ДОДАТОК А

Графічний матеріал кваліфікаційної роботи

Харківський національний університет радіоелектроніки

МЕТОДИ ГЕНЕРАЦІЇ ТА ПРОХОДЖЕННЯ ЛАБІРИНТІВ

Виконав:
магістрант групи СПм-23-1 Вітко В. О.

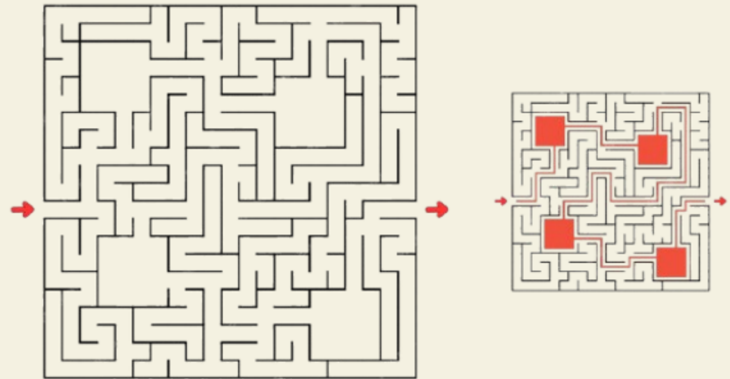
Науковий керівник:
доцент каф. ЕОМ Іващенко Г. С.

Харківський національний університет радіоелектроніки
Кафедра ЕОМ

2

Актуальність проблеми

- Ігрова індустрія
- Робототехніка
- Теорія графів
- Інженерія



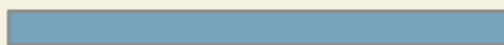
3

Класифікація лабіринтів

4

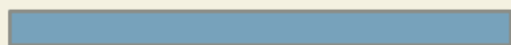
Аналіз існуючих рішень

Алгоритми генерації лабіринтів:



- Метод бінарного дерева
- Алгоритм Олдоса-Бродера
- Алгоритм Еллера

Алгоритми пошуку шляху:



- Пошук у ширину
- Алгоритм Дейкстри
- Алгоритм А*

5

Постановка задачі

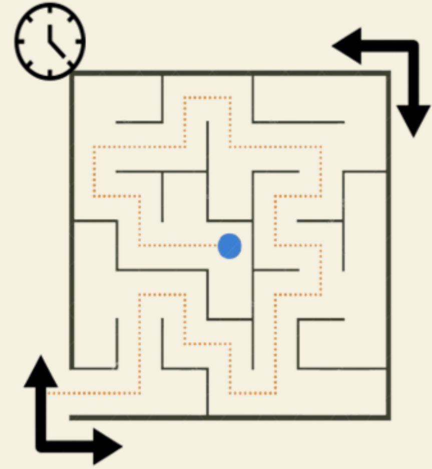
- Дослідження існуючих методів генерації лабіринтів та пошуку шляху
- Реалізація обраних алгоритмів для генерації та вирішення лабіринтів
- Розробка тестового програмного середовища для проведення досліджень
- Аналіз ефективності роботи алгоритмів генерації та проходження лабіринтів

6

Технології розробки тестового програмного середовища

7

Обрані умови аналізу



8

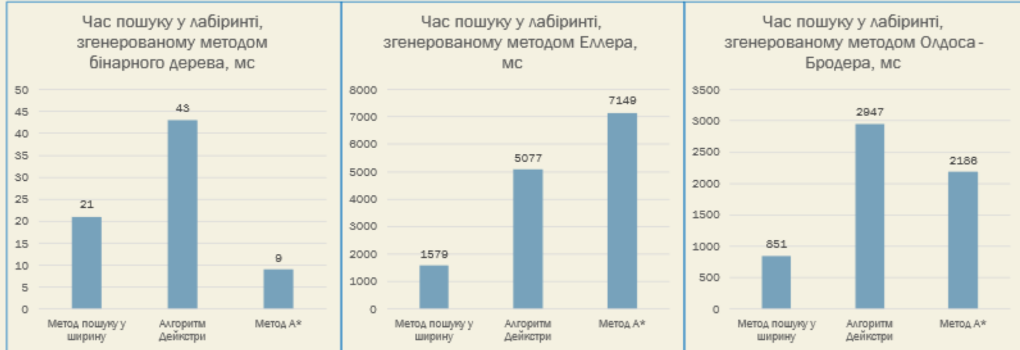
Аналіз алгоритмів генерації лабіринтів

Алгоритм генерації	Час генерації лабіринтів різної розмірності, мс		
	1000x1000	10000x10000	50000x50000
Метод бінарного дерева	13	1386	24256
Алгоритм Еллера	150	61527	1055954
Алгоритм Олдоса-Бродера	397	80767	2068884

