sección, se encarga de seleccionar con un Switch uno de los ArrayList dependiendo de la situación en la que se encuentre el código, y al final, cada caso del Switch ejecutará la línea de código en la que se inicializa una de las salas del ArrayList seleccionado de forma aleatoria.

Para ser específicos, este método solo tiene cuatro casos, las cuales son:

- Caso uno: “Se selecciona la sala con salidas a la izquierda y a la derecha.”
- Caso dos: “Se selecciona la sala con salidas a la izquierda, a la derecha y hacia abajo”
- Caso tres: “Se selecciona la sala con salidas a la izquierda, a la derecha y hacia arriba”
- Caso cuatro: “Se selecciona la sala con salidas a la izquierda, a la derecha, hacia abajo y hacia arriba”

Esto está diseñado así para optimizar el código que se encarga de trazar la dirección de la ruta, ya que, en varias partes se repetían las mismas líneas de código. Sin embargo, los otros Arraylist, como sólo se utilizaban en un caso específico, se considera innecesario agregarlos al switch, aun así, estos funcionan casi con la misma línea de código que inicializa las salas de manera aleatoria, la diferencia es que, en estos casos, no se llama al método antes mencionado, si no que se coloca la línea de código.

4. Verificación de La Ruta

4.1 Los rangos

En el código, el camino que tomará la ruta se ejecuta con un método llamado “Movimiento”, este método contendrá todas las condiciones necesarias para trazar la ruta, con la peculiaridad de que el camino generado sólo puede moverse hacia la izquierda, la derecha y hacia abajo. Teniendo estas tres posibilidades, se creó una variable con un rango de números del uno al cinco para crear las condiciones que moverán el camino. Esta variable, que se llama “dirección”, está diseñada para trabajar con la función “Random.Range”, con la cual cada vez que el camino necesite saber hacia dónde ir, utilizará la función para escoger un número aleatorio del uno al cinco y dependiendo de su elección, tomará un camino u otro. Para ser específicos estas son las direcciones que tomaron:

- Si “dirección” es igual a uno o dos la ruta irá a la derecha
- Si “dirección” es igual a tres o cuatro la ruta irá a la Izquierda
- Si “dirección” es igual a cinco la ruta irá hacia Abajo

Cabe aclarar que el rango puede variar en algunas ocasiones dependiendo de las circunstancias, sin embargo, en términos simples esta es la idea general de su funcionamiento.

4.2 El Margen

Otro elemento que utilizará la ruta es el margen, este margen consta de variables públicas que determinarán los límites de generación y asignación de salas, estas variables son: “PerimetroMinimo_X”, “PerimetroMaximo_X” y “PerimetroMinimo_Y”.

Estas variables, al ser públicas, pueden ser editadas en el Inspector, con lo que puede tener cualquier valor asignado por el usuario. Como notarán, en nuestras variables no se encuentra una llamada “PerimtroMaximo_Y”, esto se debe a que, técnicamente, el “PerímetroMáximo Y” ya sería marcado por las Posiciones Iniciales que hicimos al comienzo, y dado a que nuestra ruta solo bajará y nunca subirá, solo es necesario marcar hasta donde tendrá que bajar nuestra ruta en la variable “PerimetroMinimo_Y”.

4.2 Las condiciones

Las condiciones a primera vista pueden sonar muy repetitivas, sin embargo, son las pequeñas diferencias las que hacen cambios importantes en la generación de mundo. Recordando la sección de Rangos, habíamos mencionado que el método que se encarga del movimiento de la ruta es el que se llama “Movimiento”, para que esté método funcione, primero que todo, debemos crear un booleano, este booleano lo vamos a llamar “Parar generación del nivel” y como su propio nombre lo explica, este booleano marcará cuando nuestra ruta podrá terminar su camino. Así que, una vez creado el booleano, le asignamos el valor de “falso” y creamos la siguiente condición:

Si “Parar generación de nivel” es igual a falso, entonces el método “Movimiento” iniciará y, por ende, comenzará la ruta.

Esta condición finalmente será colocada en el método start, específicamente después de haber inicializado la sala inicial, así cada vez que se inicie la escena, también se va a iniciar el camino.

4.2.1 Código Ruta Derecha

En la Figura 4, se muestra el diagrama de ejecución de esta instrucción. Como se puede observar en esta figura, para que el camino vaya hacia la derecha, es necesario que la variable “dirección” sea igual a uno o sea igual a dos. Si la condición se cumple, habrá otra condición la cual dice que, si la posición en X es menor al Perimetro Maximo en X entonces, se iniciara la “Asignación Aleatoria de salas” con los casos del cero a tres. no sin antes crear una nueva posición con “Vector2” en la cual la posición en x se le sumarán diez unidades.

Una vez sea inicializada la nueva habitación en su nueva posición, la variable “dirección” volverá a sacar un valor aleatorio, en este caso, su rango será del uno al cuatro. Aquí, para evitar contradicciones, se crea otra condición que diga:

Si la dirección es igual a tres (Izquierda), la dirección cambiará a dos (Derecha)

De lo contrario, habrá otra condición que dice:

Si la dirección es igual a cuatro (Izquierda), la dirección cambiará a cinco (Abajo)

Esta condición existe para evitar que el no regrese el camino por donde ya vino. Ahora, retomando la primera condición, si esta no se cumple, es decir, la posición en X es mayor al Perimetro Maximo en X, en vez de usar un rango para dictaminar su próxima posición, el código le asigna a la dirección el valor de cinco o, en otras palabras, el camino se ve forzado a bajar.

