Informe del primer proyecto de programación

Basado en las especificaciones del proyecto sobre hacer un juego multijugador con temática de laberinto nació Laberinthya, un trabajo hecho completamente en consola (o sea, no tiene cosas raras). El primer menú que ofrece el juego es bastante intuitivo ya que da a elegir cuatro opciones (iniciar partida, ver tablero, ver reglas y salir), las cuales se explicarán a continuación. La opción de ver tablero la creé con la intención de comprobar desde el inicio una de las condiciones que exigía el proyecto (un tablero que fuera aleatorio), sin la necesidad de iniciar partida varias veces. Ver reglas es una especie de tutorial sobre el juego en forma de menú que además de guiar a los futuros jugadores, también sirve para aclarar las otras condiciones del juego: existencia de varias fichas elegibles con habilidades únicas y misma cantidad para cada jugador, tiempos de enfriamiento para estas habilidades, imposibilidad de que algún jugador no pueda seguir jugando, etc. La opción de iniciar partida da comienzo al juego. Con esto se va pidiendo paso a paso a los jugadores que elijan entre varias opciones para personalizar a sus personajes y, si entraste por error, siempre puedes presionar en la última opción: salir. Si te decides a jugar, luego de que cada jugador cree su personaje saldrá un menú informativo donde se encuentran cada uno de sus datos. Si por accidente elegiste algo que no querías puedes usar la opción de volver al inicio que saldrá casi al iniciar la partida, aunque esto reiniciará todo el proceso para todos los jugadores.

A continuación se mostrarán una serie de datos sobre este juego que sirven además para una lectura más entendible de este resumen:

* Siempre que se hable de muros, me refiero al símbolo ‘#’.
* Siempre que se hable de cofres, me refiero al símbolo ‘@’.
* Siempre que se hable de caminos, me refiero al símbolo ‘.’.
* Siempre que se hable de puertas, me refiero a la transformación de un muro en un camino.

Cuando comencé a trabajar en el proyecto ya tenía claro que iba a necesitar de tres cosas fundamentales: jugadores, un campo para jugar y algo que conectara ambos conceptos para crear un ambiente interactivo. Lo primero que hice fue crear una clase llamada Player que poseía los atributos básicos de los jugadores, o sea, el nombre, las habilidades y la velocidad, los dos primeros representados por atributos de clase de tipo string y el último por otro de tipo int. Luego creé un array de tipo Player de tamaño 3 en el método Main de la clase Program, seguido de un bucle for que recorre el array y lo inicializa en sus dos primeros elementos, o en todos, dependiendo de la cantidad de jugadores que sean, para así poder darle características individuales a cada uno. Mi primer problema fue crear una forma de elegir las habilidades de manera que hubiera variedad, pero sin extender demasiado el proceso. La solución fue crear el concepto de “empleo”, que consiste en que, al ser un laberinto (en mi imaginación dentro de una cueva), cada jugador puede elegir su profesión o empleo antes de comenzar a jugar. Cada una de estas cinco profesiones trae consigo una habilidad especial y velocidad únicas. Además, luego de elegir tu empleo puedes escoger dos de cuatro habilidades diferentes que son comunes para todos y le aportan más dinamismo al juego. Para personalizar a cada jugador el método Constructor de la clase Player llama a otros tres métodos de la misma clase los cuales llamé *Dar\_nombre, Dar\_empleo* y *Dar\_habilidades*. Estos métodos van llenando los distintos atributos de la clase con valores elegibles que van siendo presentados con mensajes mediante algunos *Console.WriteLine*.

Lo próximo que hice fue crear una nueva clase llamada Board donde el único atributo es una array de arrays de tipo char llamado tablero. El Constructor de esta clase inicializa este array con unos 30 arrays de 152 elementos cada uno, y luego comienza a guardar valores en estos espacios con el método *Inicializar*. Este método comienza rellenando las filas y columnas de los extremos con muros. Luego, divide al mapa en unos 8 pedazos iguales (4 superiores y 4 inferiores llamados terrenos), con tres columnas y una fila llenas de muros, para después abrir distintas puertas por todo el mapa, excepto claro, en los muros exteriores. Para las puertas verticales usé un bulcle doble for que recorre todo el tablero (menos los bordes) con una condicional if y una variable Random que lanza un número del 1 al 9, llamada de la misma forma. Esto lo que hace es que si la casilla del tablero (o sea la fila y columna de cada vuelta del bucle doble for) tiene guardado un muro y random lanza un 7, se crea una puerta.

El problema con esto es que había veces donde no se creaba ni una sola puerta entre dos terrenos. Para arreglar este error creé algunas casillas al final del método con un valor constante que aseguraban al menos una puerta entre terrenos y además evitaban que se creara una puerta en la casilla que interceptaba los grupos de muros verticales y horizontales, para evitar así casillas inalcanzables. El próximo problema que me surgió fue estético, ya que la abundante cantidad de puertas que conectaban a los terrenos superiores con los inferiores hacía ver el mapa, en mi opinión, poco trabajado. Mi respuesta fue crear los métodos *Fila\_14* y *Cortes*. *Fila\_14* recorre por separado las cuatro secciones que dividen a los terrenos inferiores de los superiores y luego, mediante dos variables Random, elige la cantidad de puertas y el tamaño de cada una usando el segundo método, *Cortes.* Para hacerlo aún más agradable a la vista, la primera y última sección tienen puertas con un tamaño variable, mientras que las dos restantes lo tienen fijo. Por cierto, todo lo que no fueran muros se rellenó con caminos.