9

Пошук шляху

	Час пошуку при використанні різних алгоритмів генерації лабіринту, мс		
Алгоритм пошуку шляху	Алгоритм бінарного дерева	Алгоритм Еллера	Алгоритм Олдоса-Бродера
Метод пошуку у ширину	21	1579	852
Алгоритм Дейкстри	43	5077	2947
Метод А*	9	7149	2186



10

Пошук шляху

	Час пошуку при використанні різних алгоритмів генерації лабіринту, мс		
Алгоритм пошуку шляху	Алгоритм бінарного дерева	Алгоритм Еллера	Алгоритм Олдоса-Бродера
Метод пошуку у ширину	2841	3738884	3940872
Алгоритм Дейкстри	5913	7852283	8754335
Метод А*	979	15753157	16053608

11

Пошук шляху

Алгоритм пошуку шляху	Час, мс
Метод пошуку у ширину	19645
Алгоритм Дейкстри	40627
Метод А*	16052

12

Висновки

- Реалізовані алгоритми генерації та пошуку шляху можуть бути використані для вирішення сучасних задач, потребуючих генерації та вирішення лабіринтів у галузі робототехніки, ігрового дизайну або оптимізації маршрутів.
- Генерація методом бінарного дерева створює лабіринти низької складності за малий час. Алгоритм Олдоса-Бродера потребує найбільшу кількість часу для виконання для генерації лабіринту
- Досліджено вплив застосованого алгоритму генерації на час пошуку шляху. Метод пошуку у ширину витрачає найменшу кількість часу для вирішення лабіринтів, згенерованих методамиЕллера та Олдоса-Бродера. Алгоритм А* є найефективнішим для лабіринтів, згенерованих методом бінарного дерева
- Результати роботи були представлені у рамках 11 та 12 міжнародної науково-технічної конференції "Проблеми інформатизації"

ДОДАТОК Б

Вихідний код застосунку

Б.1 Реалізація алгоритмів генерації лабіринту

Б.1.1 Алгоритм бінарного дерева

```
using UnityEngine;

//Binary Tree Genetate Algorithm
public class BinaryTreeMazeGenerator : IMazeGenerator
{
    // Generate alrogithm
    public byte[,] GenerateMaze(ref int width, ref int height)
    {

        // Reducing Width and Height to Odd Values
        if (width % 2 == 0) width++;
        if (height % 2 == 0) height++;

        // Creating an array for the labyrinth
        byte[,] maze = new byte[width, height];
        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < height; y++)
            {
                maze[x, y] = 1;
            }
        }

        // Generating
        for (int x = 2; x < height; x+=2)
        {
            for (int y = 0; y < width-1; y+=2)
            {
                maze[x, y] = 0;
                if (Random.Range(0, 2) == 0)
                {
                    maze[x, y + 1] = 0; // Open the passage on
the right
                }
                else
                {
                    maze[x - 1, y] = 0; // Open the passage on
the top
                }
```

```
            }
        }

        // Removing the walls at the edges
        for (int i = 0; i < height; i++) maze[0, i] = 0;
        for (int i = 1; i < width; i++) maze[i, width - 1] = 0;

        return maze;
    }
}
```