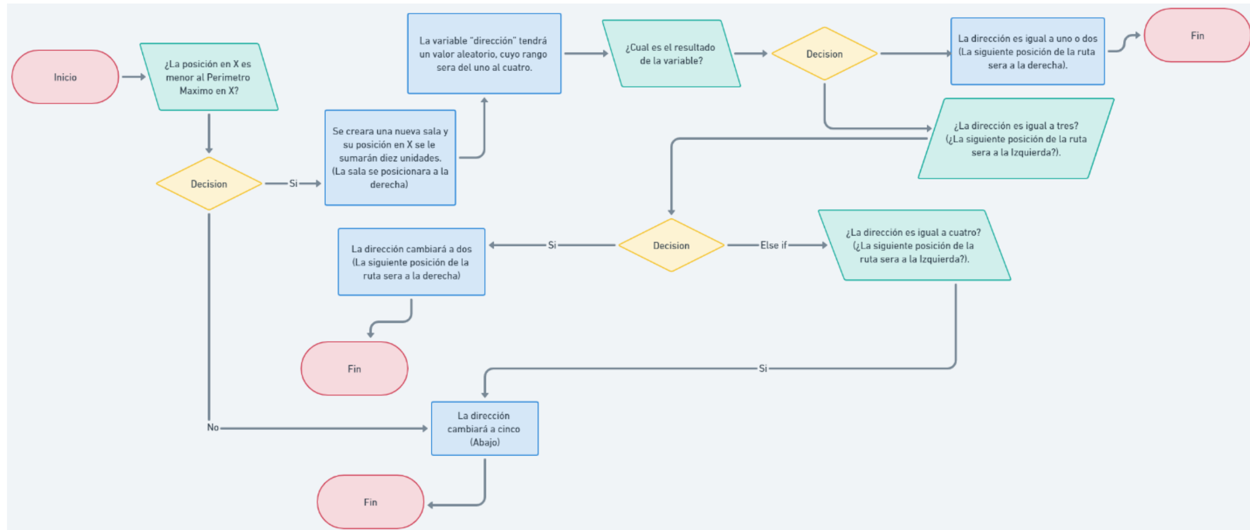

Figura 4. Diagrama de flujo. Ruta que se dirige hacia la Derecha

4.2.2 Código Ruta Izquierda

En la Figura 5, se muestra el diagrama de ejecución de esta instrucción. Como se puede observar en esta figura, Para que el camino vaya hacia la izquierda, es necesario que la variable "Dirección" sea igual a tres o sea igual a cuatro. Si la condición se cumple, habrá otra condición la cual dice que, si la posición en X es mayor al Perimetro Maximo en X entonces, se iniciara la "Asignación aleatoria de salas" con los casos del cero a tres. no sin antes crear una nueva posición con "Vector2" en la cual la posición en X se le restan diez unidades.

Una vez sea inicializada la nueva habitación en su nueva posición, la variable “Dirección” volverá a sacar un valor Aleatorio, en este caso, su rango será del tres al seis. Retomando la primera condición, si esta no se cumple, es decir, la posición en X es menor al Perimetro Maximo en X, el código le asigna a la dirección el valor de cinco o, y al igual que la ruta de la derecha, el camino se verá forzado a bajar.

