Burning Forest Simulator

Evaluation

2022 MOD002702 TRI1 F01CAM

Alan Nardo (2182389)

Table of Contents

Evaluation **1**

Class Diagrams 2

Tests 3

Appendix 1 (Source Code) 4

Map.cs 4

Cell.cs 8

Uitilities.cs 9

Terrain.cs 10

Wind.cs 11

EqualityComparer.cs 12

Appendix 2 (Instructions) 13

Appendix 3 (Algorithms) 14

Evaluation

The Burning Forest Simulator uses a 21x21 two-dimensional array created on startup to use as a model for displaying the forest. Trees are represented by green '&' symbols, a fire is represented by a red 'x' and an empty plot is replaced with a yellow underscore (_). A two-dimensional array was chosen because accessing elements by index is very fast. Each member of the array is a Cell class object, containing two properties character-type state (burning, tree or empty) and integer-type terrain (0, 1 or 2). Cells that are on fire have their coordinates stored in a List object of integer Tuples. A generic list was chosen over a linked list because the design of the project only requires insertion and removal of elements.

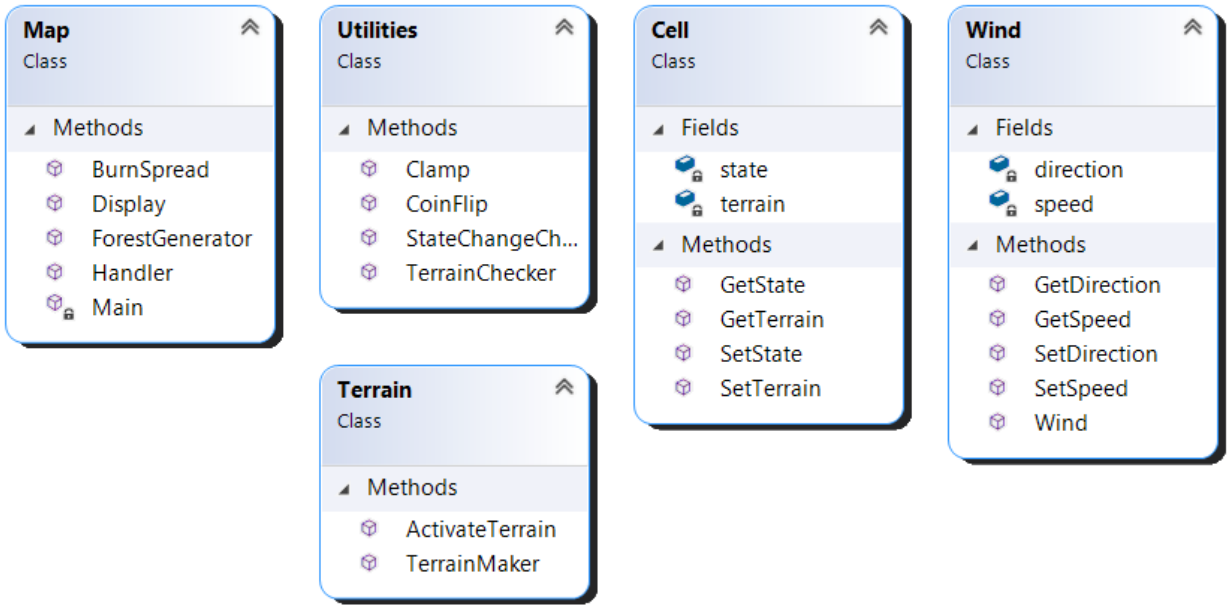
Classes store data of various aspects of the map. The optional Wind class has two properties, integer-type speed and character-type direction, which is one of the four cardinal directions. Wind properties are optional for the user to select when the simulation begins. The reason the Wind and Cell classes were designed as such is that their properties are used repeatedly to calculate the spread of the fire across the forest. These properties can be easily adjusted and accessed by using setter and getter methods, respectively.

An optional terrain setting (see **Error! Reference source not found.**) can be selected by the user when the simulation begins. The terrain can be either dry, which burns 100% of the time; wet, which never burns; or normal which has a 50% chance to catch fire.

The Map class is the main entry point for the application, where the simulation and all the reliant classes are initialized. As well a Utilities class contains many helper methods, such as the CoinFlip method which determines if a tree catches fire and the StateChangeChecker method which is used to handle the logic of fire spread on modified terrain or on regular terrain. The BurningChecker method in the Map class is the logical core of the program which updates the list of burning spots as well as the states of Cell objects as they burn out or catch fire.