Б.1.2 Алгоритм Олдоса-Бродера

```
//Oldus Broder Genetate Algorithm

public class AldousBroderMazeGenerator : IMazeGenerator
{
    // Generate alrogithm
    public byte[,] GenerateMaze(ref int width, ref int height)
    {
        // Reducing Width and Height to Odd Values
        if (width % 2 == 0) width++;
        if (height % 2 == 0) height++;

        // Creating an array for the labyrinth
        byte[,] maze = new byte[width, height];
        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < height; y++)
            {
                maze[x, y] = 1;
            }
        }

        // Counting the total number of cells in the maze
        int totalCells = (width / 2 + 1) * (height / 2 + 1);
        int visitedCells = 1;

        // Determining a random starting position
        int currentX = UnityEngine.Random.Range(0, width / 2) *
2;
        int currentY = UnityEngine.Random.Range(0, height / 2) *
2;
        maze[currentX, currentY] = 0;

        // Directions
        int[][] directions = new int[][]
        {
            new int[] { 0, -2 }, // up
            new int[] { 0, 2 },  // down
            new int[] { -2, 0 }, // left
```

```
            new int[] { 2, 0 }   // right
        };

        // Execution of the algorithm before visiting all cells
        while (visitedCells < totalCells)
        {
            // Random direction
            int[] direction =
directions[UnityEngine.Random.Range(0, directions.Length)];
            int nextX = currentX + direction[0];
            int nextY = currentY + direction[1];

            // Is the cell in Bounds?
            if (IsInBounds(nextX, nextY, width, height))
            {
                // Is the cell visited?
                if (maze[nextX, nextY] == 1)
                {
                    // Remove the wall between the current and
next cells
                    maze[currentX + direction[0] / 2, currentY +
direction[1] / 2] = 0;

                    // Mark the next cell as visited
                    maze[nextX, nextY] = 0;
                    visitedCells++;
                }

                // Move to the next cell
                currentX = nextX;
                currentY = nextY;
            }
        }

        return maze;
    }

    // Checking if the coordinates are inside the boundaries
    private bool IsInBounds(int x, int y, int width, int height)
    {
        return x >= 0 && x < width && y >= 0 && y < height;
    }
}
```

Б.1.3 Алгоритм Еллера

```
using System.Collections.Generic;

//Eller Genetate Algorithm
public class EllerMazeGenerator : IMazeGenerator
{
```

```
    private System.Random isConnection = new System.Random();

    public byte[,] GenerateMaze(ref int width, ref int height)
    {
        // Reducing Width and Height to Odd Values
        if (width % 2 == 0) width++;
        if (height % 2 == 0) height++;

        // Creating an array for the labyrinth
        byte[,] maze = new byte[height, width];
        for (int y = 1; y < height; y += 2)
        {
            for (int x = 1; x < width; x += 2)
            {
                maze[y, x] = 1;
            }
        }

        // Initial structure of sets for rows
        int[] sets = new int[(width / 2) + 1];
        int nextSet = 1;

        // Processing each row of the maze
        for (int y = 0; y < height; y += 2)
        {
            // Create new sets if needed
            for (int x = 0; x < sets.Length; x++)
            {
                if (sets[x] == 0) sets[x] = nextSet++;
            }

            // Creating horizontal connections
            for (int x = 0; x < sets.Length - 1; x++)
            {
                if (isConnection.Next(2) == 1 || sets[x] ==
sets[x + 1])
                {
                    // Put up a wall
                    maze[y, (x * 2) + 1] = 1;
                }
                else
                {
                    // Combine sets
                    int targetSet = sets[x + 1];
                    for (int i = 0; i < sets.Length; i++)
                    {
                        if (sets[i] == targetSet) sets[i] =
sets[x];
                    }
                }
            }

            // If this is the last line, join all sets
```

```
            if (y + 2 >= height)
            {
                for (int x = 0; x < sets.Length - 1; x++)
                {
                    if (sets[x] != sets[x + 1])
                    {
                        maze[y, (x * 2) + 1] = 0; // Remove the
wall
                        int targetSet = sets[x + 1];
                        for (int i = 0; i < sets.Length; i++)
                        {
                            if (sets[i] == targetSet) sets[i] =
sets[x];
                        }
                    }
                }
                break; // End the row processing loop
            }

            // Create vertical connections
            var connectedSets = new HashSet<int>();
            for (int x = 0; x < sets.Length; x++)
            {
                if (isConnection.Next(2) == 0 ||
connectedSets.Contains(sets[x]))
                {
                    // Put up a wall
                    maze[y + 1, x * 2] = 1;
                }
                else
                {
                    connectedSets.Add(sets[x]); // Guarantee
connection for a set
                }
            }

            // Guarantee at least one connection for a set
            foreach (int set in sets)
            {
                if (!connectedSets.Contains(set))
                {
                    for (int x = 0; x < sets.Length; x++)
                    {
                        if (sets[x] == set)
                        {
                            maze[y + 1, x * 2] = 0; // Remove
the wall
                            connectedSets.Add(set);
                            break;
                        }
                    }
                }
            }
```

```
            // Update sets for the next row
            for (int x = 0; x < sets.Length; x++)
            {
                if (maze[y + 1, x * 2] == 1) sets[x] = 0; //
Cells with a border get new sets
            }
        }

        return maze;
    }
}
```