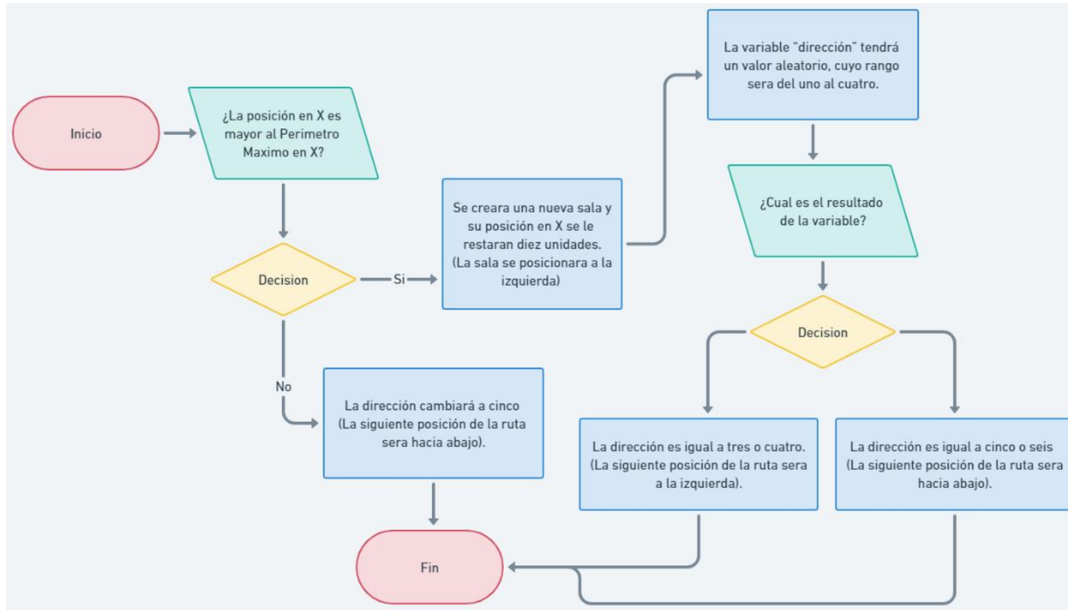

Figura 5. Diagrama de flujo. Ruta que se dirige hacia la Izquierda

4.2.3 Código Ruta Abajo

En la Figura 6, se muestra el diagrama de ejecución de esta instrucción. Como se puede observar en esta figura, Si el camino va hacia la hacia abajo, primero necesitará que la variable “dirección” sea igual a cinco. Si la condición se cumple, habrá otra condición la cual dice que, si entonces, se iniciará la “Asignación Aleatoria de salas” con los casos del dos al tres. Y nuevamente, su posición cambiará utilizando “Vector2”, en este caso, posición en Y se le sumarán diez unidades.

Una vez sea inicializada la nueva habitación en su nueva posición, la variable “Dirección” volverá a sacar un valor aleatoria, en este caso, su rango será del uno al cinco.

Finalmente, volviendo la primera condición, si esta no se cumple, es decir, la posición en Y es menor al Perímetro Mínimo en Y, La variable “Parar generación del nivel” se vuelve verdadera, acabando así la ruta.

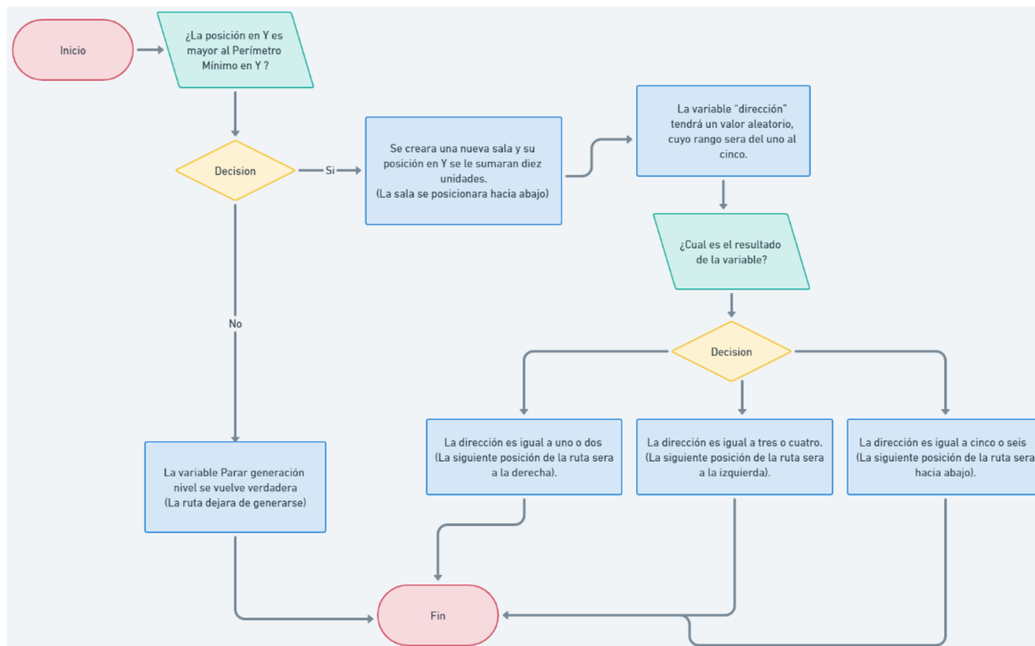

Figura 6. Diagrama de flujo. Ruta que se dirige hacia abajo

5. Resolviendo Contradicciones

Si bien, lo que comentamos puede tener lógica a primera vista, existen dos pequeños errores que no se consideran inicialmente. el primero “¿Qué pasa con la sala final?, ¿Cuándo el jugador sabrá que acaba la ruta?” y la segunda “¿Qué pasa si la ruta quiere bajar más de una vez?”. En resumidas cuentas, las respuestas a estas dos preguntas son: 1) El código no considera la sala final de momento, 2) Si la ruta quiere bajar más de una vez, puede colocar una sala incorrecta y con ello, bloquear la ruta.

Para resolver el segundo problema se pensó en simplemente, destruir y reemplazar la sala. Ahora la pregunta es, ¿Cómo se puede hacer eso? Si lo recuerdan, habíamos mencionado que nuestras habitaciones son Prefabs, dentro de los Prefabs, colocaremos un Punto de Spawn, específicamente en el punto de origen del Prefab.

Tras crear el Punto de Spawn, se creará una nueva capa en el inspector, recordemos que las capas, o como Unity las llama, las Layers, se usan para poder catalogar los GameObjects en una escena, tomando eso en cuenta, está Layer la nombraremos como “Máscara”. ¿Por qué? Si tenemos un Punto de Spawn en todas nuestras salas con la Layer Máscara puesta, el código podría identificar si existe un objeto en la misma posición que otro, es decir, si un objeto está encima de otro, ya que, estarían en la misma Layer. Sin embargo, esto lo veremos a profundidad más adelante, por ahora la creación de la Layer, simplemente está hecha para no interferir en las demás Layers que Unity crea desde un inicio y que el código no identifique objetos importantes en las salas como un “objeto encimado”.

