ДОДАТОК А

Текст програми

```
using System;
public class FloatKey : IComparable<FloatKey> {
    public float PrimaryKey { get; set; }
    public float SecondaryKey { get; set; }
    public FloatKey(float primaryKey, float secondaryKey) {
        PrimaryKey = primaryKey;
        SecondaryKey = secondaryKey;
    }
    public override bool Equals(object obj) {
        if (!(obj is FloatKey)) {
            return false;
        }
        FloatKey key = (FloatKey) obj;
        return PrimaryKey.Equals(key.PrimaryKey) &&
               SecondaryKey.Equals(key.SecondaryKey);
    }
    public override int GetHashCode() {
        var hashCode = 1502939027;
        hashCode = hashCode * -1521134295 + (int) PrimaryKey;
        hashCode = hashCode * -1521134295 + (int) SecondaryKey;
        return hashCode;
    }
    public override string ToString() {
        return "(" + PrimaryKey + ", " + SecondaryKey + ")";
    }
    public int CompareTo(FloatKey other) {
        if (PrimaryKey < other.PrimaryKey) {
            return -1;
        }
        if (PrimaryKey > other.PrimaryKey) {
            return 1;
```

```
        }
        if (SecondaryKey < other.SecondaryKey) {
            return -1;
        }
        if (SecondaryKey > other.SecondaryKey) {
            return 1;
        }
        return 0;
    }
    public static bool operator <(FloatKey left, FloatKey right) {
        return left.CompareTo(right) < 0;
    }
    public static bool operator >(FloatKey left, FloatKey right) {
        return left.CompareTo(right) > 0;
    }
    public static bool operator <=(FloatKey left, FloatKey right) {
        return !(left > right);
    }
    public static bool operator >=(FloatKey left, FloatKey right) {
        return !(left < right);
    }
}

using System;
public class Vertex {
public Hexagon Hexagon { get; private set; }
    public float g { get; set; }
    public float rhs { get; set; }
    public float OldMovementCost { get; set; }
    public Vertex(Hexagon hexagon) {
        Hexagon = hexagon;
        Reset();
    }
    public void Reset() {
```

```
        g = float.PositiveInfinity;
        rhs = float.PositiveInfinity;
        OldMovementCost = float.PositiveInfinity;
    }
    public float GetMovementCost(Vertex startVertex) {
        if (!Equals(startVertex) && Hexagon.unit != null) {
            return float.PositiveInfinity;
        }
        return Hexagon.GetMovementCost();
    }
    public float CalculateHeuristicCost(Vertex goal) {
        return Math.Abs(Hexagon.coord.x - goal.Hexagon.coord.x) +
Math.Abs(Hexagon.coord.y - goal.Hexagon.coord.y);
    }
    public override int GetHashCode() {
        var hashCode = 1502939027;
        hashCode = hashCode * -1521134295 + Hexagon.GetHashCode();
        return hashCode;
    }
    public override bool Equals(object obj) {
        if (obj is Vertex) {
            Vertex objVertex = (Vertex) obj;
            return Hexagon.Equals(objVertex.Hexagon);
        }
        return false;
    }
    public override string ToString() {
        return Hexagon.coord.ToString();
    }

}

using Priority_Queue;
public class VertexPriorityQueue : SimplePriorityQueue<Vertex, FloatKey> {
    public FloatKey TopKey() {
```

```
        return GetPriority(First);
    }
}

public class Coordinate<T> {
    public T x;
    public T y;
    public Coordinate(T x, T y) {
        this.x = x;
        this.y = y;
    }
    public override bool Equals(object obj) {
        var coordinate = obj as Coordinate<T>;
        return coordinate != null &&
               x.Equals(coordinate.x) &&
               y.Equals(coordinate.y);
    }
    public override int GetHashCode() {
        var hashCode = 1502939027;
        hashCode = hashCode * -1521134295 + x.GetHashCode();
        hashCode = hashCode * -1521134295 + y.GetHashCode();
        return hashCode;
    }
    public override string ToString() {
        return "(" + x + ", " + y + ")";
    }
}

public class PathFindingResult {
    public Hexagon NextStep { get; set; }
    public float MovePointsLeft { get; set; }
}
```

```
using UnityEngine;

public class CameraMovement : MonoBehaviour {

    private Vector3 dragOrigin;

    private Vector3 dragDifference;

    private bool drag;

    private void LateUpdate() {

        HandleMovement();

        HandleZooming();

    }

    private void HandleMovement() {

        HandleKeyboardMovement();

        HandleMouseMovement();

    }

    private void HandleKeyboardMovement() {

        if (Input.GetKey("w")) {

            transform.position = new Vector3(transform.position.x,