This program is quite memory efficient as many of the class objects are created one time, only being updated as the simulation progresses. This prevents collisions that can occur, for example, when Random objects are created too fast in succession. By having a list of coordinate tuples, these coordinates can be used to quickly access the array location and update their states, saving on having to iterate through both levels of the array.

Class Diagrams



Created with Class Diagram plug-in for Visual Studio 2019.

Tests

Tests				
No.	**Input**	**Expected**	**Output**	**Comments**
1	Grid sequencing	A 21x21 grid of trees, with 10,10 burning	A 21x21 grid of trees, with 10,10 burning	Testing to make sure the grid generation function worked.
2	30 Random.Next()	Approx. 15 Even / 15 Odd RNG	Exactly 15 Even / 15 Odd RNG	Testing the distribution of randomly generation numbers to be approx. 50/50
3	Terrain values	Random cells with have dry or wet terrain	Cells have wet and dry terrain	Cells have terrain based on an RNG value (see Appendix 3)
4	Fire spread	Fire will not spread to wet and will spread to dry	Fire spreads to dry and does not spread to wet tiles	Works as expected
5	Wind	Wind will cause fire to spread to +1 tile	Fire spread to 1 tile only	Due to the nature of how the math is calculated, adding wind did not work and is currently omitted from the project unless time permits *Was able to implement effectively, and has variable wind speed*

Appendix 1 (Source Code)

Note: for readability purposes, some spacing or indentation may not match the actual .cs file, as submitted.

Map.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace forestListed
{
    class Map
    {
        static void Main(string[] args)
        {

            Console.WriteLine("########################################");
            Console.WriteLine("BURNING FOREST SIMULATOR by Alan Nardo");
            Console.WriteLine("########################################");
            Wind wind = new Wind();
            Cell[,] forestMap;
            Random randy = new Random();
            List<Tuple<int, int>> burnspots = new List<Tuple<int, int>>()
            {
                new Tuple<int,int> ( 10, 10 )
            };
            Terrain terrain = new Terrain(); //terrain generation.
            bool active = Terrain.ActivateTerrain();
            forestMap = ForestGenerator(randy, terrain, active);

            Console.WriteLine(
                 "Press W to activate terrain, any other key to continue without.");
            ConsoleKey response = Console.ReadKey().Key;
            Console.WriteLine();
            if (response == ConsoleKey.W)
            {
                wind = new Wind();
                wind.SetDirection(randy);
                wind.SetSpeed(randy);
            }

            Display(burnspots, forestMap, wind, active);
            Handler(burnspots, forestMap, randy, wind, active);
        } // end main

        public static Cell[,] ForestGenerator(
                Random rand, Terrain terr, bool isActive
        )
        {
            Cell[,] forestGrid = new Cell[21, 21];
```

```
            for (int i = 0; i < forestGrid.GetLength(0); i++)
            {
                for (int j = 0; j < forestGrid.GetLength(1); j++)
                {
                    forestGrid[i, j] = new Cell();
                    forestGrid[i, j].SetState('&');
                    if (isActive)
                    {
                        int terrain = terr.TerrainMaker(rand, i);
                        forestGrid[i, j].SetTerrain(terrain);
                    }
                }
            }

            //make sure centre point is on fire and not wet
            forestGrid[10, 10].SetState('x');
            forestGrid[10, 10].SetTerrain(0);

            return forestGrid;
        }
        public static void Display(
            List<Tuple<int, int>> fires,
            Cell[,] forest,
            Wind wind,
            bool active
            )
        {
            PaintMap();
            if (active)
            {
                Console.WriteLine("Terrain mods are on.");
            }
            else
            {
                Console.WriteLine("Terrain mods are off.");
            }
            if (wind.GetSpeed() > 0)
            {
                Console.WriteLine("Wind is blowing " + char.ToUpper(
                 wind.GetDirection()) + " at speed " + wind.GetSpeed() + ".");
            }
            else
            {
                Console.WriteLine("There is no wind.");
            }
            Console.WriteLine(fires.Count + " fires burning.");
            for (int i = 0; i < forest.GetLength(0); i++) //loop through outer
            {
                for (int j = 0; j < forest.GetLength(1); j++) //again on inner
                {
                    // a e s t h e t i c s
                    if (forest[i, j].GetState() == 'x')
                    {