Con esta lógica ahora crearemos un script que sea independiente al que se encarga de realizar la ruta, este script lo nombraremos “Destructor de salas”

Dada la finalidad del script, lo primero que debemos hacer es identificar el tipo de sala en la que se encuentra nuestra ruta, para eso, colocaremos una variable tipo Int llamada “tipo de sala”, esta variable será publica,

por lo que en el usuario será capaz de editarla desde el inspector. Después en el código, colocaremos un método llamado “Destruir sala” este método al ser llamado ejecutara la destrucción de un GameObject.

Recordando, anteriormente creamos un Punto de Spawn en los prefabs de las salas, dentro de estos Puntos de Spawn ahora colocaremos el script que acabamos de crear, y una vez colocado asignaremos el tipo de sala en el inspector. en nuestro caso, nuestras salas se dividen en:

- Tipo de sala cero: “Sala con salidas a la izquierda y a la derecha.”
- Tipo de sala uno: “Sala con salidas a la izquierda, a la derecha y hacia abajo”
- Tipo de sala dos: “Sala con salidas a la izquierda, a la derecha y hacia arriba”
- Tipo de sala tres: “Sala con salidas a la izquierda, a la derecha, hacia abajo y hacia arriba”

Tras haber asignado lo anterior a todos nuestros prefabs, volveremos al Código de la ruta.

En la condición “si la posición en Y es mayor al Perímetro Mínimo en Y entonces” crearemos una línea de código con la función Colider2D, este colider2D lo llamaremos “Detector”, y básicamente se encargará de identificar si hay algún objeto encima de nuestra posición, y para funcionar, tendrá que utilizar la Layer Macará que creamos con anterioridad. Ahora, con nuestro detector listo, iniciaremos una condición que diga:

“si el detector identifica que no hay una sala con salidas a la izquierda, a la derecha y hacia abajo” o “si el Detector identifica que no hay una sala con salidas a la izquierda, a la derecha, hacia abajo y hacia arriba” entonces se destruirá la sala e iniciará la “asignación aleatoria de salas”. Con esto que acabamos de agregar al código, en términos simples, hará que cada vez que la ruta decida ir hacia abajo, identifique si la sala en la que se encuentra tiene salidas hacia arriba y abajo, así, si en dado caso la ruta trata de bajar más de dos veces, se garantizara que las salas no se bloqueen a sí mismas.

Sin embargo, esto genera otro error, por ejemplo, si la ruta solo quiere bajar una vez, esto no importara, porque la sala que estaba será remplazada por una que tenga una salida hacia abajo, esto es porque el código está diseñado para considerar que la ruta vuelva a bajar sin ningún límite. Esto quiere decir que debemos crear otra condición antes de que destruya y remplace la sala, la condición debe considerar cuando la ruta quiere dejar de bajar. Para esto, es necesario que nuestro código sepa cuantas veces se moverá hacia abajo, entonces crearemos una variante llamada “Contador hacia abajo” esta variable será igual a cero, y será necesario inicializar ese valor en la ruta de Izquierda y la ruta de Derecha. sin embargo, en la ruta de Hacia abajo, la variable sumara una unidad cada vez que se inicie esta ruta. Bajo esta lógica, si el código toma la ruta hacia abajo el contador comenzará a aumentar, pero si cambia la ruta, se reiniciará el contador a cero.

Ahora que nuestro código sabe contar cuantas veces se va hacia abajo, comenzaremos a crear nuestra condición, en este caso, la condición dictara que si el contador es igual o mayor a dos entonces se destruirá la sala y se volverá a inicializar la asignación aleatoria de salas, en este caso con el valor tres, para que el único tipo de sala que pueda crear sea una sala que tenga una entrada de arriba y abajo, evitando así el bloqueo que teníamos desde un inicio.

Si en dado caso, el contador hacia abajo es menor a dos, entonces nuevamente se destruirá la sala, sin embargo, ahora inicializaremos el método para la asignación aleatoria de salas con el valor de uno, para que así, solo las salas que solo una entrada arriba, y salidas hacia la izquierda y derecha puedan ser creadas, arreglando así el segundo problema que generamos. Para un mejor apoyo visual, en la figura X podrán encontrar un diagrama de flujo con lo que explicamos con anterioridad.