transform.position.y + Constants.CAMERA_MOVEMENT_SPEED, transform.position.z);

        }

        if (Input.GetKey("s")) {

            transform.position = new Vector3(transform.position.x,
transform.position.y - Constants.CAMERA_MOVEMENT_SPEED, transform.position.z);

        }

        if (Input.GetKey("a")) {

            transform.position = new Vector3(transform.position.x -
Constants.CAMERA_MOVEMENT_SPEED, transform.position.y, transform.position.z);

        }

        if (Input.GetKey("d")) {

            transform.position = new Vector3(transform.position.x +
Constants.CAMERA_MOVEMENT_SPEED, transform.position.y, transform.position.z);

        }

    }

    private void HandleMouseMovement() {

        if (Input.GetMouseButton(Constants.MIDDLE_MOUSE_BUTTON)) {

            dragDifference = Camera.main.ScreenToWorldPoint(Input.mousePosition) -
Camera.main.transform.position;
```

```
            if (!drag) {
                drag = true;
                dragOrigin = Camera.main.ScreenToWorldPoint(Input.mousePosition);
            }
        } else {
            drag = false;
        }
        if (drag) {
            Camera.main.transform.position = dragOrigin - dragDifference;
        }
    }
    private void HandleZooming() {
        var cameraSize = gameObject.GetComponent<Camera>().orthographicSize;

        if (cameraSize > Constants.MAX_SCROLL && Input.GetAxis("Mouse ScrollWheel") >
0f || Input.GetKey("up")) {
            cameraSize -= Constants.CAMERA_SCROLL_SPEED;
        }
        if (cameraSize < Constants.MIN_SCROLL && Input.GetAxis("Mouse ScrollWheel") <
0f || Input.GetKey("down")) {
            cameraSize += Constants.CAMERA_SCROLL_SPEED;
        }
        gameObject.GetComponent<Camera>().orthographicSize = cameraSize;
    }
}

using UnityEngine;
using UnityEngine.UI;
public class EndTurnHandler : MonoBehaviour {
    public Map map;
    private void Start() {
        var btn = gameObject.GetComponent<Button>();
        btn.onClick.AddListener(TaskOnClick);
    }
```

```
    private void TaskOnClick() {
        foreach (var player in map.GetPlayers()) {
            RefreshUnitsMovementPoints(player);
        }
    }
    private void RefreshUnitsMovementPoints(Player player) {
        foreach (var unit in player.units) {
            unit.HandleTurnEnd();
        }
    }
}

public class MouseInputHandler {
    private Map map;
    private Hexagon previusHexagon;
    private Unit previusUnit;
    public MouseInputHandler(ref Map map) {
        this.map = map;
    }
    public void HandleLMBClick(Coordinate<int> coord) {
        var hexagon = map.GetHexagon(coord);
        HandleHexagonClick(hexagon);
        HandleUnitClick(hexagon);
        HandleTerrainClick(hexagon);
    }
    private void HandleHexagonClick(Hexagon hexagon) {
        if (previusHexagon != null) {
            previusHexagon.SetSprite(Hexagon.SPRITE_BASE);
        }
        hexagon.SetSprite(Hexagon.SPRITE_SELECTED);
        previusHexagon = hexagon;
    }
    private void HandleUnitClick(Hexagon hexagon) {
        var unit = hexagon.unit;
```

```
        if (unit != null) {
            previusUnit = unit;
        }
    }
    private void HandleTerrainClick(Hexagon hexagon) {
        var unit = hexagon.unit;
        if (unit == null && previusUnit != null) {
            previusUnit.MoveTo(ref hexagon);
            previusUnit = null;
        }
    }
    public void HandleRMBClick() {
        if (previusHexagon != null) {
            previusHexagon.SetSprite(Hexagon.SPRITE_BASE);
        }
        previusUnit = null;
    }
}

using UnityEngine;
public class MouseInputListener : MonoBehaviour {
    public Coordinate<int> coord;
    public MouseInputHandler mouseInputHandler;
    private void OnMouseDown() {
        mouseInputHandler.HandleLMBClick(coord);
    }
    private void Update() {
        if (Input.GetMouseButtonDown(Constants.RIGHT_MOUSE_BUTTON)) {
            mouseInputHandler.HandleRMBClick();
        }
    }
}

using UnityEngine;
```

```
public abstract class CustomGameObject : ScriptableObject {

    public readonly GameObject gameObject;

    public float layer;

    public CustomGameObject() {

    }

    public CustomGameObject(GameObject gameObject, float layer) {

        this.gameObject = gameObject;

        this.layer = layer;

        name = gameObject.name.Replace("(Clone)", "");

        gameObject.name = name;

    }

}
```

```
using UnityEngine;

public abstract class OnHexagonObject : CustomGameObject {

    public Hexagon hexagon;

