RogueShooter – All Scripts

Generated: 2025-10-06 07.38 UTC
Files: 88
Scanned: Assets/scripts

RogueShooter – All Scripts

Assets/scripts/Camera/CameraController.cs

```
using UnityEngine;
using Unity.Cinemachine;

// <summary>
// This script controls the camera movement, rotation, and zoom in a Unity game using the Cinemachine package.
// It allows the player to move the camera using WASD keys, rotate it using Q and E keys, and zoom in and out using the mouse scroll wheel.
// The camera follows a target object with a specified offset, and the zoom level is clamped to a minimum and maximum value.
// </summary>
public class CameraController : MonoBehaviour
{
    private const float MIN_FOLLOW_Y_OFFSET = 2f;
    private const float MAX_FOLLOW_Y_OFFSET = 18f;//12f;

    public static CameraController Instance { get; private set; }
    [SerializeField] private CinemachineCamera cinemachineCamera;

    private CinemachineFollow cinemachineFollow;
    private Vector3 targetFollowOffset;

    private float moveSpeed = 10f;
    private float rotationSpeed = 100f;
    private float zoomSpeed = 5f;

    private void Awake()
    {
        if (Instance != null)
        {
            Debug.LogError("CameraController: More than one CameraController in the scene! " + transform + " - " + Instance);
            Destroy(gameObject);
            return;
        }

        Instance = this;
    }

    private void Start()
    {
        cinemachineFollow = cinemachineCamera.GetComponent<CinemachineFollow>();
        targetFollowOffset = cinemachineFollow.FollowOffset;
    }

    private void Update()
    {
        HandleMovement(moveSpeed);
        HandleRotation(rotationSpeed);
        HandleZoom(zoomSpeed);
    }

    private void HandleMovement(float moveSpeed)
    {
        Vector2 inputMoveDirection = InputManager.Instance.GetCameraMoveVector();
        Vector3 moveVector = transform.forward * inputMoveDirection.y + transform.right * inputMoveDirection.x;
        transform.position += moveSpeed * Time.deltaTime * moveVector;
    }

    private void HandleRotation(float rotationSpeed)
    {
        Vector3 rotationVector = new Vector3(0, 0, 0);
        rotationVector.y = InputManager.Instance.GetCameraRotateAmount();
        transform.eulerAngles += rotationSpeed * Time.deltaTime * rotationVector;
    }

    private void HandleZoom(float zoomSpeed)
    {
        float zoomIncreaseAmount = 1f;
        targetFollowOffset.y += InputManager.Instance.GetCameraZoomAmount() * zoomIncreaseAmount;

        targetFollowOffset.y = Mathf.Clamp(targetFollowOffset.y, MIN_FOLLOW_Y_OFFSET, MAX_FOLLOW_Y_OFFSET);
        cinemachineFollow.FollowOffset = Vector3.Lerp(cinemachineFollow.FollowOffset, targetFollowOffset, Time.deltaTime * zoomSpeed);
    }


    public float GetCameraHeight()
    {
        return targetFollowOffset.y;
    }

}
```

Assets/scripts/Camera/CameraManager.cs

```
using System;
using UnityEngine;

public class CameraManager : MonoBehaviour
{
    [SerializeField] private GameObject actionCameraGameObject;

    [SerializeField] private float actionCameraVerticalPosition = 2.5f;
    private void Start()
    {
      //  BaseAction.OnAnyActionStarted += BaseAction_OnAnyActionStarted;
      //  BaseAction.OnAnyActionCompleted += BaseAction_OnAnyActionCompleted;

      //  HideActionCamera();
    }

    void OnEnable()
    {
        BaseAction.OnAnyActionStarted += BaseAction_OnAnyActionStarted;
        BaseAction.OnAnyActionCompleted += BaseAction_OnAnyActionCompleted;
        HideActionCamera();
    }

    void OnDisable()
    {
        BaseAction.OnAnyActionStarted -= BaseAction_OnAnyActionStarted;
        BaseAction.OnAnyActionCompleted -= BaseAction_OnAnyActionCompleted;
    }

    private void ShowActionCamera()
    {
        actionCameraGameObject.SetActive(true);
    }

    private void HideActionCamera()
    {
        actionCameraGameObject.SetActive(false);
    }

    private void BaseAction_OnAnyActionStarted(object sender, EventArgs e)
    {
        switch (sender)
        {
            case ShootAction shootAction:
                Unit shooterUnit = shootAction.GetUnit();
                Unit targetUnit = shootAction.GetTargetUnit();

                Vector3 cameraCharacterHeight = Vector3.up * actionCameraVerticalPosition; //1.7f;
                Vector3 shootDir = (targetUnit.GetWorldPosition() - shooterUnit.GetWorldPosition()).normalized;
                float shoulderOffsetAmount = 0.5f;
                Vector3 shoulderOffset = Quaternion.Euler(0, 90, 0) * shootDir * shoulderOffsetAmount;
                Vector3 actionCameraPosition =
                    shooterUnit.GetWorldPosition() +
                    cameraCharacterHeight +
                    shoulderOffset +
                    (shootDir * -1);

                actionCameraGameObject.transform.position = actionCameraPosition;
                actionCameraGameObject.transform.LookAt(targetUnit.GetWorldPosition() + cameraCharacterHeight);
                ShowActionCamera();
                break;
        }
    }

    private void BaseAction_OnAnyActionCompleted(object sender, EventArgs e)
    {
        switch (sender)
        {
            case ShootAction shootAction:
                HideActionCamera();
                break;
        }
    }

}
```