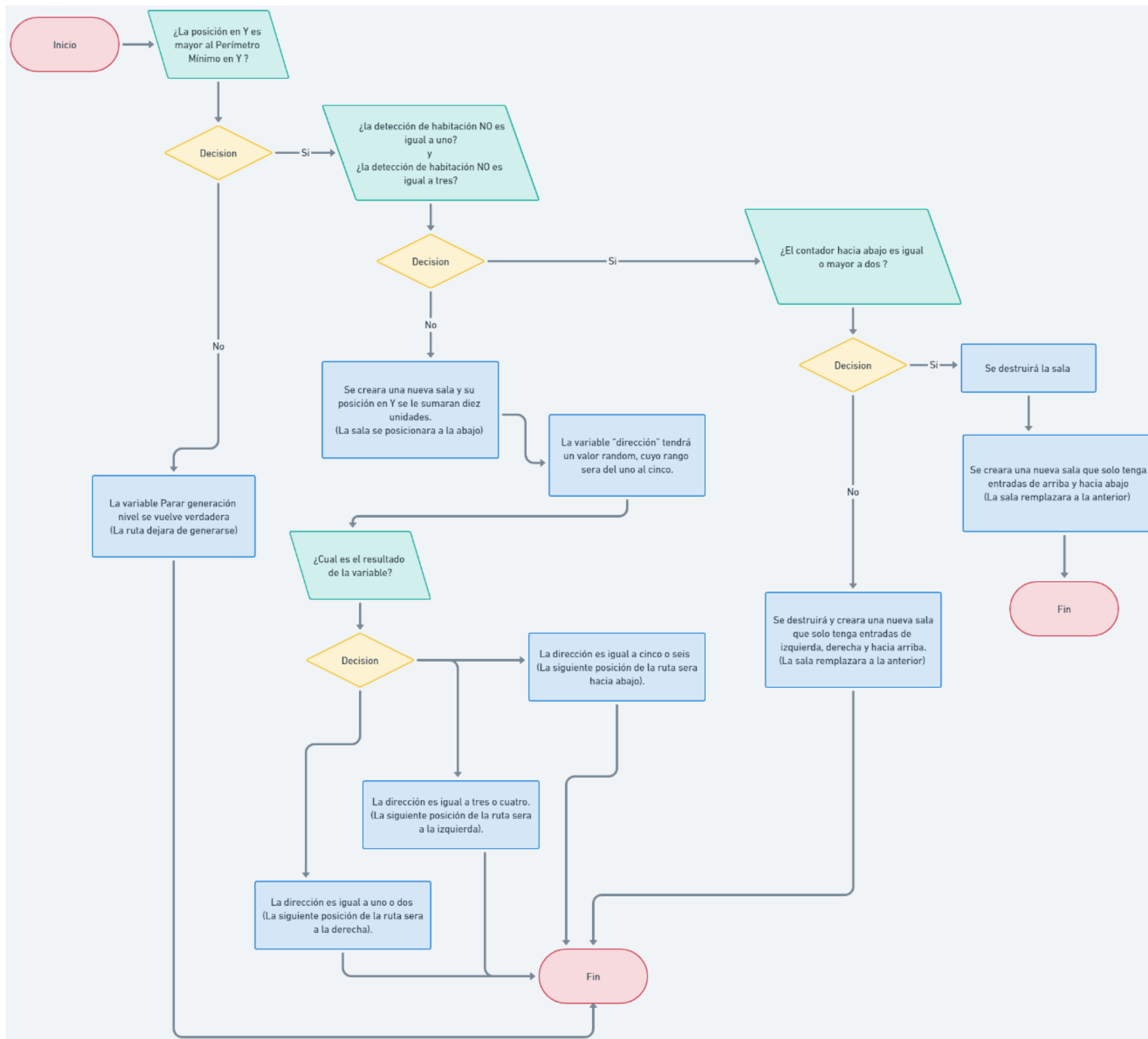

Figura 7. Diagrama de flujo. Ruta que se dirige hacia abajo considerando y arreglando los problemas planteados.

5.1 Sala final

Si recordamos, nuestro código no considera la creación de una sala final, así que volveremos a la ruta hacia abajo de nuestro código, en la parte donde no se cumplía la condición "si la posición en Y es mayor al Perímetro Mínimo en Y", en esa parte, lo que hacía nuestro código era que simplemente cambiaba el booleano Para la generación nivel de falso a verdadero. Para generar nuestra sala final, usaremos el código que generamos para destruir la sala, en este caso, nuevamente colocaremos un Detector que utiliza la Layer Máscara, e iniciaremos la destrucción de la última habitación generada, una vez destruida, instanciamos la asignación aleatoria de salas pero que solo tenga salas finales, y con eso se concluye la creación de la ruta exitosamente.

6. La generación aleatoria de salas

Aunque nuestra ruta haya concluido con éxito, aún sigue habiendo espacios vacíos en nuestra escena. Para esto, crearemos un código que se encargue de rellenar esos espacios vacíos con salas aleatorias. El código creará una variable global, que funcionará para colocar una LayerMask en el inspector, está LayerMask funcionará para seleccionar nuestra Layer Máscara. Después crearemos otra variable que llamará al script que genera la ruta del nivel. Tras haber creado nuestras variables, iniciaremos un método update, en el que colocaremos un detector. Una vez puesto el detector, crearemos una condición que dicte, si el detector tiene

un valor nulo, y también la variable Parar generación es verdadera, entonces se creará una a sala aleatoria, rellenando así los espacios vacíos.

Resultados

La inclusión del árbol de BSP en el algoritmo que utilizamos nos permite crear un escenario completo formado por bloques predefinidos, que haciendo uso de un sistema de rangos, calcula las mejores posibilidades para que una sala sea construida siguiendo una ruta de inicio a fin lo que permitirá que el jugador siempre tenga un lugar al cuál llegar, además de generar salar extras a los alrededores para dar esta sensación de mazmorra y completar el nivel. Al instanciar las salas, cada una toma de una lista de *prefabs* un valor pseudo aleatorio y elige la sala con la configuración adecuada para generar el mundo.