    public OnHexagonObject(GameObject gameObject, float layer, ref Hexagon hexagon) :
base(gameObject, layer) {

        this.hexagon = hexagon;

        MoveTo(ref hexagon);

    }

    public void MoveTo(ref Hexagon goal) {

        ObjectMover.MoveObject(gameObject, goal, layer);

        ChangeParent(ref goal);

    }

    protected abstract void ChangeParent(ref Hexagon newParent);

}
```

```
using UnityEngine;
public abstract class Terrain : OnHexagonObject {

    public float movementCost;

    public Terrain(GameObject gameObject, float layer, ref Hexagon hexagon) :
base(gameObject, layer, ref hexagon) {

    }

}
```

```
public class FeatureFactory {
    public Feature NewForestFeature(ref Hexagon hexagon) {
        var prefab = ResourceLoader.LoadPrefab(Constants.PREFAB_PATH_FOREST);
        var feature = new Feature(prefab, Constants.FEATURE_LAYER, ref hexagon) {
            movementCost = Constants.MOVE_COST_FOREST
        };
        return feature;
    }
}

public class HexagonFactory {
    private readonly MouseInputHandler mouseInputHandler;
    public HexagonFactory(MouseInputHandler mouseInputHandler) {
        this.mouseInputHandler = mouseInputHandler;
    }
    public Hexagon NewHexagon(Coordinate<int> coord, bool displayCoordinates) {
        var prefab = ResourceLoader.LoadPrefab(Constants.PREFAB_PATH_HEXAGON);
        var inputListener = prefab.GetComponent<MouseInputListener>();
        inputListener.mouseInputHandler = mouseInputHandler;
        inputListener.coord = coord;
        var hexagon = new Hexagon(prefab, coord, displayCoordinates);
        return hexagon;
    }
}

public class HighlighterFactory {
    public Highlighter NewCalculatedHighlighter(Hexagon hexagon) {
        var prefab =
ResourceLoader.LoadPrefab(Constants.PREFAB_PATH_HIGHLIGHTER_CALCULATED);
        return new Highlighter(prefab, Constants.HIGHLIGHTER_LAYER, ref hexagon);
    }
    public Highlighter NewRecalculatedHighlighter(Hexagon hexagon) {
        var prefab =
ResourceLoader.LoadPrefab(Constants.PREFAB_PATH_HIGHLIGHTER_RECALCULATED);
```

```
        return new Highlighter(prefab, Constants.HIGHLIGHTER_LAYER, ref hexagon);
    }
}

public class LandscapeFactory {
    public Landscape NewPlain(ref Hexagon hexagon) {
        var prefab = ResourceLoader.LoadPrefab(Constants.PREFAB_PATH_PLAIN);
        var landscape = new Landscape(prefab, Constants.LANDSCAPE_LAYER, ref hexagon)
{
            movementCost = Constants.MOVE_COST_PLAIN
        };
        return landscape;
    }
    public Landscape NewHill(ref Hexagon hexagon) {
        var prefab = ResourceLoader.LoadPrefab(Constants.PREFAB_PATH_HILL);
        var landscape = new Landscape(prefab, Constants.LANDSCAPE_LAYER, ref hexagon)
{
            movementCost = Constants.MOVE_COST_HILL
        };
        return landscape;
    }
    public Landscape NewMountain(ref Hexagon hexagon) {
        var prefab = ResourceLoader.LoadPrefab(Constants.PREFAB_PATH_MOUNTAIN);
        var landscape = new Landscape(prefab, Constants.LANDSCAPE_LAYER, ref hexagon)
{
            movementCost = Constants.MOVE_COST_MOUNTAIN
        };
        return landscape;
    }
}

public class UnitFactory {
    public Unit NewInfantry(ref Hexagon hexagon, ref Player player, IPathFinder
pathFinder) {
        var prefab = ResourceLoader.LoadPrefab(Constants.PREFAB_PATH_INFANTRY);
```

```
        var unit = new Unit(prefab, Constants.UNIT_LAYER, ref hexagon, ref player,
pathFinder) {
            hexagon = hexagon,
            movePointsTotal = Constants.MOVE_POINTS_INFANTRY,
            movePointsLeft = Constants.MOVE_POINTS_INFANTRY
        };
        return unit;
    }
}


using UnityEngine;
public class Feature : Terrain {
    public Feature(GameObject gameObject, float layer, ref Hexagon hexagon) :
base(gameObject, layer, ref hexagon) {
    }
    protected override void ChangeParent(ref Hexagon newParent) {
        hexagon.feature = null;
        newParent.feature = this;
        hexagon = newParent;
    }
}

using System;
using System.Collections.Generic;
using System.Diagnostics.CodeAnalysis;
using System.Linq;
using UnityEngine;
public class Hexagon : CustomGameObject {
    private const float TEXTURE_SIZE_X = 5.8f;
    private const float TEXTURE_SIZE_Y = 5;
    public static readonly Sprite SPRITE_BASE =
ResourceLoader.LoadSprite(Constants.SPRITE_PATH_HEXAGON_BASE);
    public static readonly Sprite SPRITE_SELECTED =
ResourceLoader.LoadSprite(Constants.SPRITE_PATH_HEXAGON_SELECTED);
    private readonly Dictionary<Unit, Vertex> vertices = new Dictionary<Unit,
