Maze Runners Pokemon Dungeon

Requisitos de uso:

1- Entorno de desarrollo: Un editor de texto con soporte para C# como Visual Studio o Visual Studio Code.
2- .NET SDK: asegúrate de tener instalado esto en tu maquina
3- Paquete NuGet de Spectre.Console: Spectre.Console es un biblioteca la cual permite visualizar las imagines y los gráficos.
4- Paquete NuGet de NAudio: es una biblioteca la cual permite que el juego tenga audio.

Es importante cumplir estos requisitos para una mejor experiencia de juego.

¿Cómo empezar a jugar?

Al ejecutar el juego aparece el menú, que contiene dos opciones (empezar a jugar y salir del juego). Al iniciar el juego se le pide a ambos jugadores introducir sus nicknames (se acepta cualquier carácter siempre, no se permite dejar el espacio en blanco). Luego se les da a los usuarios la opción de seleccionar la cantidad de fichas con las que van a jugar, que serán entre 1 a 5 fichas. Cada jugador seleccionara sus fichas acorde a la cantidad que seleccionaron anteriormente.

Las fichas, basadas en el juego POKEMON, tienen la siguiente composición: Nombre, Velocidad, Habilidad.

Velocidad: representa la cantidad de casillas que se puede mover la ficha en un turno.

Habilidad: representa el efecto que puede realizar la ficha.

Tipos de habilidades

1- Ignorante: Al activar la habilidad, si la ficha pasa por una trampa, esta no se activará.
2- Oversoul: Convierte en 1 la velocidad de todas las fichas enemigas por tres turnos.
3- Sombra trampa: Crea en la posición en la que se encuentra una trampa de ralentización.
4- Lullaby: Duerme dos fichas enemigas aleatorias y les impide moverse por tres turnos.
5- Agilidad: Aumenta la velocidad en 3 durante el turno en que se activó.
6- Ventisca: Congela todas las fichas enemigas durante un turno y permite que se vuelva a jugar otro turno.
7- Don Aural: Durante 5 turnos esta ficha no será afectada por las habilidades de las fichas enemigas.
8- Teletransporte: Cuando la ficha activa esta habilidad la envía una posición aleatoria y validad del tablero.

Todas estas habilidades tendrán un cooldowns (las habilidades tendrán un tiempo de enfriamiento antes de volver a activarlas) para que sea un juego justo.

Selección de dificultad:

Para hacerlo más interesante, el jugador podrá elegir la dificultad, entre las siguientes: Tutorial, Normal y Pesadilla siendo esta ultima la más desafiante.

El tutorial consistirá en un pequeño laberinto donde se podrán ver las trampas y adaptarse al modo de juego.

En la dificultad normal el laberinto se vuelve más grande y las trampas no pueden verse, dificultando el paso hacia la meta.

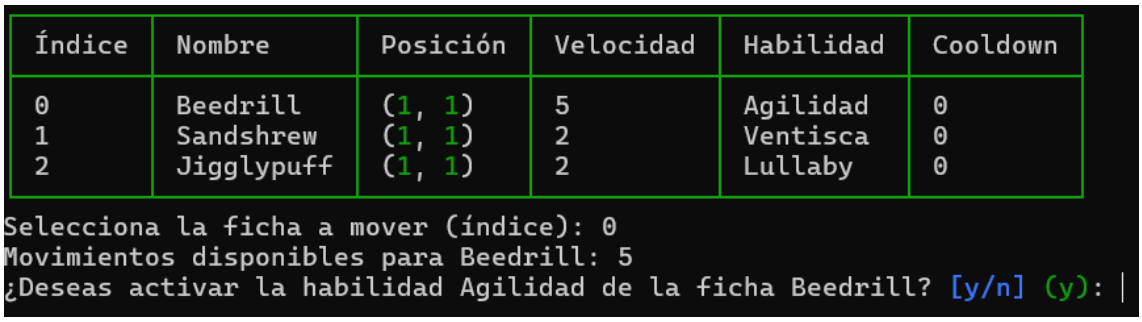
En el modo pesadilla el laberinto es mucho más grande y complicado, las trampas no se ven y aumentan su número, complicando mucho más el paso hacia la meta.