Assets/scripts/Camera/FloorVisibility.cs

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

public class FloorVisibility : MonoBehaviour
{

    [SerializeField] private bool dynamicFloorPosition;
    [SerializeField] private List<Renderer> ignoreRendererList;
    private HashSet<Renderer> ignoreSet;
    private Renderer[] rendererArray;
    private int floor;
    private bool? lastVisible;           // vältä turhat muutokset
    private Unit unit;                   // jos kohde on Unit tai sen alla
    private bool forceHidden;            // ulkoinen lukko (esim. kuolema)

    private void Awake()
    {
        rendererArray = GetComponentsInChildren<Renderer>(true);
        unit = GetComponentInParent<Unit>(); // tai GetComponent<Unit>() jos scripti istuu suoraan Unitissa

        if (unit != null)
        {
            // reagoi heti piilotukseen/poistoon
            unit.OnHiddenChangedEvent += OnUnitHiddenChanged;
            forceHidden = unit.IsHidden();
        }

        ignoreSet = new HashSet<Renderer>(ignoreRendererList);
    }

    private void Start()
    {
        floor = LevelGrid.Instance.GetFloor(transform.position);
        Recompute();
    }

    private void OnDestroy()
    {
        if (unit != null) unit.OnHiddenChangedEvent -= OnUnitHiddenChanged;
    }

    private void Update()
    {
        if (dynamicFloorPosition)
        {
            floor = LevelGrid.Instance.GetFloor(transform.position);
        }

        Recompute();
    }
    private void Recompute()
    {
        // 1) kamerakorkeuteen perustuva perusnäkyvyys
        float cameraHeight = CameraController.Instance.GetCameraHeight();
        float floorHeightOffset = 2f;
        bool cameraWantsVisible = (cameraHeight > LevelGrid.FLOOR_HEIGHT * floor + floorHeightOffset) || floor == 0;

        // 2) unitin piilotus "lukitsee" näkymättömäksi
        bool visible = cameraWantsVisible && !forceHidden;

        if (lastVisible.HasValue && lastVisible.Value == visible) return; // ei muutosta
        lastVisible = visible;

        ApplyVisible(visible);
    }

    private void ApplyVisible(bool visible)
    {
        foreach (var r in rendererArray)
        {
            if (!r) continue;
            if (ignoreSet.Contains(r)) continue;
            r.enabled = visible;
        }
    }

    // Jos haluat ulkopuolelta pakottaa piiloon (esim. ragdollin spawner tms.)
    public void SetForceHidden(bool hidden)
    {
        forceHidden = hidden;
        Recompute();
    }

    private void OnUnitHiddenChanged(bool hidden)
    {
        forceHidden = hidden;
        Recompute();
    }

    public void AddIgnore(Renderer r)
    {
        ignoreRendererList.Add(r);
        ignoreSet.Add(r);
    }
    public void RemoveIgnore(Renderer r)
    {
        ignoreRendererList.Remove(r);
        ignoreSet.Remove(r);
    }
}
```

Assets/scripts/Camera/Look At Camera.cs

```
using UnityEngine;