Vertex>();
```

```
    private TextMesh textMesh;

    public Landscape landscape;

    public Feature feature;

    private Highlighter highlighter;

    public Unit unit;

    public Coordinate<int> coord;

    public Coordinate<float> realCoord;

    public Hexagon() {

    }

    public Hexagon(GameObject gameObject, Coordinate<int> coord, bool
displayCoordinates) : base(gameObject, Constants.HEXAGON_LAYER) {

        SetCoordinates(coord);

        if (displayCoordinates) {

            SetText(ResourceLoader.LoadTextMesh(Constants.PREFAB_PATH_TEXT_BASIC,
coord.ToString()));

        }

    }

    private void SetCoordinates(Coordinate<int> coord) {

        this.coord = coord;

        var realX = coord.x * TEXTURE_SIZE_X * 0.75f;

        var realY = coord.y * TEXTURE_SIZE_Y;

        if (coord.x % 2 == 1) {

            realY += TEXTURE_SIZE_Y * 0.5f;

        }

        realCoord = new Coordinate<float>(realX, realY);

        gameObject.transform.position = new Vector3(realCoord.x, realCoord.y, layer);

    }

    public Vertex GetVertex(Unit unit) {

        if (!vertices.ContainsKey(unit)) {

            vertices.Add(unit, new Vertex(this));

        }

        return vertices[unit];

    }

    public float GetMovementCost() {
```

```
        var movementCost = landscape.movementCost;
        if (feature != null) {
            movementCost += feature.movementCost;
        }
        return movementCost;
    }
    public void SetSprite(Sprite sprite) {
        gameObject.GetComponent<SpriteRenderer>().sprite = sprite;
    }
    public void SetText(TextMesh textMesh) {
        if (this.textMesh != null) {
            this.textMesh.gameObject.SetActive(false);
        }
        ObjectMover.MoveObject(textMesh.gameObject, this, Constants.TEXT_LAYER);
    }
    public void SetHighlighter(Highlighter highlighter) {
        if (this.highlighter != null) {
            this.highlighter.gameObject.SetActive(false);
        }
        this.highlighter = highlighter;
    }
    public bool IsNeighbour(Hexagon hexagon) {
        return GetPlainDistance(hexagon) == 1;
    }
    public int GetPlainDistance(Hexagon hexagon) {
        double realDistance = Math.Sqrt(Math.Pow(hexagon.realCoord.x - realCoord.x,
2) + Math.Pow(hexagon.realCoord.y - realCoord.y, 2));
        return Convert.ToInt32(realDistance / 5);
    }
    public override int GetHashCode() {
        var hashCode = 1502939027;
        hashCode = hashCode * -1521134295 + gameObject.GetHashCode();
        return hashCode;
```

```
    }
    public override bool Equals(object obj) {
        if (!(obj is Hexagon)) {
            return false;
        }
        Hexagon hexagon = (Hexagon) obj;
        return gameObject.Equals(hexagon.gameObject);
    }
}

using UnityEngine;
public class Highlighter : OnHexagonObject {
    public Highlighter(GameObject gameObject, float layer, ref Hexagon hexagon) :
base(gameObject, layer, ref hexagon) {
    }
    protected override void ChangeParent(ref Hexagon newParent) {
        hexagon.SetHighlighter(this);
    }
}

using UnityEngine;
public class Landscape : Terrain {
    public Landscape(GameObject gameObject, float layer, ref Hexagon hexagon) :
base(gameObject, layer, ref hexagon) {
    }
    protected override void ChangeParent(ref Hexagon newParent) {
        hexagon.landscape = null;
        newParent.landscape = this;
        hexagon = newParent;
    }
}

using System.Collections.Generic;
public class Player {
```

```
    public string name;
    public readonly HashSet<Unit> units = new HashSet<Unit>();
    public Player(string name) {
        this.name = name;
    }
}

using System.Collections.Generic;
using System.Linq;
using UnityEngine;
public class Map {
    private Hexagon[,] hexagons;
    private List<Player> players;
    private readonly Dictionary<Hexagon, HashSet<Hexagon>> neighbourDictionary = new
Dictionary<Hexagon, HashSet<Hexagon>>();
    public Hexagon[,] GetHexagons() {
        return hexagons;
    }
    public void SetHexagons(Hexagon[,] hexagons) {
        this.hexagons = hexagons;
        InitNeighbourDictionary();
    }
    private void InitNeighbourDictionary() {
        foreach (Hexagon hexagon in hexagons) {
            AddNeighboursToDictionary(hexagon);
        }
    }
    private void AddNeighboursToDictionary(Hexagon hexagon) {
        foreach (var potentialNeighbour in hexagons) {
            if (hexagon.IsNeighbour(potentialNeighbour)) {
                if (!neighbourDictionary.ContainsKey(hexagon)) {