Empezar a jugar:

Tras cumplir con todos los parámetros para jugar se generara lo siguiente:

Este es un ejemplo del laberinto normal, donde todas las fichas comenzaran en el emoji de la puerta, y la condición de victoria es que todas las fichas de un jugador lleguen al emoji de la meta. Los números de los laterales ayudan para comprobar la posición de tu ficha visualmente. Dependiendo de la dificultad los obstáculos para dificultar la movilidad en los laberintos varían entre piedra, árbol y telaraña.

Bajo el laberinto se va a mostrar toda la información de las fichas:



Cada columna representa un tipo de información:

Índice: Representa la ficha a seleccionar, que va desde 0 hasta la n-1 cantidad de fichas seleccionadas.

Nombre: nombre de la ficha.

Posición: representa en colores la posición de la ficha para poder ubicarla mejor en el laberinto de la forma (x,y) donde las x representan los números verticales y las y los horizontales.

Velocidad: Cantidad de casillas que se puede mover la ficha por turnos.

Habilidad: Nombre de la habilidad de la ficha.

Cooldown: Tiempo de enfriamiento de la habilidad, mientras este en 0 se puede utilizar.

Se le pregunta al jugador la ficha a mover por el índice y si desea activar la habilidad o no con las siguientes teclas (y: para sí activar la habilidad; n: para no activar la habilidad) y puedes empezar a moverte con las teclas de dirección situadas en la parte inferior del teclado.

Diviértase hasta que gane el mejor corredor de laberintos.

Detalles de la implementación:

Mi proyecto cuenta con 5 clases:

Maze: Encargada de la generación y visualización del laberinto.

Player: Encargada del jugador y movimiento de las fichas a través del laberinto.

Ficha: Lógica de las fichas, incluyendo habilidades y efectos de las trampas.

Game: Incluye toda la lógica del juego.

MusicPlay: Encargada de hacer que suene la música.

Apoyándome de la biblioteca Spectre.Console para crear el visual, lo primero que hice fue implementar la generación aleatoria del laberinto:

```
/// <summary>
/// Genera el laberinto utilizando un algoritmo de crecimiento basado en fronteras.
/// </summary>
2 references
public void GenerateMaze()
{
    // Inicializa todas las casillas como paredes
    for (int x = 0; x < Width; x++)
    {
        for (int y = 0; y < Height; y++)
        {
            maze[y, x] = WALL; // Marca la celda como pared
        }
    }

    // Punto de inicio del laberinto
    var startX = 1;
    var startY = 1;
    maze[startY, startX] = PATH; // Marca el punto de inicio como camino

    var frontier = new List<(int x, int y)>();
    AddFrontier(startX, startY, frontier); // Añade las fronteras iniciales

    // Algoritmo de generación de laberinto
    while (frontier.Count > 0)
    {
        var current = frontier[rand.Next(frontier.Count)]; // Selecciona una celda aleatoria de la frontera
        frontier.Remove(current); // Elimina la celda seleccionada de la frontera

        var neighbors = GetNeighbors(current.x, current.y); // Obtiene los vecinos válidos de la celda seleccionada
        if (neighbors.Count > 0)
        {
            var neighbor = neighbors[rand.Next(neighbors.Count)]; // Selecciona un vecino aleatorio
            maze[current.y, current.x] = PATH; // Marca la celda actual como camino
            maze[(current.y + neighbor.y) / 2, (current.x + neighbor.x) / 2] = PATH; // Marca la celda intermedia como camino
            AddFrontier(current.x, current.y, frontier); // Añade las nuevas fronteras

            // Asegurarse de que hay al menos una vuelta en el camino
            if ((current.x == startX && current.y == startY + 1) || (current.x == startX + 1 && current.y == startY))
            {
                var next = frontier[rand.Next(frontier.Count)]; // Selecciona una celda adicional de la frontera
                maze[next.y, next.x] = PATH; // Marca la celda adicional como camino
                maze[(next.y + neighbor.y) / 2, (next.x + neighbor.x) / 2] = PATH; // Marca la celda intermedia como camino
                AddFrontier(next.x, next.y, frontier); // Añade las nuevas fronteras
            }
        }
    }
```