Б.2 Реалізація алгоритмів пошуку шляху

Б.2.1 Пошук у ширину

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

// Breadth First Search Algorithm
public class BreadthFirstSearch : IPathfindingAlgorithm
{
    // Generated maze
    public List<Vector2Int> FindPath(byte[,] maze)
    {
        // Size of the labyrinth
        int rows = maze.GetLength(0);
        int cols = maze.GetLength(1);

        // Announcement of start and end points
        Vector2Int start = Vector2Int.zero;
        Vector2Int goal = Vector2Int.zero;

        // Determining the start and finish points
        for (int x = 0; x < rows; x++)
        {
            for (int y = 0; y < cols; y++)
            {
                if (maze[x, y] == 2)
                {
                    start = new Vector2Int(x, y);
                }
                else if (maze[x, y] == 3)
                {
                    goal = new Vector2Int(x, y);
                }
            }
        }
```

```
        // Check for the presence of a start and end point
        if (maze[start.x, start.y] != 2 || maze[goal.x, goal.y]
!= 3)
        {
            Debug.LogError("Start or end point not found!");
            return new List<Vector2Int>();
        }

        // Checking the boundaries
        bool IsInBounds(Vector2Int pos) => pos.x >= 0 && pos.x <
rows && pos.y >= 0 && pos.y < cols;

        // Shifts for neighbors
        Vector2Int[] directions = { Vector2Int.up,
Vector2Int.down, Vector2Int.left, Vector2Int.right };

        // Queue for BFS
        Queue<Vector2Int> queue = new Queue<Vector2Int>();
        queue.Enqueue(start);

        // Dictionary for tracking previous points
        Dictionary<Vector2Int, Vector2Int?> cameFrom = new
Dictionary<Vector2Int, Vector2Int?>
        {
            [start] = null
        };

        // BFS main loop
        while (queue.Count > 0)
        {
            Vector2Int current = queue.Dequeue();

            // If the goal is reached, restore the path
            if (current == goal)
            {
                List<Vector2Int> path = new List<Vector2Int>();
                while (current != null &&
cameFrom.ContainsKey(current))
                {
                    path.Add(current);
                    current =
cameFrom[current].GetValueOrDefault();
                }
                path.Reverse();
                return path;
            }

            // Checking neighbors
            foreach (Vector2Int direction in directions)
            {
                Vector2Int neighbor = current + direction;
```

```
                if (IsInBounds(neighbor) && maze[neighbor.x, neighbor.y] != 1 && !cameFrom.ContainsKey(neighbor))
                {
                    queue.Enqueue(neighbor);
                    cameFrom[neighbor] = current;
                }
            }
        }

        // If path not found
        Debug.LogWarning("Path not found!");
        return new List<Vector2Int>();
    }
}
```

Б.2.2 Алгоритм Дейкстри

```
using System.CodeDom.Compiler;
using System.Collections.Generic;
using UnityEngine;

// Dejkstra Search Algorithm
public class DejkstraSearch : IPathfindingAlgorithm
{
    // Generated maze
    public List<Vector2Int> FindPath(byte[,] maze)
    {
        // Size of the labyrinth
        int rows = maze.GetLength(0);
        int cols = maze.GetLength(1);

        // Announcement of start and end points
        Vector2Int start = Vector2Int.zero;
        Vector2Int goal = Vector2Int.zero;

        // Determining the start and finish points
        for (int x = 0; x < rows; x++)
        {
            for (int y = 0; y < cols; y++)
            {
                if (maze[x, y] == 2)
                {
                    start = new Vector2Int(x, y);
                }
                else if (maze[x, y] == 3)
                {
                    goal = new Vector2Int(x, y);
                }
            }
        }
```

```
        // Check for the presence of a start and end point
        if (maze[start.x, start.y] != 2 || maze[goal.x, goal.y]
!= 3)
        {
            Debug.LogError("Start or end point not found!");
            return new List<Vector2Int>();
        }

