CSE 3541 Lab 2 Report
Yihone Chu
Profesor Shareef Naeem
9/24/2024

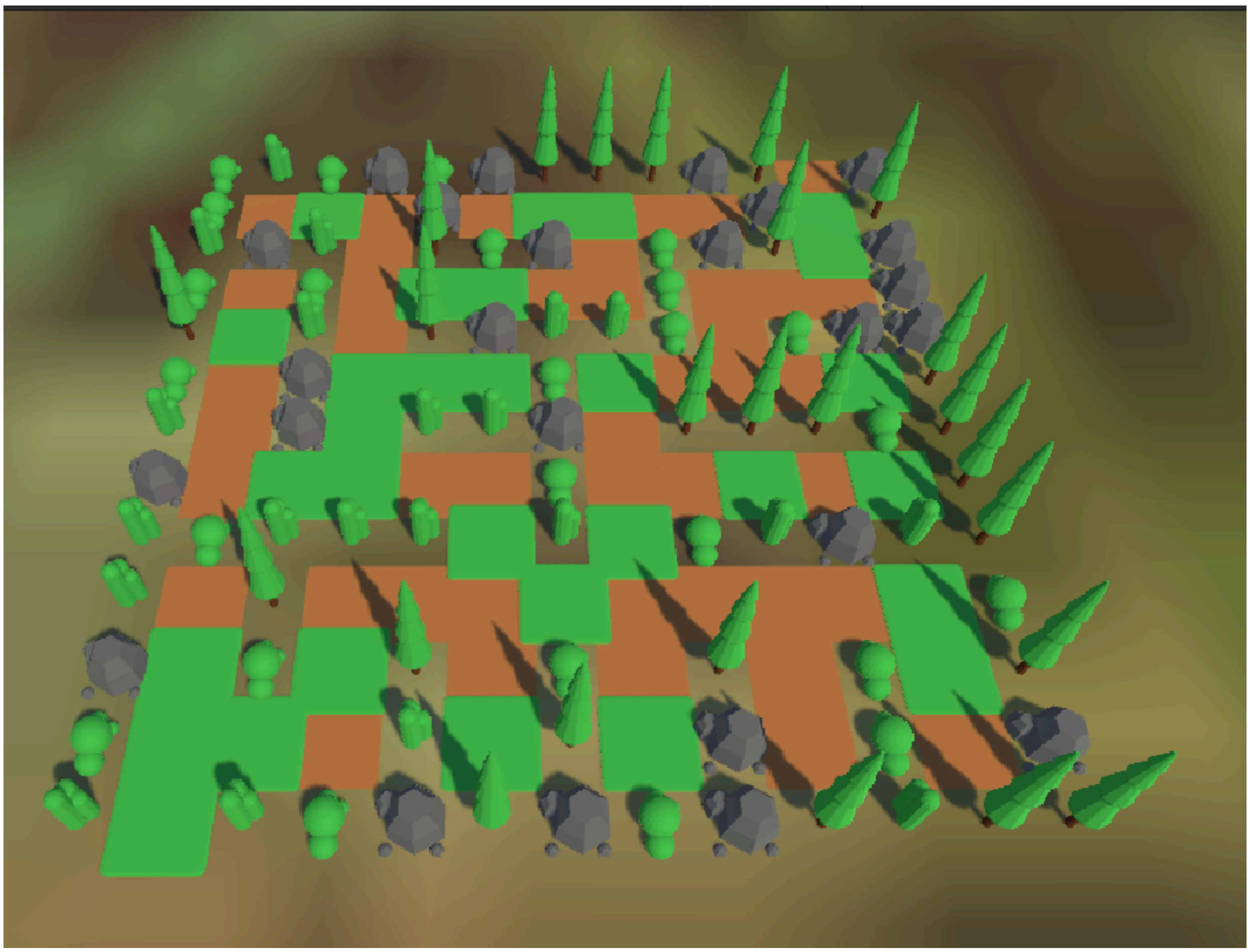

Figure 1. Maze created using Prim's Algorithm

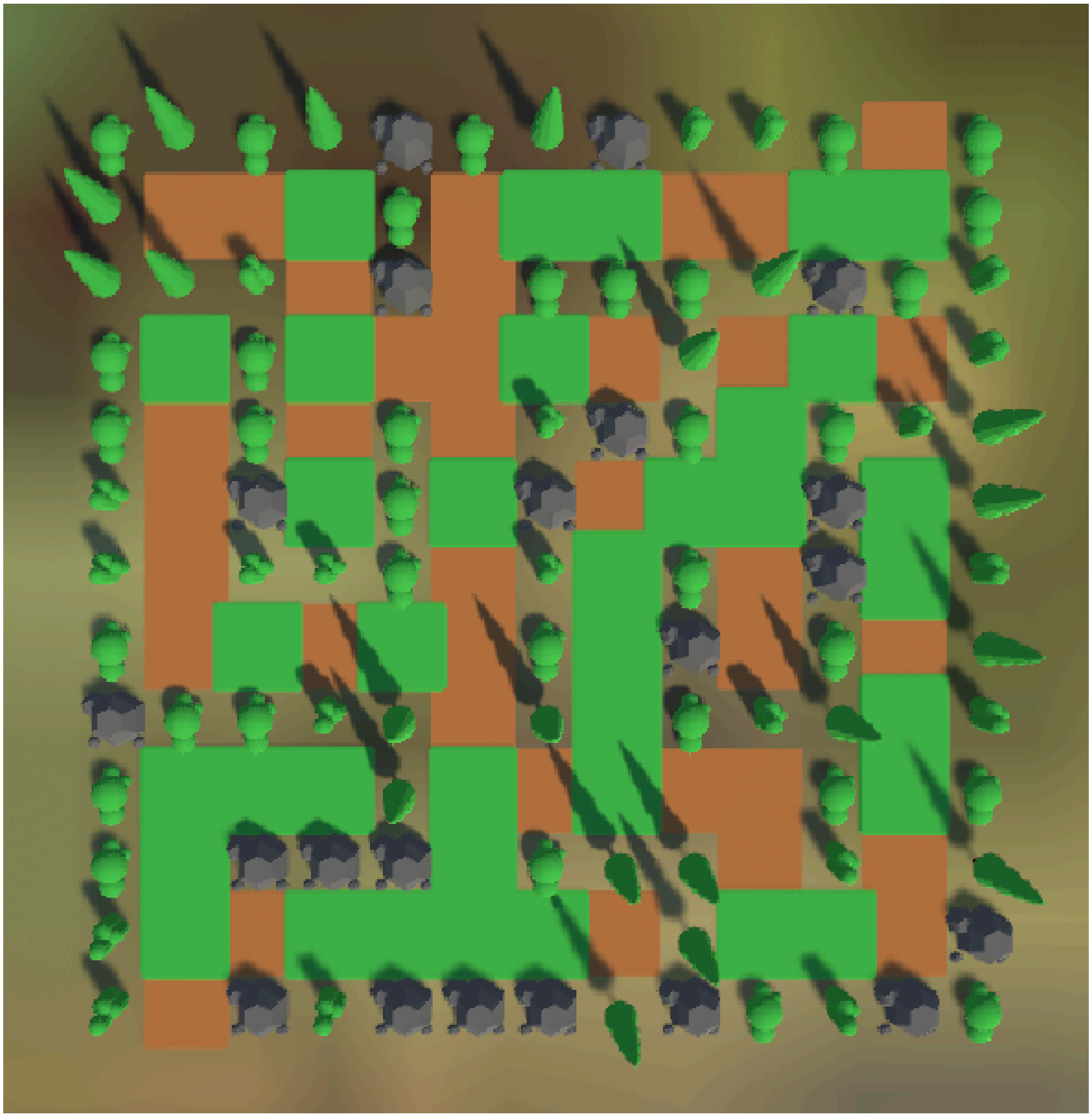

Figure 2. Birds Eye View of Maze created using Prim's Algorithm

In the field of computer science, algorithms play a crucial role in solving complex problems efficiently. One such problem is the generation of mazes, which has applications in game design, robotics, and artificial intelligence. This report presents the implementation of a maze generation algorithm using Prim's algorithm, which creates a unique and traversable maze structure.

To enhance user interaction with the generated maze, the report also explores a rotation feature that employs affine transformations. This allows users to dynamically manipulate the maze's orientation, improving navigation and engagement within the game environment. Together, these components demonstrate the integration of algorithmic design and interactive gameplay.

I did not have any discussions at all regarding the lab. However, I did extensively look at documentation as well as some instructional videos on YouTube.