Este código funciona de la siguiente manera, se recorre toda la matriz ¨maze¨ y se inicializa cada celda como una pared. Se declara la posición (1,1) como punto de inicio y se designa como camino, se añade la ¨frontera¨ inicial alrededor del punto de inicio. Se selecciona aleatoriamente una ¨frontera¨ y se elimina de la lista, se obtienen los ¨vecinos¨ validos de la celda seleccionada. Se selecciona un ¨vecino¨ aleatorio y se marca como camino la celda actual y la celda intermedia. Se añaden nuevas ¨fronteras¨ desde la celda actual .Se asegura que hay al menos una vuelta en el camino añadiendo una celda adicional en la frontera si es necesario. Marca la penúltima celda como camino para asegurar la salida. Se bloquea la última fila excepto la salida .Se colocan obstáculos y trampas al azar, asegurando que la casilla final no quede bloqueada .Se hacen inaccesibles los bordes del laberinto .Se asegura que la meta (salida) no esté bloqueada por trampas de teletransportación.

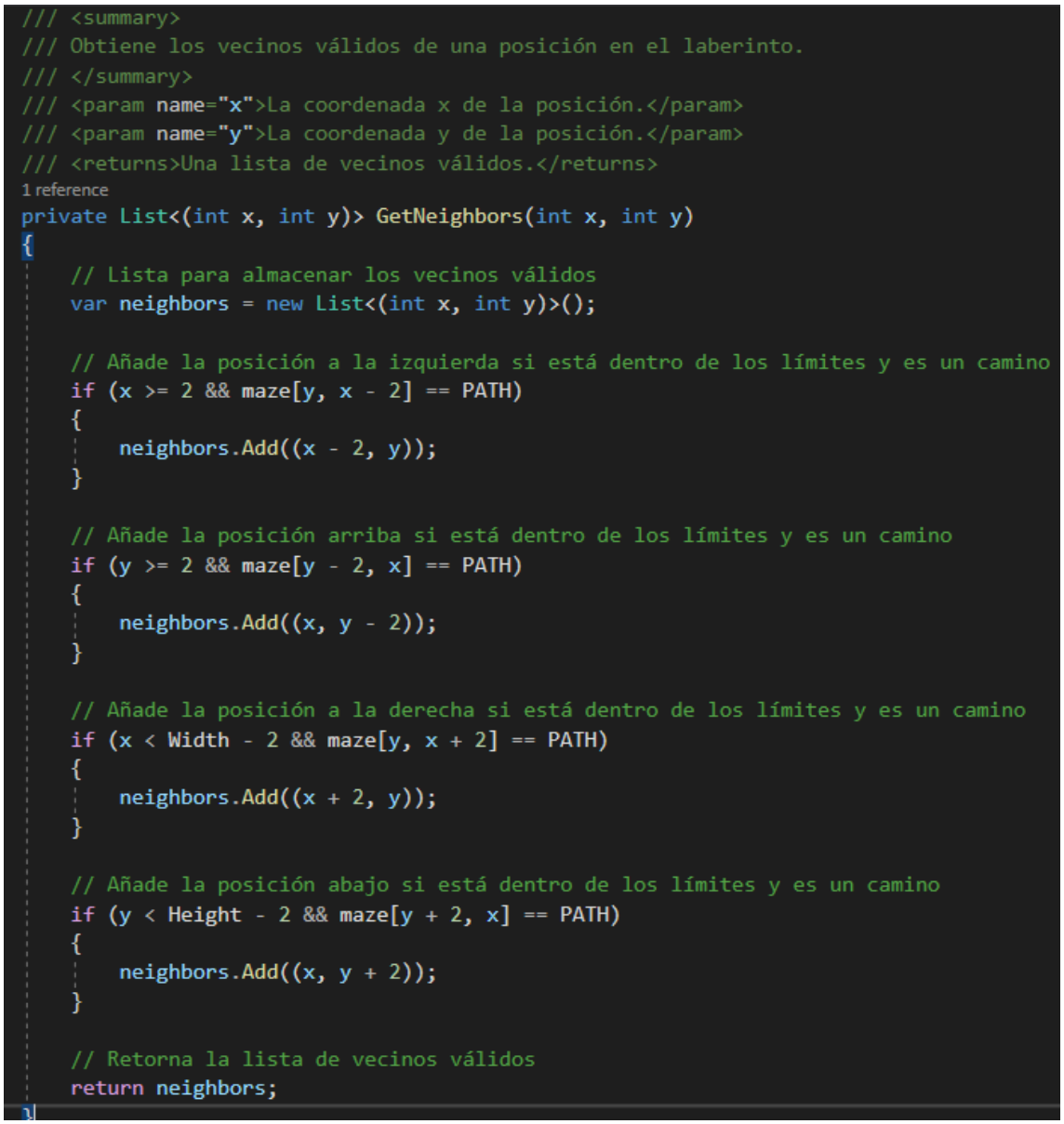
Esto apoyado por estos dos métodos:

```
/// <summary>
/// Añade posiciones de frontera para el algoritmo de generación del laberinto.
/// </summary>
/// <param name="x">La coordenada x de la posición.</param>
/// <param name="y">La coordenada y de la posición.</param>
/// <param name="frontier">La lista de posiciones de frontera.</param>
3 references
private void AddFrontier(int x, int y, List<(int x, int y)> frontier)
{
    // Añade la posición a la izquierda si está dentro de los límites y es una pared
    if (x >= 2 && maze[y, x - 2] == WALL)
    {
        frontier.Add((x - 2, y));
    }

    // Añade la posición arriba si está dentro de los límites y es una pared
    if (y >= 2 && maze[y - 2, x] == WALL)
    {
        frontier.Add((x, y - 2));
    }

    // Añade la posición a la derecha si está dentro de los límites y es una pared
    if (x < Width - 2 && maze[y, x + 2] == WALL)
    {
        frontier.Add((x + 2, y));
    }

    // Añade la posición abajo si está dentro de los límites y es una pared
    if (y < Height - 2 && maze[y + 2, x] == WALL)
    {