        // Checking the boundaries
        bool IsInBounds(Vector2Int pos) => pos.x >= 0 && pos.x <
rows && pos.y >= 0 && pos.y < cols;

        // Shifts for neighbors
        Vector2Int[] directions = { Vector2Int.up,
Vector2Int.down, Vector2Int.left, Vector2Int.right };

        // Minimum distances from the starting point
        Dictionary<Vector2Int, int> distances = new
Dictionary<Vector2Int, int>();
        distances[start] = 0;

        // Dictionary for tracking previous points
        Dictionary<Vector2Int, Vector2Int?> cameFrom = new
Dictionary<Vector2Int, Vector2Int?>
        {
            [start] = null
        };

        // Priority queue for processing nodes
        SortedSet<(int distance, Vector2Int point)>
priorityQueue = new SortedSet<(int, Vector2Int)>(new
DistanceComparer());
        priorityQueue.Add((0, start));

        while (priorityQueue.Count > 0)
        {
            // Extract the node with the smallest distance
            var currentPair = priorityQueue.Min;
            priorityQueue.Remove(currentPair);

            int currentDistance = currentPair.distance;
            Vector2Int current = currentPair.point;

            // Recovering the path when reaching the goal
            if (current == goal)
            {
                List<Vector2Int> path = new List<Vector2Int>();
                while (current != null &&
cameFrom.ContainsKey(current))
                {
                    path.Add(current);
                    current =
cameFrom[current].GetValueOrDefault();
```

```
                }
                path.Reverse();
                return path;
            }

            // Checking neighbors
            foreach (Vector2Int direction in directions)
            {
                Vector2Int neighbor = current + direction;

                if (IsInBounds(neighbor) && maze[neighbor.x,
neighbor.y] != 1)
                {
                    int newDistance = currentDistance + 1;

                    if (!distances.ContainsKey(neighbor) ||
newDistance < distances[neighbor])
                    {
                        // Update the distance to the neighbor
                        if (distances.ContainsKey(neighbor))
                        {
                            // Remove the old entry from the
queue

priorityQueue.Remove((distances[neighbor], neighbor));
                        }

                        distances[neighbor] = newDistance;
                        cameFrom[neighbor] = current;
                        priorityQueue.Add((newDistance,
neighbor));
                    }
                }
            }
        }

        // If path not found
        Debug.LogWarning("Path not found!");
        return new List<Vector2Int>();
    }

    // Comparator for SortedSet
    private class DistanceComparer : IComparer<(int distance,
Vector2Int point)>
    {
        public int Compare((int distance, Vector2Int point) a,
(int distance, Vector2Int point) b)
        {
            int distanceComparison =
a.distance.CompareTo(b.distance);
            if (distanceComparison == 0)
            {
                // If the distances are equal, compare by point
```