                        Console.BackgroundColor = ConsoleColor.DarkYellow;
```

```
                    Console.ForegroundColor = ConsoleColor.DarkRed;
                }
                else if (forest[i, j].GetState() == '&')
                {
                    Console.ForegroundColor = ConsoleColor.DarkGreen;
                }
                else
                {
                    Console.ForegroundColor = ConsoleColor.DarkYellow;
                }
                if (forest[i, j].GetTerrain() == 1) //dry
                {
                    Console.BackgroundColor = ConsoleColor.Gray;
                }
                else if (forest[i, j].GetTerrain() == 2) //wet
                {
                    Console.BackgroundColor = ConsoleColor.Cyan;
                }
                else
                {
                    Console.BackgroundColor = ConsoleColor.Black;
                }
                Console.Write(string.Format("{0}", forest[i, j].GetState()));
            }

            Console.Write(Environment.NewLine);
        }

        void PaintMap()
        {
            for(int i = 0; i < fires.Count; i++)
            {
                forest[fires[i].Item1, fires[i].Item2].SetState('x');
            }
        }
    }//end display

    public static void Handler(
        List<Tuple<int, int>> fires,
        Cell[,] forest,
        Random rand,
        Wind wind,
        bool active
        )
    {
        Console.ForegroundColor = ConsoleColor.White;
        Console.WriteLine("Please press 'Enter' to continue...");
        ConsoleKeyInfo keyPress = Console.ReadKey();

        if (keyPress.Key == ConsoleKey.Enter)
        {
            Console.Clear();
            List<Tuple<int, int>> updatedList = BurnSpread(
                                                fires, forest, rand, wind);
            Display(updatedList, forest, wind, active);
```

```
            if (updatedList.Count == 0) //if they are the same, the game ends
            {
                active = false;
                Console.ForegroundColor = ConsoleColor.White;
                Console.WriteLine("The fire cannot spread anymore the sim is over
. Press R to restart, any other key to quit.");
                ConsoleKeyInfo option = Console.ReadKey();
                if (option.Key == ConsoleKey.R)
                {
                    Console.Clear(); //start a new game!
                    active = true;
                    bool terrain = Terrain.ActivateTerrain();
                    Cell[,] forestGrid =
                              ForestGenerator(rand, new Terrain(), terrain);
                    Console.WriteLine("#######################################");
                    Console.WriteLine("BURNING FOREST SIMULATOR by Alan Nardo");
                    Console.WriteLine("#######################################");
                    Display(updatedList, forest, wind, active);
                    Handler(updatedList, forest, rand, wind, active);
                }
                else
                {
                    Environment.Exit(0);
                }
            }
            Handler(updatedList, forest, rand, wind, active);
        }
    } // end Handler

    public static List<Tuple<int, int>> BurnSpread(
        List<Tuple<int, int>> burnzones,
        Cell[,] forest,
        Random rand,
        Wind wind
        )
    {
        List<Tuple<int, int>> newburns = new List<Tuple<int, int>>();

        if (burnzones.Count > 0)
        {
            for (int i = 0; i < burnzones.Count; i++)
            {
                IDictionary<Tuple<int, int>, char> returns =
                     new Dictionary<Tuple<int, int>, char>();


                int x = burnzones[i].Item1;
                int y = burnzones[i].Item2;

                int s = Utilities.Clamp(x + 1, 0, 21);
                int n = Utilities.Clamp(x - 1, 0, 21);
                int w = Utilities.Clamp(y - 1, 0, 21);
                int e = Utilities.Clamp(y + 1, 0, 21);
```

```
                int smax = Utilities.Clamp(s + 1, 0, 21);
                int nmax = Utilities.Clamp(n + 1, 0, 21);
                int wmax = Utilities.Clamp(w + 1, 0, 21);
                int emax = Utilities.Clamp(e + 1, 0, 21);

                switch (wind.GetDirection())
                {
                    case 's':
                        smax = Utilities.Clamp(smax + wind.GetSpeed(), 0, 21);
                        break;
                    case 'n':
                        nmax = Utilities.Clamp(nmax + wind.GetSpeed(), 0, 21);
                        break;
                    case 'e':
                        emax = Utilities.Clamp(emax + wind.GetSpeed(), 0, 21);
                        break;
                    case 'w':
                        wmax = Utilities.Clamp(wmax + wind.GetSpeed(), 0, 21);
                        break;
                }