Todos los componentes que están en la escena instanciados se construyen por medio de un *GameObject* de tipo prefab, cada uno de estos prefabs, tienen la capacidad de tener una serie de componentes que nos permiten modificar su transformación en el espacio, su *SpriteRenderer*, su *Collider2D* que nos permite agregar *hitbox, hurtbox, floorPoints, shotingPoints;* Script y su Animator. Los prefabs se pueden clasificar de la siguiente manera:

- **Cámaras:** Cámara ortográfica que sigue al personaje dentro del nivel, se configura dentro de un prefab para poder utilizar en otras escenas.
- **Habitaciones:** Incluyen todas las salas inicio, final, izquierda, derecha, abajo y extras, en las que posteriormente se instanciarán el resto de los prefabs de este listado.
- **Escenarios:** Sprites y Tiles que se usan para crear los escenarios, desde elementos para el background, el midground y el foreground.
- **Personaje:** Personaje principal del juego capaz de ser utilizado en diferentes escenas.
- **Enemigos:** Personajes no controlados por el jugador que infringen daño al personaje principal y tienen patrones de comportamiento.
- **Items:** Consumibles en el juego y elementos que permiten al jugador poder cambiar de nivel y cargar una escena nueva.

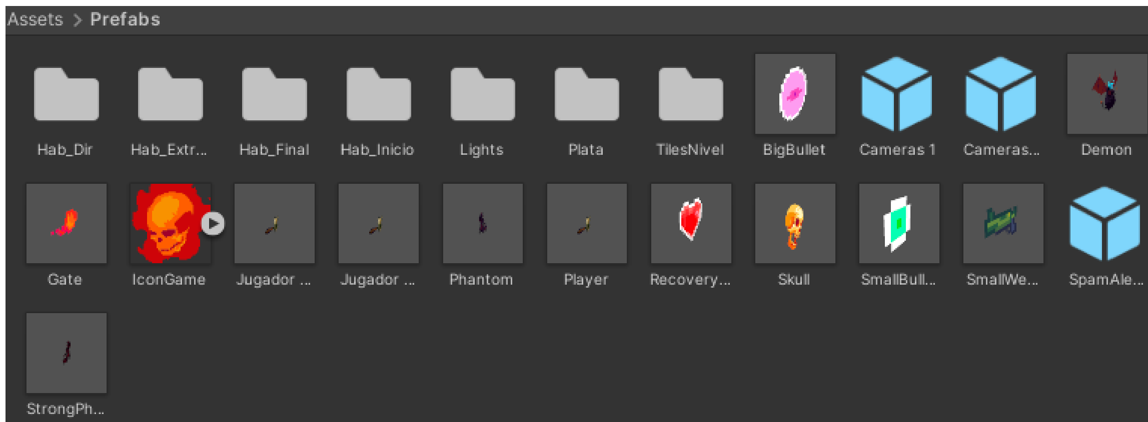

Figura 8. Prefabs de personaje, enemigos, tiles y habitaciones dentro de Unity.

Una vez seleccionada la sala inicial, el juego utiliza el sistema de rangos y crea todas las salas hasta que se cumple la condición de la creación de la sala fina, en este momento, el juego evalúa las salas ya creadas y autogenera salas extras con diferentes enemigos que servirán para complementar el laberinto con un tamaño de chunks total de 8x4.

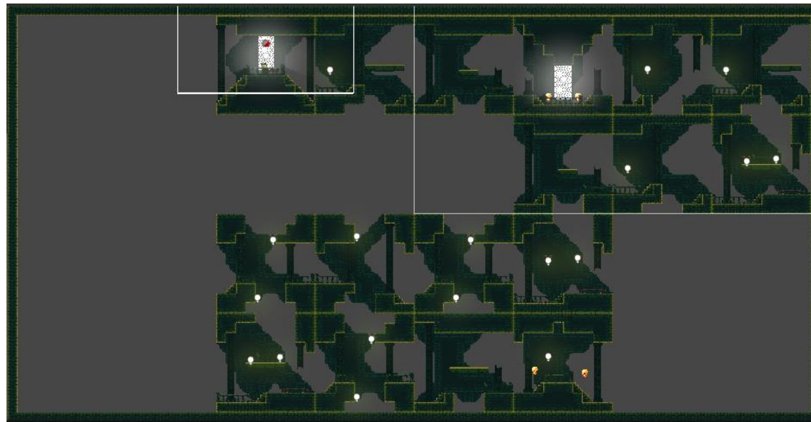

Figura 9. Creación de la sala inicial hasta la sala final.

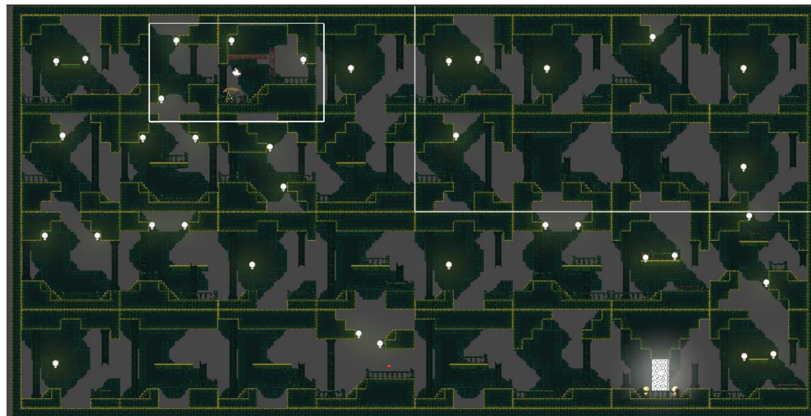

Figura 10. Creación de las salas extras una vez instanciada la sala final.