```
for uniqueness
                return a.point.x == b.point.x ?
a.point.y.CompareTo(b.point.y) : a.point.x.CompareTo(b.point.x);
            }
            return distanceComparison;
        }
    }
}
```

Б.2.3 Алгоритм A*

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


// A* Search Algorithm with Priority Queue Implementation
public class AStarSearch : IPathfindingAlgorithm
{
    // Implementation of a simple Priority Queue
    private class PriorityQueue<TElement, TPriority>
    {
        private List<(TElement Element, TPriority Priority)>
_elements = new List<(TElement, TPriority)>();
        private IComparer<TPriority> _comparer;

        public PriorityQueue(IComparer<TPriority> comparer =
null)
        {
            _comparer = comparer ?? Comparer<TPriority>.Default;
        }

        public int Count => _elements.Count;

        public void Enqueue(TElement element, TPriority
priority)
        {
            _elements.Add((element, priority));
        }

        public TElement Dequeue()
        {
            int bestIndex = 0;

            for (int i = 1; i < _elements.Count; i++)
            {
                if (_comparer.Compare(_elements[i].Priority,
_elements[bestIndex].Priority) < 0)
                {
                    bestIndex = i;
                }
            }
```

```
                TElement bestElement = _elements[bestIndex].Element;
                _elements.RemoveAt(bestIndex);
                return bestElement;
            }
        }


        // Main A* Search Algorithm
        public List<Vector2Int> FindPath(byte[,] maze)
        {
            // Size of the labyrinth
            int rows = maze.GetLength(0);
            int cols = maze.GetLength(1);

            // Announcement of start and end points
            Vector2Int start = Vector2Int.zero;
            Vector2Int goal = Vector2Int.zero;

            // Determining the start and finish points
            for (int x = 0; x < rows; x++)
            {
                for (int y = 0; y < cols; y++)
                {
                    if (maze[x, y] == 2) start = new Vector2Int(x,
y);
                    if (maze[x, y] == 3) goal = new Vector2Int(x,
y);
                }
            }


            // Checking the boundaries
            bool IsInBounds(Vector2Int pos) => pos.x >= 0 && pos.x <
rows && pos.y >= 0 && pos.y < cols;

            // Manhattan Heuristic
            int Heuristic(Vector2Int a, Vector2Int b) =>
Mathf.Abs(a.x - b.x) + Mathf.Abs(a.y - b.y);

            // Shifts for neighbors
            Vector2Int[] directions = { Vector2Int.up,
Vector2Int.down, Vector2Int.left, Vector2Int.right };

            // Initialize the priority queue (open set) with the
start node
            PriorityQueue<Vector2Int, int> openQueue = new
PriorityQueue<Vector2Int, int>();
            openQueue.Enqueue(start, Heuristic(start, goal));

            // G-cost dictionary
            Dictionary<Vector2Int, int> gScore = new
Dictionary<Vector2Int, int>
```

```
        {
            [start] = 0
        };

        // Dictionary to restore the path
        Dictionary<Vector2Int, Vector2Int?> cameFrom = new
Dictionary<Vector2Int, Vector2Int?>
        {
            [start] = null
        };

        // Closed set to keep track of processed nodes
        HashSet<Vector2Int> closedSet = new
HashSet<Vector2Int>();

        // Main loop
        while (openQueue.Count > 0)
        {
            // Dequeue the node with the lowest fScore
            Vector2Int current = openQueue.Dequeue();

            // If you have reached your goal
            if (current == goal)
            {
                List<Vector2Int> path = new List<Vector2Int>();
                while (current != null &&
cameFrom.TryGetValue(current, out Vector2Int? previous))
                {
                    path.Add(current);
                    current = previous.GetValueOrDefault();
                }
                path.Reverse();
                return path;
            }


            // Add current node to closed set
            closedSet.Add(current);

            // Processing neighbors
            foreach (Vector2Int direction in directions)
            {
                Vector2Int neighbor = current + direction;

                if (!IsInBounds(neighbor) || maze[neighbor.x,
neighbor.y] == 1 || closedSet.Contains(neighbor))
                {
                    continue; // Skip out-of-bounds, walls, or
already processed nodes
                }

                // Tentative G-score
                int tentativeGScore = gScore[current] + 1; //
```

```
Assuming cost between adjacent nodes is 1

                // If neighbor is not in gScore or
tentativeGScore is better
                if (!gScore.ContainsKey(neighbor) ||
tentativeGScore < gScore[neighbor])
                {
                    gScore[neighbor] = tentativeGScore;
                    int fScore = tentativeGScore +
Heuristic(neighbor, goal);

                    // Enqueue the neighbor with the updated
fScore
                    openQueue.Enqueue(neighbor, fScore);
                    cameFrom[neighbor] = current;
                }
            }
        }

        // If path not found
        return new List<Vector2Int>();
    }
}
```