In this lab, I have implemented a maze generation algorithm using Prim's algorithm. Additionally, I have developed a rotation feature that allows for interactive manipulation of the maze using affine transformations. This includes the GridTraversal<T> class for maze generation and the RotateMaze class for user-controlled rotation. I also added prefabs to enhance the visual appearance of the maze.

```csharp
// Generates a maze starting from the specified cell using Prim's algorithm.
1 reference
public IEnumerable<((int Row, int Column) From, (int Row, int Column) To)> GenerateMaze(int startRow, int startColumn)
{
    HashSet<(int Row, int Column)> visited = new HashSet<(int Row, int Column)>();
    HashSet<(int Row, int Column)> unvisited = new HashSet<(int Row, int Column)>();

    // Populate the Unvisited set with all vertices in the grid
    PopulateUnvisited(unvisited);

    var start = (startRow, startColumn);
    unvisited.Remove(start);
    visited.Add(start);

    List<((int Row, int Column) From, (int Row, int Column) To)> eligibleEdges = new List<((int Row, int Column) From, (int Row, int Column) To)>();

    // Add the starting cell's neighbors' edges to the eligible edges
    foreach (var neighbor in grid.Neighbors(startRow, startColumn))
    {
        if (unvisited.Contains(neighbor))
        {
            eligibleEdges.Add((start, neighbor));
        }
    }

    while (unvisited.Count > 0 && eligibleEdges.Count > 0)
    {
        // Select a random edge
        var randomEdge = eligibleEdges[new Random().Next(eligibleEdges.Count)];
        yield return randomEdge;

        // Mark the new vertex as visited
        var (from, to) = randomEdge;
        visited.Add(to);
        unvisited.Remove(to);

        // Update eligible edges with the neighbors of the newly visited vertex
        foreach (var neighbor in grid.Neighbors(to.Row, to.Column))
        {
            if (unvisited.Contains(neighbor))
            {
                eligibleEdges.Add((to, neighbor));
            }
        }

        // Remove edges that connect to already visited vertices
        eligibleEdges.RemoveAll(edge => visited.Contains(edge.To));
    }
}

1 reference
private void PopulateUnvisited(HashSet<(int Row, int Column)> unvisited)
{
    // Loop through all cells in the grid
    for (int row = 0; row < grid.NumberOfRows; row++)
    {
        for (int column = 0; column < grid.NumberOfColumns; column++)
        {
            // Add each vertex (row, column) to the Unvisited set
            unvisited.Add((row, column));
        }
    }
}
```

Figure 3. Implementation of Prim's Algorithm for Maze Generation

In this implementation, Ihave created a GridTraversal<T> class to generate mazes using Prim's algorithm, a greedy algorithm typically used for finding minimum spanning trees but adapted here for maze generation. The class is designed to be flexible and operates on any grid structure that implements the IGridGraph<T> interface. This modular design allows for reuse in different contexts, making the maze generation algorithm more versatile.

At the core of the maze generation process is the GenerateMaze method, which begins by initializing two sets: visited and unvisited. The visited set contains cells that are already part of the maze, while the unvisited set holds the remaining cells that have not yet been connected. The algorithm starts at a given cell, marks it as visited, and adds its neighbors to a list of eligible edges, which represent possible connections to new cells.

Prim's algorithm then proceeds by selecting a random edge from the eligible edges, connecting the current maze to a new, unvisited cell. This process repeats iteratively, expanding the maze by marking new cells as visited and removing them from the unvisited set. As new cells are connected, their neighboring cells are added to the list of eligible edges. To maintain the integrity of the maze and prevent cycles, edges leading to already visited cells are removed.

An additional private method, PopulateUnvisited, ensures that all cells in the grid are initially added to the unvisited set. This method iterates through every row and column of the grid, marking each cell as unvisited, thus setting the stage for the algorithm to explore and connect the maze's cells.

```csharp
public class RotateMaze : MonoBehaviour
{
    public GameObject createdMaze; // Reference to your CreatedMaze object
    public float rotationSpeed = 100f; // Speed of rotation
    private bool isRotating = false; // Toggle for rotation
    private Controls controls;

    // Reference to your LevelGeneration script
    public LevelGeneration levelGeneration;

    private Vector3 mazeCenter; // Store the calculated center of the maze

    Unity Message | 0 references
    private void Awake()
    {
        controls = new Controls();
        controls.MazeControls.Enable(); // Enable the action map
        controls.MazeControls.ToggleRotation.performed += OnToggleRotation; // Subscribe to the action

        // Calculate the maze center once at the beginning
        mazeCenter = CalculateCenter();
    }

    Unity Message | 0 references
    private void OnDisable()
    {
        controls.MazeControls.ToggleRotation.performed -= OnToggleRotation; // Unsubscribe to avoid memory leaks
        controls.Disable(); // Disable the action map
    }

    2 references
    private void OnToggleRotation(InputAction.CallbackContext context)
    {
        isRotating = !isRotating; // Toggle the rotation state
    }

    Unity Message | 0 references
    private void Update()
    {
        if (isRotating && createdMaze != null)
        {
            // Rotate around the center of the maze
            createdMaze.transform.RotateAround(mazeCenter, Vector3.up, rotationSpeed * Time.deltaTime);
        }
    }

    1 reference
    private Vector3 CalculateCenter()
    {
        // Calculate the center based on level generation parameters
        float centerX = (levelGeneration.NumberOfColumns * levelGeneration.CellWidth) / 2f;
        float centerZ = (levelGeneration.NumberOfRows * levelGeneration.CellHeight) / 2f;

        return new Vector3(centerX, 0, centerZ);
    }
}
```

Figure 4. Rotation Feature using Affine Transformations

In this implementation, you have created a RotateMaze class that enables the user to rotate the generated maze in real-time using Unity's 3D engine. This feature enhances user interactivity by allowing dynamic manipulation of the maze's orientation, thereby improving the overall gameplay experience. The class makes use of affine transformations, specifically rotation, to rotate the maze around its calculated center point.