                    neighbourDictionary.Add(hexagon, new HashSet<Hexagon>());
                }
                neighbourDictionary[hexagon].Add(potentialNeighbour);
```

```
            }
        }
    }
    public void SetPlayers(List<Player> players) {
        this.players = players;
    }
    public IEnumerable<Player> GetPlayers() {
        return players;
    }
    public Hexagon GetHexagon(Coordinate<int> coord) {
        return hexagons[coord.x, coord.y];
    }
    public IEnumerable<Hexagon> GetNeighbours(Hexagon hexagon) {
        return neighbourDictionary[hexagon];
    }
    public IEnumerable<Vertex> GetSuccessors(Vertex vertex, Unit unit) {
        return GetNeighbours(vertex.Hexagon)
            .Select(hexagon => hexagon.GetVertex(unit));
    }
    public IEnumerable<Vertex> GetSuccessorsWithSelf(Vertex vertex, Unit unit) {
        return new List<Vertex>(GetSuccessors(vertex, unit)) {vertex};
    }
    public IEnumerable<Vertex> GetPredecessors(Vertex vertex, Unit unit) {
        return GetNeighbours(vertex.Hexagon)
            .Select(hexagon => hexagon.GetVertex(unit));
    }
    public IEnumerable<Vertex> GetPredecessorsWithSelf(Vertex vertex, Unit unit) {
        return new List<Vertex>(GetPredecessors(vertex, unit)) {vertex};
    }
}

using UnityEngine;
public class Unit : OnHexagonObject {
```

```
    private readonly IPathFinder pathFinder;
    public Hexagon currentGoal;
    public Player player;
    public float movePointsTotal;
    public float movePointsLeft;
    public bool isHighlightable;
    public Unit(GameObject gameObject, float layer, ref Hexagon hexagon, ref Player
player, IPathFinder pathFinder) : base(gameObject, layer,
        ref hexagon) {
        this.player = player;
        player.units.Add(this);
        this.pathFinder = pathFinder;
    }
    public new void MoveTo(ref Hexagon goal) {
        PathFindingResult result = pathFinder.FindNextStepInPath(hexagon, goal,
this);
        if (result != null) {
            currentGoal = goal;
            movePointsLeft = result.MovePointsLeft;
            Hexagon nextStep = result.NextStep;
            base.MoveTo(ref nextStep);
            if (result.NextStep.Equals(currentGoal)) {
                currentGoal = null;
            }
        }
    }
    protected override void ChangeParent(ref Hexagon newParent) {
        hexagon.unit = null;
        newParent.unit = this;
        hexagon = newParent;
    }
    public void HandleTurnEnd() {
        movePointsLeft = movePointsTotal;
        if (currentGoal != null && movePointsLeft > 0) {
```

```
            MoveTo(ref currentGoal);
        }
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
public class DStarLitePathFinder : IPathFinder {
    private const float FLOAT_TOLERANCE = 0.001f;
    private readonly HighlighterFactory highlighterFactory = new
HighlighterFactory();
    private readonly Map map;
    private VertexPriorityQueue priorityQueue;
    private float additionalHeuristic;
    private Vertex startVertex;
    private Vertex goalVertex;
    private Vertex lastVertex;
    public DStarLitePathFinder(Map map) {
        this.map = map;
    }
    public PathFindingResult FindNextStepInPath(Hexagon start, Hexagon goal, Unit
unit) {
        float movePointsLeft = unit.movePointsLeft;
        if (goalVertex == null || !goalVertex.Equals(goal.GetVertex(unit))) {
            StartMovement(start, goal, unit);
        } else {
            UpdatePathIfFoundChanges(unit);
        }
        if (float.IsPositiveInfinity(startVertex.rhs)) {
            return null;
        }
        foreach (var vertex in priorityQueue) {
            foreach (var successor in map.GetSuccessorsWithSelf(vertex, unit)) {
                successor.OldMovementCost = successor.GetMovementCost(startVertex);
```

```
            }
        }
        while (movePointsLeft > 0) {
            Vertex tentativeNext = ComputeNextMove(startVertex, unit);
            float tentativeMovementCost = tentativeNext.GetMovementCost(startVertex);
            if (movePointsLeft - tentativeMovementCost > 0) {
                startVertex = tentativeNext;
                movePointsLeft -= tentativeMovementCost;
                if (startVertex.Equals(goalVertex)) {
                    break;
                }
            } else {
                break;
            }
        }
        return new PathFindingResult {
            NextStep = startVertex.Hexagon,
            MovePointsLeft = movePointsLeft
        };
    }
    private void StartMovement(Hexagon start, Hexagon goal, Unit unit) {