                //burns to the north
                for (int d = n; d < nmax; d++)
                {
                    if (forest[d, y].GetState() == '&')
                    {
                        Tuple<int, int> north = new Tuple<int, int>(d, y);
                        forest[d, y].SetState(
                             Utilities.StateChangeCheck(forest[d, y], rand));
                        if (!returns.ContainsKey(north))
                        {
                            returns.Add(north, forest[d, y].GetState());
                        }
                    }
                }
                //burns to the west
                for (int c = w; c < wmax; c++)
                {
                    if (forest[x, c].GetState() == '&')
                    {
                        Tuple<int, int> west = new Tuple<int, int>(x, c);
                        forest[x, c].SetState(
                             Utilities.StateChangeCheck(forest[x, c], rand));
                        if (!returns.ContainsKey(west))
                        {
                            returns.Add(west, forest[x, c].GetState());
                        }
                    }
                }
                //burns to the south
                for (int a = s; a < smax; a++)
                {
                    if (forest[a, y].GetState() == '&')
                    {
                        Tuple<int, int> south = new Tuple<int, int>(a, y);
```

```
                            forest[a, y].SetState(
                                 Utilities.StateChangeCheck(forest[a, y], rand));
                            if (!returns.ContainsKey(south))
                            {
                                returns.Add(south, forest[a, y].GetState());
                            }
                        }
                    }
                    //burns to the east
                    for (int b = e; b < emax; b++)
                    {
                        if (forest[x, b].GetState() == '&')
                        {
                            Tuple<int, int> east = new Tuple<int, int>(x, b);
                            forest[x, b].SetState(
                                 Utilities.StateChangeCheck(forest[x, b], rand));
                            if (!returns.ContainsKey(east))
                            {
                                returns.Add(east, forest[x, b].GetState());
                            }
                        }
                    }
                    foreach (KeyValuePair<Tuple<int, int>, char> entry in returns)
                    {
                        if (entry.Value == 'x')
                        {
                            newburns.Add(entry.Key);
                        }
                    }
                    forest[x, y].SetState('_');
                    returns.Clear();
                }
            }
            var hashSet = new HashSet<Tuple<int, int>>(newburns);
            return hashSet.ToList();
        }
    }
}
```

Cell.cs

```
namespace forest
{
    internal class Cell
    {
        private char state = '&';
        private int terrain = 0;

        public void SetState(char state)
        {
            this.state = state;
        }

        public char GetState()
        {
            return state;
        }
        public void SetTerrain(int terrain)
        {
            this.terrain = terrain;
        }
        public int GetTerrain()
        {
            return terrain;
        }
    }
}
```

Uitilities.cs

```
using System;

namespace forestListed
{
    internal class Utilities
    {
        /* THE TERNINATORS */
        public static char CoinFlip(Random rnd, char currentState)
        {
            return rnd.Next() % 2 == 0 ? 'x' : currentState;
        }
        public static char TerrainChecker(int terrainType)
        {
            return terrainType == 1 ? 'x' : '&';
        }
        public static char StateChangeCheck(Cell cell, Random rand)
        {
            return cell.GetTerrain() == 0
                ? CoinFlip(rand, cell.GetState())
                : TerrainChecker(cell.GetTerrain());
        }
        public static int Clamp(int value, int min, int max)
        {
            return (value < min) ? min : (value > max) ? max : value;
        }
    }
}
```

Terrain.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace forestListed
{
    internal class Terrain
    {
        public int TerrainMaker(Random rand, int index)
        {
            int terrain = 0;
            int roll = rand.Next(21);

            if (roll == index)
            {
                switch (roll % 2)
                {
                    case 1:
                        terrain = 1; //dry
                        return terrain;
                    case 0:
                        terrain = 2; //wet
                        return terrain;
                }
            }
         return terrain;
        }

        public static bool ActivateTerrain()
        {
            Console.WriteLine("Press Y to activate terrain,
                                any other key to continue without.");
            ConsoleKey reply = Console.ReadKey().Key;
            Console.WriteLine();
            if (reply == ConsoleKey.Y)
            {
                return true;
            }
            return false;
        } // end ActivateTerrain
    }
}
```