/// <summary>
/// Turn wordUI elemenets ( Like Unit Health and action points) toward to camera.
/// </summary>
public class LookAtCamera : MonoBehaviour
{
    [SerializeField] private bool invert;

    private Transform cameraTransform;

    private void Awake()
    {
        cameraTransform = Camera.main.transform;
    }

    private void LateUpdate()
    {
        if (invert)
        {
            Vector3 dirToCamera = (cameraTransform.position - transform.position).normalized;
            transform.LookAt(transform.position + dirToCamera * -1);
        } else
        {
            transform.LookAt(cameraTransform);
        }
    }
}
```

Assets/scripts/Camera/ScreenShake.cs

```
/*
using Unity.Cinemachine;
using UnityEngine;


public class ScreenShake : MonoBehaviour
{
    public static ScreenShake Instance { get; private set; }


    private CinemachineImpulseSource cinemachineImpulseSource;

    private void Awake()
    {

        // Ensure that there is only one instance in the scene
        if (Instance != null)
        {
            Debug.LogError("ScreenShake: More than one ScreenShake in the scene!" + transform + " " + Instance);
            Destroy(gameObject);
            return;
        }

        Instance = this;

        cinemachineImpulseSource = GetComponent<CinemachineImpulseSource>();
    }

    public void Shake(float intensity = 1f)
    {
        cinemachineImpulseSource.GenerateImpulse(intensity);
    }
}
*/

using Unity.Cinemachine;
using UnityEngine;


public class ScreenShake : MonoBehaviour
{
    public static ScreenShake Instance { get; private set; }

    [SerializeField]
    private CinemachineImpulseSource cinemachineRecoilImpulseSource;

    [SerializeField]
    private CinemachineImpulseSource cinemachineExplosiveImpulseSource;

    private void Awake()
    {

        // Ensure that there is only one instance in the scene
        if (Instance != null)
        {
            Debug.LogError("ScreenShake: More than one ScreenShake in the scene!" + transform + " " + Instance);
            Destroy(gameObject);
            return;
        }

        Instance = this;

        // cinemachineRecoilImpulseSource = GetComponent<CinemachineImpulseSource>();
    }

    public void ExplosiveCameraShake(float ShakeStrength)
    {
        cinemachineExplosiveImpulseSource.GenerateImpulse(ShakeStrength);
    }

    public void RecoilCameraShake(float ShakeStrength)
    {
        cinemachineRecoilImpulseSource.GenerateImpulse(ShakeStrength);
    }
}
```

Assets/scripts/DebuggingAndTesting/PathDiagHotkey.cs

```
using UnityEngine;

public class PathDiagHotkey : MonoBehaviour
{
    public KeyCode dumpKey = KeyCode.O;
    public KeyCode resetKey = KeyCode.P;

    void Update()
    {
        var diag = PathfindingDiagnostics.Instance;
        if (diag == null) return;

        if (Input.GetKeyDown(dumpKey))
        {
            Debug.Log(
                $"[PathDiag] Samples={diag.SamplesCount} | Avg={diag.AvgMs:F3} ms | P50={diag.P50Ms:F3} ms | P95={diag.P95Ms:F3} ms | Calls={diag.CallsTotal} | 
OK={diag.SuccessesTotal} | Fail={diag.FailuresTotal}"
            );
        }

        if (Input.GetKeyDown(resetKey))
        {
            diag.ResetStats();
            Debug.Log("[PathDiag] Reset");
        }
    }
}
```

Assets/scripts/DebuggingAndTesting/PathfindingDiagnostics.cs

```
#if PERFORMANCE_DIAG

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

[DefaultExecutionOrder(-10000)]
public class PathfindingDiagnostics : MonoBehaviour
{
    public static PathfindingDiagnostics Instance { get; private set; }

    [Header("On/Off")]
    public bool enabledRuntime = false;     // kytkin pelissä

    [Header("Window")]
    public int windowSize = 200;            // montako viimeisintä mittausta pidetään

    // Näkyvät lukemat
    public int SamplesCount => samples.Count;
    public double AvgMs { get; private set; }
    public double P95Ms { get; private set; }
    public double P50Ms { get; private set; } // mediaani
    public int CallsTotal { get; private set; }
    public int SuccessesTotal { get; private set; }
    public int FailuresTotal => CallsTotal - SuccessesTotal;

    struct Sample { public double ms; public bool success; public int pathLen; public int expanded; }
    readonly Queue<Sample> samples = new Queue<Sample>();

    void Awake()
    {
        if (Instance != null) { Destroy(gameObject); return; }
        Instance = this;
        DontDestroyOnLoad(gameObject);
    }

    public void AddSample(double ms, bool success, int pathLen, int expanded)
    {
        if (!enabledRuntime) return;