        startVertex = start.GetVertex(unit);
        goalVertex = goal.GetVertex(unit);
        lastVertex = startVertex;
        Initialize(unit);
        ComputeShortestPath(unit, hexagon =>
highlighterFactory.NewCalculatedHighlighter(hexagon));
    }
    private void Initialize(Unit unit) {
        ClearVerticesInfo(unit);
        priorityQueue = new VertexPriorityQueue();
        additionalHeuristic = 0;
        goalVertex.rhs = 0;
```

```
        priorityQueue.Enqueue(goalVertex, new
FloatKey(startVertex.CalculateHeuristicCost(goalVertex), 0));
    }
    private void ClearVerticesInfo(Unit unit) {
        foreach (Hexagon hexagon in map.GetHexagons()) {
            Vertex vertex = hexagon.GetVertex(unit);
            if (vertex != null) {
                vertex.Reset();
            }
        }
    }
    private void ComputeShortestPath(Unit unit, Action<Hexagon> highlighter) {
        while (priorityQueue.TopKey() < CalculateKey(startVertex) || startVertex.rhs
> startVertex.g) {
            Vertex topVertex = priorityQueue.First;
            FloatKey kOld = priorityQueue.TopKey();
            FloatKey kNew = CalculateKey(topVertex);
            if (kOld < kNew) {
                priorityQueue.UpdatePriority(topVertex, kNew);
            } else if (topVertex.g > topVertex.rhs) {
                topVertex.g = topVertex.rhs;
                priorityQueue.Remove(topVertex);
                foreach (Vertex predecessor in map.GetPredecessors(topVertex, unit))
{
                    if (!predecessor.Equals(goalVertex)) {
                        predecessor.rhs = CalculateRhs(predecessor, topVertex);
                    }
                    UpdateVertex(predecessor);
                    if (unit.isHighlightable) {
                        highlighter.Invoke(predecessor.Hexagon);
                    }
                }
            } else {
                float gOld = topVertex.g;
```

```
                topVertex.g = float.PositiveInfinity;
                foreach (Vertex predecessor in map.GetPredecessorsWithSelf(topVertex,
unit)) {
                    if (Math.Abs(predecessor.rhs - (CalculateEdgeCost(topVertex) +
gOld)) < FLOAT_TOLERANCE && !predecessor.Equals(goalVertex)) {
                        predecessor.rhs = CalculateUpdatedRhs(predecessor, unit);
                    }
                    UpdateVertex(predecessor);
                    if (unit.isHighlightable) {
                        highlighter.Invoke(predecessor.Hexagon);
                    }
                }
            }
        }
    }
    private void UpdatePathIfFoundChanges(Unit unit) {
        IEnumerable<Vertex> changed = FindChangedVertices(unit);
        if (changed.Count() != 0) {
            additionalHeuristic += lastVertex.CalculateHeuristicCost(startVertex);
            lastVertex = startVertex;
            UpdateShortestPath(changed, unit);
        }
    }
    private IEnumerable<Vertex> FindChangedVertices(Unit unit) {
        List<Vertex> changed = new List<Vertex>();
        foreach (var vertex in priorityQueue) {
            foreach (var successor in map.GetSuccessorsWithSelf(vertex, unit)) {
                float oldCost = successor.OldMovementCost;
                float newCost = successor.GetMovementCost(startVertex);
                if (Math.Abs(oldCost - newCost) > FLOAT_TOLERANCE) {
                    changed.Add(successor);
                }
            }
        }
```

```
        return changed;
    }
    private void UpdateShortestPath(IEnumerable<Vertex> changed, Unit unit) {
        foreach (Vertex changedVertex in changed) {
            if (!changedVertex.Equals(goalVertex)) {
                foreach (Vertex successor in map.GetSuccessors(changedVertex, unit))
{
                    float cOld = successor.OldMovementCost;
                    if (cOld > CalculateEdgeCost(successor)) {
                        changedVertex.rhs = CalculateRhs(changedVertex, successor);
                    } else if (Math.Abs(changedVertex.rhs - (cOld + successor.g)) <
FLOAT_TOLERANCE) {
                        changedVertex.rhs = CalculateUpdatedRhs(changedVertex, unit);
                    }
                }
            }
            if (unit.isHighlightable) {
                highlighterFactory.NewRecalculatedHighlighter(changedVertex.Hexagon);
            }
            UpdateVertex(changedVertex);
        }
        ComputeShortestPath(unit, hexagon =>
highlighterFactory.NewRecalculatedHighlighter(hexagon));
    }
    private FloatKey CalculateKey(Vertex vertex) {