Wind.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace forestListed
{
    internal class Wind
    {
        private int speed = 0;
        private char direction;

        public Wind() { }
        public void SetSpeed(Random rand)
        {
            speed = rand.Next(1, 4);
        }

        public void SetDirection(Random rand)
        {
            IDictionary<int, char> directions = new Dictionary<int, char>()
            {
                {0, 'n' },
                {1, 's' },
                {2, 'e' },
                {3, 'w' },
            };

            int roll = rand.Next(4);
            direction = directions[roll];
        }

        public char GetDirection()
        {
            return direction;
        }

        public int GetSpeed()
        {
            return speed;
        }
    }
}
```

EqualityComparer.cs

```
using System;
using System.Collections.Generic;

namespace forestListed
{
    /*
     * Credit to user Viktor Lova @ Stackoverflow:
       https://stackoverflow.com/a/17275935/11790979
    */
    class SameTuplesComparer<T1, T2> : EqualityComparer<Tuple<T1, T2>>
    {
        public override bool Equals(Tuple<T1, T2> t1, Tuple<T1, T2> t2)
        {
            return t1.Item1.Equals(t2.Item1) && t1.Item2.Equals(t2.Item2);
        }


        public override int GetHashCode(Tuple<T1, T2> t)
        {
            return base.GetHashCode();
        }
    }
}
```

Appendix 2 (Instructions)

To begin the simulation, first open the .exe file. The user is prompted for two inputs, sequentially, to activate terrain and wind mods or not. These mods affect the rate and pattern of how the fire spreads through the forest. Terrain mods give random elements a dry or wet terrain based on the algorithm (see Appendix 3 (Algorithms)), or a wind mod which sets a random wind speed and direction, respectively, if elected. The simulator will always begin from the middle element, (10,10) and has a 50% chance to spread to each of its neighbors, assuming mods are inactive. To spread the fire, press 'Enter' when prompted by the console to allow the burn to spread. This continues until there are no more trees with fires adjacent, or the entire forest has burned down. When the simulation ends, the user can either restart a new simulation by pressing the 'R' key or quit the simulation, by pressing any other key.

Appendix 3 (Algorithms)

Terrain Generation Algorithm (Terrain.TerrainMaker)

The algorithm for generating terrain values is rather simple and costs very low overhead due to the simple mathematical calculations it performs. The algorithm requires two inputs, a Random object `rand` and an integer value `index`. `Rand` is used to generate a pseudo-random integer variable no higher than 20, the maximum length of the arrays that make up the forest. This value is stored in the variable `roll`. If `roll` is equal to `index`, then a switch statement begins to test the value of the product of `roll` and `index`, modulo 2. If the remainder of this value is 1, the terrain is given a value 1, for drier terrain, and if that value is 0, the terrain is given a value 2 for wetter terrain.

The terrain styles affect burn rate in the forest. A dry plot will catch fire 100% of the time if there is a fire adjacent to it. A wet plot will have a 0% chance to catch fire, no matter how many fires are adjacent to it. This allows to impact the spreading of the forest fire in a small way, by being neither too common nor too overpowering in terms of how the burn rate is affected.

Burn Spread Algorithm (Utilities.CoinFlip)

The coinflip algorithm is a very simple approach to generating approximately 50/50 chance of occurrences. This is only one of a multitude of ways to get a ~ 50% outcome. A pseudo-random integer is created by calling rnd.Next(), from which modulo 2 is taken. If this operation is even (equal to 0), then the state of the cell is changed to burning (‘x’), otherwise the tree remains.

EqualityComparer (EqualityComparer.cs)

The EqualityComparer is a hashset algorithm that performs rapid deduplication on an enumerable object. Having too many objects in the list can cause a stack overflow, so in order to prevent this, each coordinate pair Tuple<int, int> must be ensured to only be in the list one time. This method gives each element a hash based on its value and then compares that has against other values, if it exists it is not added to the list. If it does not, it will be added. Since this method is only performed once at the end of the BurningChecker method, it is quite memory efficient.

Wind (Wind.cs)

The wind is a modification which can be enabled by the user on startup. It generates a random windspeed from 1 to 3, which is how many tiles from the fire it may possibly spread and a random direction, based off a key-value pair in a dictionary, which selects the key by generating a random integer 0 to 3.