        CallsTotal++;
        if (success) SuccessesTotal++;

        samples.Enqueue(new Sample { ms = ms, success = success, pathLen = pathLen, expanded = expanded });
        while (samples.Count > windowSize) samples.Dequeue();

        RecomputeStats();
    }

    void RecomputeStats()
    {
        if (samples.Count == 0)
        {
            AvgMs = P95Ms = P50Ms = 0;
            return;
        }

        double sum = 0;
        List<double> arr = new List<double>(samples.Count);
        foreach (var s in samples) { sum += s.ms; arr.Add(s.ms); }

        arr.Sort();
        AvgMs = sum / samples.Count;
        P50Ms = Percentile(arr, 0.50);
        P95Ms = Percentile(arr, 0.95);
    }

    static double Percentile(List<double> sorted, double p)
    {
        if (sorted.Count == 0) return 0;
        double idx = (sorted.Count - 1) * p;
        int lo = (int)Math.Floor(idx);
        int hi = (int)Math.Ceiling(idx);
        if (lo == hi) return sorted[lo];
        double w = idx - lo;
        return sorted[lo] * (1 - w) + sorted[hi] * w;
    }

    // Helppo nollaus napista
    public void ResetStats()
    {
        samples.Clear();
        CallsTotal = 0;
        SuccessesTotal = 0;
        AvgMs = P95Ms = P50Ms = 0;
    }
}

#else

using UnityEngine;

// Stubbi, joka kääntyy release-buildiin mutta ei tee mitään
public class PathfindingDiagnostics : MonoBehaviour
{
    public static PathfindingDiagnostics Instance => null;
    public bool enabledRuntime => false;
    public void AddSample(double ms, bool success, int pathLen, int expanded) { }
    public void ResetStats() { }
}

#endif
```

Assets/scripts/DebuggingAndTesting/ScreenLogger.cs

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

public class ScreenLogger : MonoBehaviour
{
   static ScreenLogger inst;
    TextMeshProUGUI text;
    readonly Queue<string> lines = new Queue<string>();
    [Range(1,100)] public int maxLines = 100;

    void Awake()
    {
        if (inst != null) { Destroy(gameObject); return; }
        inst = this;
        DontDestroyOnLoad(gameObject);

        // Canvas
        var canvasGO = new GameObject("ScreenLogCanvas");
        var canvas = canvasGO.AddComponent<Canvas>();
        canvas.renderMode = RenderMode.ScreenSpaceOverlay;
        canvas.sortingOrder = 9999;

        // Text
        var tgo = new GameObject("Log");
        tgo.transform.SetParent(canvasGO.transform);
        var rt = tgo.AddComponent<RectTransform>();
        rt.anchorMin = new Vector2(0, 0);
        rt.anchorMax = new Vector2(1, 0);
        rt.pivot = new Vector2(0.5f, 0);
        rt.offsetMin = new Vector2(10, 10);
        rt.offsetMax = new Vector2(-10, 210);

        text = tgo.AddComponent<TextMeshProUGUI>();
        text.fontSize = 18;
        text.textWrappingMode = TextWrappingModes.NoWrap;

        Application.logMessageReceived += HandleLog;
    }

    void OnDestroy() { Application.logMessageReceived -= HandleLog; }

    void HandleLog(string msg, string stack, LogType type)
    {
        string prefix = type == LogType.Error || type == LogType.Exception ? "[ERR]" :
                        type == LogType.Warning ? "[WARN]" : "[LOG]";
        lines.Enqueue($"{System.DateTime.Now:HH:mm:ss} {prefix} {msg}");
        while (lines.Count > maxLines) lines.Dequeue();
        if (text != null) text.text = string.Join("\n", lines);
    }
}
```

Assets/scripts/DebuggingAndTesting/Testing.cs

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

/// <summary>
/// This class is responsible for testing the grid system and unit actions in the game.
/// It provides functionality to visualize the grid positions and interact with unit actions.
/// </summary>
public class Testing : MonoBehaviour
{

    [SerializeField] private Unit unit;
    private void Start()
    {

    }

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.T))
        {

            // ScreenShake.Instance.Shake(5f);

           // ScreenShake.Instance.RecoilCameraShake();

            //Show pathfind line
            /*
            GridPosition mouseGridPosition = LevelGrid.Instance.GetGridPosition(MouseWorld.GetMouseWorldPosition());
            GridPosition startGridPosition = new GridPosition(0, 0, 0);