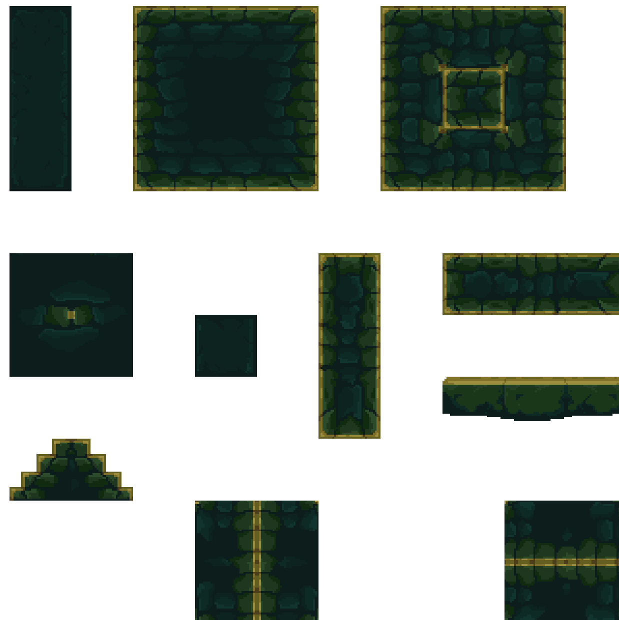

Figura 11. Tiles desarrollados para los niveles.

Ahora que el nivel ha sido generado, el programa también hace la instancia de los enemigos y del personaje jugable, cada uno de estos prefabs, contiene animaciones que fueron creadas con hojas de sprites y animadas con el componente animator dentro de Unity. Fueron creados diferentes controladores de animación cada uno con un trigger específico que, dependiendo de los inputs del jugador, el personaje principal tiene las opciones de caminar, atacar, brincar y al quedarse quieto tiene una animación de espera.

Los enemigos dentro del juego, como los son los fantasmas o las calaveras, tienen la capacidad de detectar la proximidad del jugador y atacarlo o seguirlo por el mapa. Las interacciones entre jugador y enemigos dieron lugar a tener integrado un sistema de vidas y de disparo de proyectiles, así como de la interacción de estos, incluyendo pantallas de inicio y fin del juego.

Figura 12. Primera columna Sprites de enemigo. Segunda columna Sprites del personaje principal.

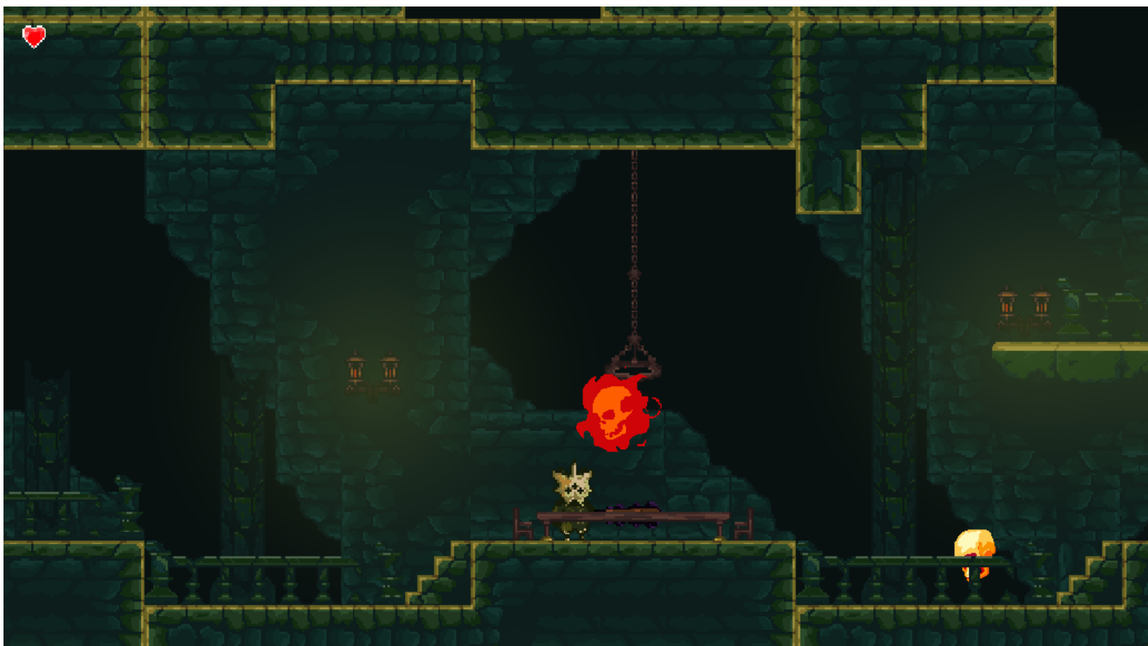

Figura 13. Captura del juego en ejecución.

Conclusiones

Este proyecto de videojuego ha obtenido resultados satisfactorios, pues se ha cumplido el objetivo de construir una demo de estilo Rouge, usando las técnicas modulares y los prefabs, que generan salas y niveles con enemigos, además distintas rutas para llegar de la sala inicial a la sala final. Basado en lo propuesto por la investigación de Darious Kazemi y de Eduardo Cebolleros, además hemos podido implementar diferentes mecánicas que aumentan la jugabilidad del demo, pudiendo permitir al jugador navegar por el mapa creado, conseguir objetos y pelear contra los enemigos. Este sistema de creación de niveles nos permite poder tener mayor control de las salas que se construyen y consideramos que es una buena introducción a la generación procedural para cualquier videojuego de este género.