Ya tenía las bases del mapa, pero aún faltaba una manera de colocar cofres, muros y llaves (condición para ganar) de tal forma que no se generaran casillas inalcanzables y que fuera algo agradable a la vista. Para este conflicto usé dos métodos: *Estructuras*  y *Generar\_Estructuras.* El método *Estructuras,* como bien indica su nombre, recorre cada uno de los terrenos y asigna una estructura prediseñada por mí y generada por *Generar\_Estructuras*. Para lograr que este proceso fuera aleatorio, le di un número a cada una e hice varios arrays con estos números ordenados de varias formas. Además, usé una variable Random que genera un número menor a la cantidad de arrays, eligiendo así con cual trabajar y de esta forma, el orden de las estructuras en el mapa.

Luego de esto lo único que me faltaba era crear algo que uniera el mapa ya inicializado con los jugadores y sus características. Ese algo fue la clase Actions, que se resume en un grupo de métodos que hacen que el juego funcione. El primero de ellos es *Caminar,* y para que el jugador “camine”, en realidad lo que pasa es que cada objeto de tipo Player guarda su posición en dos atributos: fila y columna, además de guardar su letra de jugador y mostrarla en alguna parte aleatoria del mapa. El proceso de caminar consiste en modificar la posición del jugador, en dependencia del sitio a donde quiera ir y mover la letra en esa dirección. Si en el lugar al que te quieres mover hay un muro, un mensaje te avisará de que has chocado con una pared y perderás un movimiento, si es un llave saldrá un mensaje que te mostrará cuantas te faltan, si hay un camino puedes pasar mientras te queden movimientos, y si hay un cofre este se destruirá una vez revisado y te mandará automáticamente con el método *Cofres\_Y\_Trampas*. Este método es sencillo, solo usa una variable Random que lanza números del 1 al 21 y mediante intervalos de 5 va cambiando lo que puede pasar. Hay tres intervalos de trampas que alteran las estadísticas de los jugadores con puntos de veneno y cosas por el estilo, uno de bonificación, y exactamente el número 21 es una llave (mira, otra forma de ganar). Para el tema de las habilidades creé dos métodos: *Habilidades*  y *Usar\_Habilidades.* El primero es el código que determina qué hace cada una y el segundo es el que da la opción de cuál habilidad usar según las que posea el jugador.

Creo que el principal problema aquí surgió a causa de los tiempos de enfriamiento. Al principio intenté crear un sistema de turnos, pero me creaba conflicto con los movimientos y se dificultaba el hecho de individualizar el tiempo de cada habilidad. Lo que hice fue crear un atributo en la clase Player que fuera un array de tipo int, llamado opciones, con el objetivo de asignarle a cada una de sus posiciones una habilidad, y de esta manera, colocar el número de turnos que hay que esperar luego de usarla. Después de que a cada jugador se le acaben los movimientos, se les resta 1 a cada valor que esté guardado en el array, a menos que sea 0 o menor, en cuyo caso su valor se vuelve 0 y permite usar la habilidad nuevamente. Un último método de esta clase es *Ver\_Habilidades*, que básicamentete muestra un menú con la descripción de todas ellas y sus tiempos de enfriamiento.

Para terminar, regresé a la clase Program para los últimos detalles. Allí creé un bucle while con la condición de que el booleano end permanezca en false. Para cambiar esto y que termine el bucle (que se traduce en ganar o salir del juego), el número del atributo llaves del jugador actual debe ser 3 (también puedes presionar la letra "l"). Este bucle trabaja con un atributo tipo int de esta misma clase llamado movimientos, el cual se iguala a la velocidad del jugador de turno y luego de cada acción resta uno, para posteriormente pasar al siguiente jugador y repetir el proceso. Para que al usar alguna habilidad o ver las habilidades no se restara un movimiento innecesariamente, luego de cada una de estas acciones se le suma 1 a movimientos. Mi último problema fue estético, ya que luego de que un jugador haga cualquier acción un Console.Clear limpia la pantalla, pero esto causaba conflicto a la hora de mostrar los mensajes de los efectos de los cofres, el de haber obtenido una llave, etc. Lo que se me ocurrió fue poner un Console.ReadLine antes del Console.Clear, pero después de la acción. De esta forma la pantalla se actualiza luego de que salgan los mensajes correspondientes y presiones cualquier tecla.

Proyecto finalizado. A modo de curiosidad quería escribir también la descripción de un par de métodos que utilicé para evitar excepciones y hacerme la vida más fácil, ambos de la clase Program. El primero fue Validar\_int. Este método devuelve un valor int ingresado por el usuario, pero si detecta que no es válido se llama a sí mismo y pide un nuevo valor; esto mediante un try y un catch que guarda la excepción, en caso de que haya, y luego se llama al método. El otro es While\_int (básicamente una forma de no escribir lo mismo varias veces). Este método pide tres entradas de tipo int que son los extremos a comparar y un valor, o sea, mientras el valor sea menor que el primero o mayor que el segundo pide que ingreses nuevamente un número. También creé otro par que fue Validar\_char y Validar\_string. El primero funciona de forma similar a Validar\_int y el segundo fue más para evitar ponerles nombres numéricos a los personajes. Eso es todo.