            List<GridPosition> gridPositionList = PathFinding.Instance.FindPath(startGridPosition, startGridPosition, out int pathLeght, 6);

            for (int i = 0; i < gridPositionList.Count - 1; i++)
            {
                Debug.DrawLine(
                    LevelGrid.Instance.GetWorldPosition(gridPositionList[i]),
                    LevelGrid.Instance.GetWorldPosition(gridPositionList[i + 1]),
                    Color.white,
                    10f
                );
            }
            */
        }

        //Resetoi pelin alkamaan alusta.
        if (Input.GetKeyDown(KeyCode.R))
        {
            if (Mirror.NetworkServer.active) {
                ResetService.Instance.HardResetServerAuthoritative();
            } else if (Mirror.NetworkClient.active) {
                // käskytä serveriä
                ResetService.Instance.CmdRequestHardReset();
            } else {
                GameReset.HardReloadSceneKeepMode();
            }
        }

    }
}
```

Assets/scripts/Editor/PathfindingLinkMonoBehaviourEditor.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEditor;
[CustomEditor(typeof(PathfindingLinkMonoBehaviour))]
public class PathfindingLinkMonoBehaviourEditor : Editor
{
    private void OnSceneGUI()
    {
        PathfindingLinkMonoBehaviour pathfindingLinkMonoBehaviour = (PathfindingLinkMonoBehaviour)target;
        EditorGUI.BeginChangeCheck();
        Vector3 newLinkPositionA = Handles.PositionHandle(pathfindingLinkMonoBehaviour.linkPositionA, Quaternion.identity);
        Vector3 newLinkPositionB = Handles.PositionHandle(pathfindingLinkMonoBehaviour.linkPositionB, Quaternion.identity);
        if (EditorGUI.EndChangeCheck())
        {
            Undo.RecordObject(pathfindingLinkMonoBehaviour, "Change Link Position");
            pathfindingLinkMonoBehaviour.linkPositionA = newLinkPositionA;
            pathfindingLinkMonoBehaviour.linkPositionB = newLinkPositionB;
        }
    }
}
```