At the heart of the rotation feature is the Update method, which continuously checks whether the user has toggled the rotation feature and whether the maze object exists. When the isRotating flag is set to true, the createdMaze object rotates around a central point. The RotateAround method in Unity is used to achieve this effect, where the maze rotates around its center at a speed defined by rotationSpeed and based on the time elapsed (Time.deltaTime). This allows for smooth and continuous rotation while maintaining a consistent frame rate.

The rotation behavior is controlled via user input. The Awake method sets up the user controls by subscribing to a toggle input (ToggleRotation) using Unity's new input system. When the toggle action is performed, the OnToggleRotation method is called, which flips the isRotating boolean, turning the rotation on or off. This gives the user full control over whether or not the maze is rotating.

Additionally, the class calculates the center of the maze dynamically. In the CalculateCenter method, the center is computed based on the number of rows and columns in the maze, along with the dimensions of individual cells. The maze center is stored as a Vector3 (representing its x, y, and z coordinates), ensuring that all rotations occur smoothly around the actual middle of the maze.

```
CLASS GridTraversal

  INITIALIZE grid (interface for the grid graph)

  FUNCTION GenerateMaze(startRow, startColumn):

    visited = empty set           // To track visited vertices

    unvisited = empty set         // To track unvisited vertices

        CALL PopulateUnvisited(unvisited)    // Populate unvisited with all grid cells

    start = (startRow, startColumn)      // Define starting cell

    REMOVE start FROM unvisited

    ADD start TO visited

    eligibleEdges = empty list       // To store eligible edges

    // Add edges from the starting cell's neighbors to eligibleEdges

    FOR each neighbor IN grid.Neighbors(startRow, startColumn):

      IF unvisited CONTAINS neighbor:

        ADD (start, neighbor) TO eligibleEdges

    // While there are unvisited cells and eligible edges to process

    WHILE unvisited is not empty AND eligibleEdges is not empty:

      randomEdge = SELECT random edge from eligibleEdges
```

```
        YIELD randomEdge        // Return the selected edge

        (from, to) = randomEdge

        ADD to TO visited       // Mark the new vertex as visited

        REMOVE to FROM unvisited

        // Add neighbors of the newly visited vertex to eligibleEdges

        FOR each neighbor IN grid.Neighbors(to.Row, to.Column):

            IF unvisited CONTAINS neighbor:

                ADD (to, neighbor) TO eligibleEdges

        // Remove edges that connect to already visited vertices

        REMOVE all edges FROM eligibleEdges WHERE edge.To IN visited


FUNCTION PopulateUnvisited(unvisited):

    FOR row FROM 0 TO grid.NumberOfRows:

        FOR column FROM 0 TO grid.NumberOfColumns:

            ADD (row, column) TO unvisited    // Add each cell to the unvisited set
```

The running time is O(nlogn) due to the following reasons.

Populating the Unvisited Set:

- The function PopulateUnvisited() loops through each cell in the grid. If there are n total cells in the grid (where n is the number of rows multiplied by the number of columns), this step takes O(n) time.

2. Handling Eligible Edges:
   - During the maze generation, the algorithm iterates while there are unvisited cells and eligible edges. In each iteration, it selects a random edge from the eligibleEdges list and updates the neighbors.
   - In the worst case, each cell could have up to 4 neighbors (for a grid), so the number of edges can grow as O(n).
3. Edge Selection (Random Edge Selection):
   - The selection of a random edge from the list takes O(1) time on average. However, adding or removing edges from the list requires more consideration.
   - Removing edges from the list involves checking if the edge connects to a visited cell, I used a simple list and linear search. It would take O(n) in the worst case.
4. Neighbor Updates and Edge Removal:
   - Each time an edge is processed, the algorithm must update the neighbors of the newly visited cell. This operation happens O(n) times (once for each cell), and for each update, checking and modifying eligible edges takes)O(n) time.

In terms of hierarchy I feel it would be really helpful if CreatedMaze had its own variables for the NumberOfColumns, NumberOfRows, CellWidth, and CellHeight. I had to work around this by having an instance of LevelGeneration within my RotateMaze script as well as create public variables within the LevelGeneration so that I could calculate the center of the Maze.

In addition, I feel like it would have been nice to know what else was going on in the code base. We were given a lot of code and it was hard to understand what the code was doing sometimes. Additional comments on the provided code would be really nice.

I did not complete any of the extra credit tasks.

The lab is moderately hard. I felt like the hardest part was the rotations. I felt Prim's Algorithm was straightforward and all I had to do was find the edge to yield return.