        float primaryKey = Math.Min(vertex.g, vertex.rhs) +
startVertex.CalculateHeuristicCost(vertex) + additionalHeuristic;
        float secondaryKey = Math.Min(vertex.g, vertex.rhs);
        return new FloatKey(primaryKey, secondaryKey);
    }
    private void UpdateVertex(Vertex vertex) {
        if (Math.Abs(vertex.g - vertex.rhs) > FLOAT_TOLERANCE &&
priorityQueue.Contains(vertex)) {
            priorityQueue.UpdatePriority(vertex, CalculateKey(vertex));
```

```
        } else if (Math.Abs(vertex.g - vertex.rhs) > FLOAT_TOLERANCE &&
!priorityQueue.Contains(vertex)) {
            priorityQueue.Enqueue(vertex, CalculateKey(vertex));
        } else if (Math.Abs(vertex.g - vertex.rhs) < FLOAT_TOLERANCE &&
priorityQueue.Contains(vertex)) {
            priorityQueue.Remove(vertex);
        }
    }
    private float CalculateRhs(Vertex start, Vertex goal) {
        return Math.Min(start.rhs, CalculateEdgeCost(goal) + goal.g);
    }
    private float CalculateUpdatedRhs(Vertex vertex, Unit unit) {
        return map.GetSuccessors(vertex, unit)
            .Min(successor => CalculateSuccessorCost(successor));
    }
    private Vertex ComputeNextMove(Vertex vertex, Unit unit) {
        return map.GetSuccessors(vertex, unit)
            .Aggregate((successor, nextSuccessor) =>
                CalculateSuccessorCost(successor) <
CalculateSuccessorCost(nextSuccessor)
                    ? successor
                    : nextSuccessor);
    }
    private float CalculateSuccessorCost(Vertex successor) {
        return CalculateEdgeCost(successor) + successor.g;
    }
    private float CalculateEdgeCost(Vertex goal) {
        return goal.GetMovementCost(startVertex);
    }
}

public interface IPathFinder {
    PathFindingResult FindNextStepInPath(Hexagon start, Hexagon goal, Unit unit);
}
```

```
using System;
class DiceRoller {
    private static readonly Random RANDOMIZER = new Random();
    public static bool Roll(int chance) {
        var dice = RANDOMIZER.Next(1, 100);
        return dice <= chance;
    }
}

using UnityEngine;
class ObjectMover {
    public static void MoveObject(GameObject gameObject, Hexagon goal, float layer) {
        var x = goal.gameObject.transform.position.x;
        var y = goal.gameObject.transform.position.y;
        gameObject.transform.position = new Vector3(x, y, layer);
    }
}

using UnityEngine;
public class ResourceLoader : MonoBehaviour {
    public static GameObject LoadPrefab(string path) {
        return Instantiate(Resources.Load(path, typeof(GameObject)) as GameObject);
    }
    public static Sprite LoadSprite(string path) {
        return Resources.Load(path, typeof(Sprite)) as Sprite;
    }
    public static TextMesh LoadTextMesh(string path, string value) {
        var textMesh = Instantiate(Resources.Load(path, typeof(TextMesh)) as
TextMesh);
        textMesh.text = value;
        return textMesh;
    }
}
```

```
class Constants {

    public const int RIGHT_MOUSE_BUTTON = 1;

    public const int MIDDLE_MOUSE_BUTTON = 2;

    public const float CAMERA_MOVEMENT_SPEED = 1f;

    public const float CAMERA_SCROLL_SPEED = 2f;

    public const int MAX_SCROLL = 10;

    public const int MIN_SCROLL = 80;

    public const float HEXAGON_LAYER = 0f;

    public const float LANDSCAPE_LAYER = -1f;

    public const float FEATURE_LAYER = -2f;

    public const float UNIT_LAYER = -3f;

    public const float TEXT_LAYER = -4f;

    public const float HIGHLIGHTER_LAYER = -5f;

    public const string PREFAB_PATH_TEXT_BASIC = "Prefabs/Text/Basic";

    public const string PREFAB_PATH_HEXAGON = "Prefabs/Hexagon/Base";

    public const string PREFAB_PATH_FOREST = "Prefabs/Feature/Forest";

    public const string PREFAB_PATH_PLAIN = "Prefabs/Landscape/Plain";

    public const string PREFAB_PATH_HILL = "Prefabs/Landscape/Hill";

    public const string PREFAB_PATH_MOUNTAIN = "Prefabs/Landscape/Mountain";

    public const string PREFAB_PATH_INFANTRY = "Prefabs/Unit/Infantry";

    public const string PREFAB_PATH_HIGHLIGHTER_CALCULATED =
"Prefabs/Highlighter/Calculated";

    public const string PREFAB_PATH_HIGHLIGHTER_RECALCULATED =
"Prefabs/Highlighter/Recalculated";

    public const string SPRITE_PATH_HEXAGON_BASE = "Sprites/Hexagon/Base";

    public const string SPRITE_PATH_HEXAGON_SELECTED = "Sprites/Hexagon/Selected";

    public const float MOVE_COST_PLAIN = 1;

    public const float MOVE_COST_HILL = 2;

    public const float MOVE_COST_MOUNTAIN = 8;

    public const float MOVE_COST_FOREST = 1;

    public const int MOVE_POINTS_INFANTRY = 3;