Assets/scripts/Enemy/EnemyAI.cs

```
using System;
using System.Collections;
using UnityEngine;
using Utp;

/// <summary>
/// Control EnemyAI. Go trough all posibble actions what current enemy Unit can do and chose the best one.
/// Listen to TurnSystem and when turn OnTurnChanged, AI state switch WaitingForEnemyTurn to the TakingTurn state
/// and try to find best action to all enemy Units. All enemy Unit do this independently based on
/// action values.
/// </summary>
public class EnemyAI : MonoBehaviour
{
    public static EnemyAI Instance { get; private set; }

    private enum State
    {
        WaitingForEnemyTurn,
        TakingTurn,
        Busy,
    }

    private State state;
    private float timer;

    void Awake()
    {
        state = State.WaitingForEnemyTurn;

        if (Instance != null && Instance != this) { Destroy(gameObject); return; }
        Instance = this;
    }

    private void Start()
    {

        if (GameModeManager.SelectedMode == GameMode.SinglePlayer)
        {
            TurnSystem.Instance.OnTurnChanged += TurnSystem_OnTurnChanged;
        }


        if (GameNetworkManager.Instance != null &&
        GameNetworkManager.Instance.GetNetWorkClientConnected() &&
        !GameNetworkManager.Instance.GetNetWorkServerActive())
        {
            // Coop gamemode using IEnumerator RunEnemyTurnCoroutine() trough the server. No local calls
            if (GameModeManager.SelectedMode == GameMode.CoOp)
                enabled = false;
        }
    }

    /*
    void OnEnable()
    {
        if (GameModeManager.SelectedMode == GameMode.SinglePlayer)
        {
            TurnSystem.Instance.OnTurnChanged += TurnSystem_OnTurnChanged;
        }
    }
    */

    void OnDisable()
    {
        if (GameModeManager.SelectedMode == GameMode.SinglePlayer)
        {
            TurnSystem.Instance.OnTurnChanged -= TurnSystem_OnTurnChanged;
        }
    }

    private void Update()
    {
        //NOTE! Only solo game!
        if (GameModeManager.SelectedMode != GameMode.SinglePlayer) return;
        if (TurnSystem.Instance.IsPlayerTurn()) return;

        //If game mode is SinglePlayer and is not PlayerTurn then runs Enemy AI.
        EnemyAITick(Time.deltaTime);
    }

    /// <summary>
    /// Enemy start taking actions after small waiting time.
    /// Update call this every frame.
    /// </summary>
    private bool EnemyAITick(float dt)
    {
        switch (state)
        {
            // It is Player turn so keep waiting untill TurnSystem_OnTurnChanged switch state to TakingTurn.
            case State.WaitingForEnemyTurn:
                return false;

            case State.TakingTurn:
                timer -= dt;
                if (timer <= 0f)
                {
                    //Return false when all Enemy Units have make they actions
                    if (SelectEnemyUnitToTakeAction(SetStateTakingTurn))
                    {
                        state = State.Busy;
                        return false;
                    }
                    else
                    {
                        // If enemy cant make actions. Return turn back to player.
                        // NOTE! In Coop mode CoopTurnCoordinator make this.
                        if (GameModeManager.SelectedMode == GameMode.SinglePlayer)
                        {
                            TurnSystem.Instance.NextTurn();
                        }

                        // Enemy AI switch back to waiting.
                        state = State.WaitingForEnemyTurn;
                        return true;
                    }
                }
                return false;

            case State.Busy:
                // When Enemy doing action just return.
                // Waiting c# Action call from base action and then call funktion SetStateTakingTurn()
                return false;
        }
        return false;
    }


    /// <summary>
    /// c# Action callback. SelectEnemyUnitToTakeAction use this and when action is ready. This occurs
    /// </summary>
    private void SetStateTakingTurn()
    {
        timer = 0.5f;
        state = State.TakingTurn;
    }

    /// <summary>
    /// Go through all enemy Units on EnemyUnit List and try to take action.
    /// </summary>
    private bool SelectEnemyUnitToTakeAction(Action onEnemyAIActionComplete)
    {
        foreach (Unit enemyUnit in UnitManager.Instance.GetEnemyUnitList())
        {
            if (enemyUnit == null)
            {
                Debug.LogWarning("[EnemyAI][UnitManager]EnemyUnit list is null:" + enemyUnit);
                continue;
            }
            if (TryTakeEnemyAIAction(enemyUnit, onEnemyAIActionComplete))
            {
                return true;
            }
        }
        return false;
    }

    /// <summary>
    /// Selected Unit Go through all possible actions what Enemy Unit can do
    /// and choosing the best one based on them action value.
    /// Then make action if have enough action points.
    /// </summary>
    private bool TryTakeEnemyAIAction(Unit enemyUnit, Action onEnemyAIActionComplete)
    {
        // Contains Gridposition and action value (How good action is)
        EnemyAIAction bestEnemyAIAction = null;

        BaseAction bestBaseAction = null;

        // Choosing the best action, based on them action value.
        foreach (BaseAction baseAction in enemyUnit.GetBaseActionsArray())
        {
            if (!enemyUnit.CanSpendActionPointsToTakeAction(baseAction))
            {
                // Enemy cannot afford this action.
                continue;
            }

            if (bestEnemyAIAction == null)
            {
                bestEnemyAIAction = baseAction.GetBestEnemyAIAction();
                bestBaseAction = baseAction;
            }
            else
            {
                // Go trough all actions and take the best one.
                EnemyAIAction testEnemyAIAction = baseAction.GetBestEnemyAIAction();
                if (testEnemyAIAction != null && testEnemyAIAction.actionValue > bestEnemyAIAction.actionValue)
                {
                    bestEnemyAIAction = baseAction.GetBestEnemyAIAction();
                    bestBaseAction = baseAction;
                }
            }
        }

        // Try to take action
        if (bestEnemyAIAction != null && enemyUnit.TrySpendActionPointsToTakeAction(bestBaseAction))
        {
            bestBaseAction.TakeAction(bestEnemyAIAction.gridPosition, onEnemyAIActionComplete);
            return true;
        }
        else
        {
            return false;
        }
    }
    /// <summary>
    /// When turn changed. Switch state to taking turn and enemy turn start.
    /// </summary>
    private void TurnSystem_OnTurnChanged(object sender, EventArgs e)
    {
        if (!TurnSystem.Instance.IsPlayerTurn())
        {
            state = State.TakingTurn;
            timer = 1f; // Small holding time before action.
        }
    }

    /// <summary>
    /// When playing online: (Coop mode) Server handle All AI actions.
    /// </summary>
    [Mirror.Server]
    public IEnumerator RunEnemyTurnCoroutine()
    {

        SetStateTakingTurn();

        while (true)
        {
            if (TurnSystem.Instance.IsPlayerTurn())
            {
                Debug.LogWarning("[EnemyAI] Players get turn before AI has ended own turn! This sould not be posibble");
                yield break;
            }

            bool finished = EnemyAITick(Time.deltaTime);
            if (finished)
                yield break; // AI-Turn ready. CoopTurnCoordinator continue and give turn back to players.

            yield return null; // wait one frame.
        }
    }
}
```