A pesar de que lo procedural es funcional. Lo jugable requiere de aún más trabajo, pues de momento solo puedes atacar unos cuantos enemigos y explorar el mundo creado. Aun así, creemos que existe un gran potencial en el proyecto pues la forma en que lo diseñamos, aprovechado las herramientas que provee Unity, le permite al desarrollador explotar las capacidades técnicas de nuestra generación procedural agregando distintas capas de diseño que pueden aportar a la jugabilidad, la estética y a las mecánicas del juego.

Terminología en Unity

Asset: Es cualquier tipo de elemento que puede ser utilizado dentro de un juego o un proyecto. Puede ir desde texturas, imágenes, sonidos, animaciones hasta los archivos que se pueden crear dentro de Unity, como los componentes de animación, luz, materiales, entre otros.

Componentes: Son las piezas funcionales de cada uno de los objetos creados, proveen las funciones necesarias para poder agregar comportamiento a un objeto que será utilizado en el juego. Los hay de muchos tipos, pero por defecto, cada objeto creado tendrá siempre un componente *Transform,* que permitirá su posicionamiento dentro de un plano al ser instanciado en la escena.

Existen también otros componentes como lo son el Sprite Renderer, que dan la posibilidad de poner una imagen o un conjunto de imágenes dentro del objeto. Scripts, códigos de programación que permiten a los desarrolladores crear mecánicas o lógicas de interacción del juego. Componentes de animación, componentes de físicas, *colliders*, componentes de audio, entre otros.

Collider: El *collider* es un tipo de componente que se le agrega a un objeto para definir su forma, con el objetivo de implementar colisiones físicas con otros objetos dentro de la escena. Este componente también permite el uso de otros componentes de físicas como lo son el Rigibody que permite que el objeto sea afectado por las físicas, como el movimiento, la gravedad, resistencia entre otros elementos.

Trigger: Es un término utilizado para mencionar que un objeto está accionando un collider dentro de otro objeto. Cuando un collider es configurado con la opción Trigger, este objeto dejará de tener capacidades físicas, permitiendo que otros colliders pasen sobre él, esto permite tener sistemas de detección y generar interacciones por medio de código.

Tilemap: El *Tilemap* es un componente dentro de un objeto que permite pintar assets de tipo Tile o azulejos sobre él, este tipo de objeto se utiliza para poder usar diferentes sprites dentro de un mismo objeto tilemap y pintar escenarios.

Prefab: Es un objeto reutilizable que contiene componentes seleccionados de manera arbitraria. Este objeto, con componentes agregados, se puede utilizar como objeto reutilizable cuando se permite hacer varias instancias durante el proyecto, tiene la ventaja de que puede construirse una sola vez, instanciado y de modificarse individualmente. Pero el objeto de tipo prefab siempre tendrá las mismas características.

Referencias

Jerez, D.(2017).Generación de contenido procedural en videojuegos basados en Unity.[Trabajo de Fin de Grado, Universidad Carlos III de Madrid] Recuperado de: https://repositorio.usm.cl/handle/11673/49444

Kazemi,D. Spelunky Generator Lessons. (s.f.). Recuperdado de: http://tinysubversions.com/spelunkyGen/

Scheihing, C.(2019). Diseño de Algoritmo de Generación Procedural Enfocado a Videojuegos. [Memoria Para Optar Al Título De Ingeniero Civil En Informática, Universidad Técnica Federico Santa María]. Recuperado de: https://e-archivo.uc3m.es/handle/10016/27604

M. Hendrikx, S. Meijer, J. Van Der Velden and A. Iosup, Procedural content gener-ation for games: A survey, ACM Transactions on Multimedia Computing, Communi-cations, and Applications (TOMM).

Cebollero, P.(2022).Sistemas de Sistema de generación procedural de niveles aplicado al género de videojuegos "Rogue-Like" [Grado en Ingeniería Informática, Universidad San Jorge Escuela de Arquitectura y Tecnología] Recuperado de: https://repositorio.usj.es/bitstream/123456789/862/1/Sistema%20de%20generaci%C3%B3n%20procedural%20de%20niveles.pdf

Barriga, N.(2018). A short Introduction to Procedural Generation Algorithms for Videogames[School of Videogames Development and Virtual Reality Engineering Universidad de Talca] Recuperado de: https://www.researchgate.net/publication/331717755_A_Short_Introduction_to_Procedural_Content_Generation_Algorithms_for_Videogames

Martínez, R.(2015). Videojuego basado en la generación procedural de mundos o niveles[Grado en Ingeniería Multimedia, Escuela Politécnica Superior] Recuperado de: https://rua.ua.es/dspace/bitstream/10045/48231/1/Videojuego_basado_en_la_generacion_procedimental_de_mu_MARTINEZ_VILAR_RUBEN.pdf

Art in heart. (2016). Gonner. (Version para computadoras) [Videojuego]. Raw Fury

Yu, D. Mossmouth, Blitworks.(2008). Spelunky (Version para computadoras) [Videojuegos]. Mossmouth, Xbox Game Studios, Microsoft Studios.

Megaman (Version de Nintendo Entertainment System) [Videojuego]. (1987). Capcom

The Legend of Zelda (Version de Nintendo Entertainment System) [Videojuego]. (1986). Nintendo