}


using System.Collections.Generic;
```

```
using UnityEngine;
public class GameLoader : MonoBehaviour {
    public enum Sizes {
        Small = 20, Medium = 35, Large = 50
    }
    public Sizes size = Sizes.TestExtraSmall;
    public GameObject endTurnButton;
    public int playerCount = 1;
    public int hillGenerationRate = 30; public int mountainGenerationRate = 10;
    public int forestGenerationRate = 30; public int unitGenerationRate = 30;
    public bool oneUnitTest = false;
    public bool displayCoordinates = false;
    private HexagonFactory hexagonFactory; private FeatureFactory featureFactory;
    private LandscapeFactory landscapeFactory; private UnitFactory unitFactory;
    private Map map = new Map();
    private void Start() {
        var sizeX = (int) size * 2; var sizeY = (int) size;
        var mouseInputHandler = new MouseInputHandler(ref map);
        hexagonFactory = new HexagonFactory(mouseInputHandler);
        landscapeFactory = new LandscapeFactory();
        featureFactory = new FeatureFactory();
        unitFactory = new UnitFactory();
        endTurnButton.GetComponent<EndTurnHandler>().map = map;
        var players = GeneratePlayers();
        var hexagons = GenerateHexagons(sizeX, sizeY);
        GenerateLandscape(ref hexagons, sizeX, sizeY);
        GenerateFeatures(ref hexagons, sizeX, sizeY);
        GenerateUnitsForPlayer(ref hexagons, sizeX, sizeY, players[0
map.SetHexagons(hexagons);
        map.SetPlayers(players);
    }
    private List<Player> GeneratePlayers() {
        var players = new List<Player>();
```

```
        for (var i = 0; i < playerCount; i++) {
            players.Add(new Player("Player" + i));
        }
        return players;
    }
    private Hexagon[,] GenerateHexagons(int sizeX, int sizeY) {
        var hexagons = new Hexagon[sizeX, sizeY];
        for (var x = 0; x < sizeX; x++) {
            for (var y = 0; y < sizeY; y++) {
                var coord = new Coordinate<int>(x, y);
                hexagons[x, y] = hexagonFactory.NewHexagon(coord,
displayCoordinates);
            }
        }
        return hexagons;
    }
    private void GenerateLandscape(ref Hexagon[,] hexagons, int sizeX, int sizeY) {
        for (var x = 0; x < sizeX; x++) {
            for (var y = 0; y < sizeY; y++) {
                GenerateSingleLandscape(ref hexagons[x, y]);
            }
        }
    }
    private void GenerateSingleLandscape(ref Hexagon hexagon) {
        if (DiceRoller.Roll(hillGenerationRate)) {
            hexagon.landscape = landscapeFactory.NewHill(ref hexagon);
        } else if (DiceRoller.Roll(mountainGenerationRate)) {
            hexagon.landscape = landscapeFactory.NewMountain(ref hexagon);
        } else {
            hexagon.landscape = landscapeFactory.NewPlain(ref hexagon);
        }
    }
    private void GenerateFeatures(ref Hexagon[,] hexagons, int sizeX, int sizeY) {
```

```
        for (var x = 0; x < sizeX; x++) {
            for (var y = 0; y < sizeY; y++) {
                GenerateSingleFeature(ref hexagons[x, y]);
            }
        }
    }
    private void GenerateSingleFeature(ref Hexagon hexagon) {
        if (!hexagon.landscape.name.Contains("Mountain") &&
DiceRoller.Roll(forestGenerationRate)) {
            hexagon.feature = featureFactory.NewForestFeature(ref hexagon);
        }
    }
    private void GenerateUnitsForPlayer(ref Hexagon[,] hexagons, int sizeX, int
sizeY, Player player) {
        if (oneUnitTest) {
            hexagons[0, 0].unit = unitFactory.NewInfantry(ref hexagons[0, 0], ref
player, new DStarLitePathFinder(map));
        } else {
            for (var x = 1; x < sizeX; x++) {
                for (var y = 1; y < sizeY; y++) {
                    GenerateSingleUnitForPlayer(ref hexagons[x, y], player);
                }
            }
        }
    }
    private void GenerateSingleUnitForPlayer(ref Hexagon hexagon, Player player) {
        if (!hexagon.landscape.name.Contains("Mountain") &&
DiceRoller.Roll(unitGenerationRate)) {
            hexagon.unit = unitFactory.NewInfantry(ref hexagon, ref player, new
DStarLitePathFinder(map));
        }
    }
}
```

ДОДАТОК Б

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	103 с.
	Інші документи	
2.	Презентаційні матеріали	плакатів

.	.	Прізвище та ініціали.	Підп.	Дата	
Розробив		Дворцов Д.В.			Розробка компонентів мультиагентної системи пошуку шляху у динамічних середах
Перевірив		Шергін В.Л.			
Н.контр.		Малєєва І.А.			
Затв.		Філатов В.О.			

Шифр групи	Код напр./спец.
СШІм-19-2	122
ХНУРЕ кафедра ШІ	