Assets/scripts/Enemy/EnemyAIAction.cs

```
using UnityEngine;

[System.Serializable]
public class EnemyAIAction
{
    public GridPosition gridPosition;
    public int actionValue;
}
```

Assets/scripts/GameLogic/BattleLogic/TurnSystem.cs

```
using System;
using UnityEngine;

public class TurnSystem : MonoBehaviour
{
    public static TurnSystem Instance { get; private set; }

    public event EventHandler OnTurnChanged;
    private int turnNumber = 1;
    private bool isPlayerTurn = true;

    private void Awake()
    {

        // Ensure that there is only one instance in the scene
        if (Instance != null)
        {
            Debug.LogError(" More than one TurnSystem in the scene!" + transform + " " + Instance);
            Destroy(gameObject);
            return;
        }
        Instance = this;
    }

    private void Start()
    {
        // Varmista, että alkutila lähetetään kaikille UI:lle
        PlayerLocalTurnGate.Set(isPlayerTurn); // true = Player turn alussa
        OnTurnChanged?.Invoke(this, EventArgs.Empty); // jos haluat myös muut UI:t liikkeelle
    }

    public void NextTurn()
    {
        // Tarkista pelimoodi
        if (GameModeManager.SelectedMode == GameMode.SinglePlayer)
        {
            Debug.Log("SinglePlayer NextTurn");
            turnNumber++;
            isPlayerTurn = !isPlayerTurn;

            OnTurnChanged?.Invoke(this, EventArgs.Empty);

            //Set Unit UI visibility
            PlayerLocalTurnGate.Set(isPlayerTurn);
        }
        else if (GameModeManager.SelectedMode == GameMode.CoOp)
        {
            Debug.Log("Co-Op mode: Proceeding to the next turn.");
            // Tee jotain erityistä CoOp-tilassa
        }
        else if (GameModeManager.SelectedMode == GameMode.Versus)
        {
            Debug.Log("Versus mode: Proceeding to the next turn.");
            // Tee jotain erityistä Versus-tilassa
        }


    }

    public int GetTurnNumber()
    {
        return turnNumber;
    }

    public bool IsPlayerTurn()
    {
        return isPlayerTurn;
    }

    // ForcePhase on serverin kutsuma. Päivittää vuoron ja kutsuu OnTurnChanged
    public void ForcePhase(bool isPlayerTurn, bool incrementTurnNumber)
    {
        if (incrementTurnNumber) turnNumber++;
        this.isPlayerTurn = isPlayerTurn;
        OnTurnChanged?.Invoke(this, EventArgs.Empty);
    }

    // Päivitä HUD verkon kautta (co-op)
    public void SetHudFromNetwork(int newTurnNumber, bool isPlayersPhase)
    {
        turnNumber = newTurnNumber;
        isPlayerTurn = isPlayersPhase;
        OnTurnChanged?.Invoke(this, EventArgs.Empty); // <- päivitää HUDin kuten SP:ssä
    }
}
```