Б.3 Реалізація тестових функцій

Б.3.1 Генерація лабіринту

```
using System.Collections.Generic;
using UnityEngine;
using System.Diagnostics; // Для измерения времени

public class MazeGenerator : MonoBehaviour
{
    private int width = 10;
    private int height = 10;

    private Vector2Int startPoint;
    private Vector2Int endPoint;

    public enum GenerationMethod { BinaryTree, AldousBroder,
Eller }
    public GenerationMethod selectedMethod =
GenerationMethod.AldousBroder;

    private byte[,] maze;
    private static byte[,] finalMaze;

    private Dictionary<GenerationMethod, IMazeGenerator>
```

```
generators;

    private void Start()
    {
        generators = new Dictionary<GenerationMethod,
IMazeGenerator>
        {
            { GenerationMethod.BinaryTree, new
BinaryTreeMazeGenerator() },
            { GenerationMethod.AldousBroder, new
AldousBroderMazeGenerator() },
            { GenerationMethod.Eller, new EllerMazeGenerator() }
        };

        GenerateMaze();
    }

    private void GenerateMaze()
    {
        if (!generators.ContainsKey(selectedMethod))
        {
            UnityEngine.Debug.LogError("The selected generation
method is not implemented!");
            return;
        }

        // Запуск таймера
        Stopwatch stopwatch = Stopwatch.StartNew();

        maze = generators[selectedMethod].GenerateMaze(ref
width, ref height);
        AddWalls();
        AddStartEndPoints();

        stopwatch.Stop();
        UnityEngine.Debug.Log($"Maze generation time:
{stopwatch.ElapsedMilliseconds} ms");

        PrintMaze();
    }

    private void AddWalls()
    {
        finalMaze = new byte[height + 2, width + 2];
        for (int x = 0; x < height; x++)
        {
            for (int y = 0; y < width; y++)
            {
                finalMaze[x + 1, y + 1] = maze[x, y];
            }
        }
        for (int x = 0; x < height + 2; x++)
            finalMaze[x, 0] = finalMaze[x, width + 1] = 1;
```

```
        for (int y = 0; y < width + 2; y++)
            finalMaze[0, y] = finalMaze[height + 1, y] = 1;
    }

    private void PrintMaze()
    {
        string mazeString = "";
        for (int x = 0; x < height + 2; x++)
        {
            for (int y = 0; y < width + 2; y++)
            {
                mazeString += finalMaze[x, y];
            }
            mazeString += "\n";
        }
        UnityEngine.Debug.Log(mazeString);
    }

    private void AddStartEndPoints()
    {
        startPoint = new Vector2Int(finalMaze.GetLength(0) - 2,
1);
        if (finalMaze[startPoint.x, startPoint.y] == 0)
        {
            finalMaze[startPoint.x, startPoint.y] = 2;
        }
        else
        {
            UnityEngine.Debug.LogError("The starting point is on
the wall!");
        }

        endPoint = new Vector2Int(1, finalMaze.GetLength(1) -
2);
        if (finalMaze[endPoint.x, endPoint.y] == 0)
        {
            finalMaze[endPoint.x, endPoint.y] = 3;
        }
        else
        {
            UnityEngine.Debug.LogError("The end point is on the
wall!");
        }
    }

    public static byte[,] GetMaze()
    {
        return finalMaze;
    }
}
```

Б.3.2 Пошук шляху

```
using System.Collections.Generic;
using UnityEngine;
using System.Diagnostics; // Для измерения времени

public class PathfindingController : MonoBehaviour
{
    private Dictionary<string, IPathfindingAlgorithm>
algorithms; // List of algorithms
    private Dictionary<string, List<Vector2Int>> paths; // Paths
found for each algorithm

    private void Start()
    {
        // Initialization of algorithms
        algorithms = new Dictionary<string,
IPathfindingAlgorithm>
        {
            { "BreadthFirstSearch", new BreadthFirstSearch() },
            { "DejkstraSearch", new DejkstraSearch() },
            { "AStar", new AStarSearch() }
        };

        // Initialize path storage
        paths = new Dictionary<string, List<Vector2Int>>();

        // Get the labyrinth
        byte[,] maze = MazeGenerator.GetMaze();

        // Check for null (if the maze was not generated)
        if (maze == null)
        {
            UnityEngine.Debug.LogError("The maze was not
generated!");
            return;
        }

        // Start search
        RunPathfinding(maze);
    }

    private void RunPathfinding(byte[,] maze)
    {
        foreach (var algorithm in algorithms)
        {
            Stopwatch stopwatch = Stopwatch.StartNew();

            // Run each algorithm
            List<Vector2Int> path =
algorithm.Value.FindPath(maze);
```

```
            stopwatch.Stop();
            UnityEngine.Debug.Log($"Algorithm {algorithm.Key}
execution time: {stopwatch.ElapsedMilliseconds} ms");

            // Save the result
            paths[algorithm.Key] = path;

            // Output the result
            if (path.Count > 0)
            {
                UnityEngine.Debug.Log($"Path found using
{algorithm.Key}");
                //UnityEngine.Debug.Log($"Path found using
{algorithm.Key}: {string.Join(" -> ", path)}");
            }
            else
            {
                UnityEngine.Debug.Log($"Path not found using
{algorithm.Key}.");
            }
        }
    }
}
```