        frontier.Add((x, y + 2));
    }
}
```



```
/// <summary>
/// Obtiene los vecinos válidos de una posición en el laberinto.
/// </summary>
/// <param name="x">La coordenada x de la posición.</param>
/// <param name="y">La coordenada y de la posición.</param>
/// <returns>Una lista de vecinos válidos.</returns>
1 reference
private List<(int x, int y)> GetNeighbors(int x, int y)
{
    // Lista para almacenar los vecinos válidos
    var neighbors = new List<(int x, int y)>();

    // Añade la posición a la izquierda si está dentro de los límites y es un camino
    if (x >= 2 && maze[y, x - 2] == PATH)
    {
        neighbors.Add((x - 2, y));
    }

    // Añade la posición arriba si está dentro de los límites y es un camino
    if (y >= 2 && maze[y - 2, x] == PATH)
    {
        neighbors.Add((x, y - 2));
    }

    // Añade la posición a la derecha si está dentro de los límites y es un camino
    if (x < Width - 2 && maze[y, x + 2] == PATH)
    {
        neighbors.Add((x + 2, y));
    }

    // Añade la posición abajo si está dentro de los límites y es un camino
    if (y < Height - 2 && maze[y + 2, x] == PATH)
    {
        neighbors.Add((x, y + 2));
    }

    // Retorna la lista de vecinos válidos
    return neighbors;
}
```

Este algoritmo. Llamado algoritmo de Prim basado en laberinto fue proporcionado por una IA.

A pesar de este interesante proyecto hubo varios desafíos a los que me tuve que enfrentar, como la visibilidad del laberinto, que cuando se tenía que mostrar junto había una separación. Donde más trabajo pase haciendo este proyecto fue a la hora de configurar las trampas con las habilidades, porque habían habilidades que estaban relacionadas con otras habilidades o con las propias trampas, lo cual fue engorroso de solucionar. Los métodos MoveFicha

y Move, a veces no funcionaban correctamente haciendo que la ficha fuera a una posición no valida como las paredes.

Agradecimientos

Este proyecto fue realizado gracias al apoyo de personas muy importantes:

María del Carmen González Valera (mi madre)

Jessica Fernández Cabrera (mi pareja y futura esposa)