Assets/scripts/GameLogic/InputManager.cs

```
#define USE_NEW_INPUT_SYSTEM
using UnityEngine;
using UnityEngine.InputSystem;

public class InputManager : MonoBehaviour
{
    public static InputManager Instance { get; private set; }

    private PlayerInputActions playerInputActions;

    private void Awake()
    {
        // Ensure that there is only one instance in the scene
        if (Instance != null)
        {
            Debug.LogError("InputManager: More than one ImputManager in the scene!" + transform + " " + Instance);
            Destroy(gameObject);
            return;
        }
        Instance = this;

#if USE_NEW_INPUT_SYSTEM
        playerInputActions = new PlayerInputActions();
        // Voit halutessasi enablettaa koko collectionin:
        // playerInputActions.Enable();
        playerInputActions.Player.Enable();
#endif
    }
#if USE_NEW_INPUT_SYSTEM
    private void OnDisable()
    {
        // Vähintään tämä: disabloi kaikki käytössä olevat mapit
        if (playerInputActions != null)
        {
            // Jos käytät vain Player-mapia:
            playerInputActions.Player.Disable();
            // Tai koko collection:
            // playerInputActions.Disable();
        }
    }

    private void OnDestroy()
    {
        // Vapauta resurssit -> poistaa finalizer-varoituksen
        playerInputActions?.Dispose();
        playerInputActions = null;

        if (Instance == this) Instance = null;
    }
#endif
```

```
    public Vector2 GetMouseScreenPosition()
    {
#if USE_NEW_INPUT_SYSTEM
        return Mouse.current.position.ReadValue();
#else
        return Input.mousePosition;
#endif
    }

    public bool IsMouseButtonDownThisFrame()
    {
#if USE_NEW_INPUT_SYSTEM
        return playerInputActions.Player.Click.WasPressedThisFrame();
#else
        return Input.GetMouseButtonDown(0);
#endif
    }

    public Vector2 GetCameraMoveVector()
    {
#if USE_NEW_INPUT_SYSTEM
        return playerInputActions.Player.CameraMovement.ReadValue<Vector2>();
#else
        Vector2 inputMoveDirection = new Vector2(0, 0);
        if (Input.GetKey(KeyCode.W))
        {
            inputMoveDirection.y = +1f;
        }
        if (Input.GetKey(KeyCode.S))
        {
            inputMoveDirection.y = -1f;
        }
        if (Input.GetKey(KeyCode.A))
        {
            inputMoveDirection.x = -1f;
        }
        if (Input.GetKey(KeyCode.D))
        {
            inputMoveDirection.x = +1f;
        }

        return inputMoveDirection;
#endif
    }

    public float GetCameraRotateAmount()
    {
#if USE_NEW_INPUT_SYSTEM
        return playerInputActions.Player.CameraRotate.ReadValue<float>();
#else
        float rotateAmount = 0;
        if (Input.GetKey(KeyCode.Q))
        {
            rotateAmount = +1f;
        }
        if (Input.GetKey(KeyCode.E))
        {
            rotateAmount = -1f;
        }

        return rotateAmount;
#endif
    }

    public float GetCameraZoomAmount()
    {
#if USE_NEW_INPUT_SYSTEM
        return playerInputActions.Player.CameraZoom.ReadValue<float>();
#else
        float zoomAmount = 0f;
        if (Input.mouseScrollDelta.y > 0)
        {
            zoomAmount = -1f;
        }
        if (Input.mouseScrollDelta.y < 0)
        {
            zoomAmount = +1f;
        }

        return zoomAmount;
#endif
    }
}
```

Assets/scripts/GameLogic/MouseWorld.cs

```
using UnityEngine;

/// <summary>
/// This class is responsible for handling mouse interactions in the game world.
/// It provides a method to get the mouse position in the world space based on the camera's perspective.
/// </summary>

public class MouseWorld : MonoBehaviour
{
    private static MouseWorld instance;
    [SerializeField] private LayerMask mousePlaneLayerMask;

    private void Awake()
    {
        instance = this;
    }

    public static Vector3 GetMouseWorldPosition()
    {
        Ray ray = Camera.main.ScreenPointToRay(InputManager.Instance.GetMouseScreenPosition());
        Physics.Raycast(ray, out RaycastHit raycastHit, float.MaxValue, instance.mousePlaneLayerMask);
        return raycastHit.point;
    }

    /// <summary>
    ///  Ignore non visible objects, floors and walls what FloorVisibily has set to hidden.
    /// </summary>
    public static Vector3 GetPositionOnlyHitVisible()
    {
        Ray ray = Camera.main.ScreenPointToRay(InputManager.Instance.GetMouseScreenPosition());
        RaycastHit[] raycastHitArray = Physics.RaycastAll(ray, float.MaxValue, instance.mousePlaneLayerMask);
        System.Array.Sort(raycastHitArray,
        (a, b) => a.distance.CompareTo(b.distance));

        foreach (RaycastHit raycastHit in raycastHitArray)
        {
            if (raycastHit.transform.TryGetComponent(out Renderer renderer))
            {
                if (renderer.enabled)
                {
                    return raycastHit.point;
                }
            }
        }
        return Vector3.zero;
    }
}
```

Assets/scripts/GameLogic/Player/PlayerController.cs

```
using System;
using Mirror;
using UnityEngine;

///<sumary>
/// PLayerController handles per-player state in a networked game.
/// Each connected player has one PlayerController instance attached to PlayerController GameObject prefab
/// It tracks whether the player has ended their turn and communicates with the UI.
///</sumary>
public class PlayerController : NetworkBehaviour
{

    [SyncVar] public bool hasEndedThisTurn;

    public static PlayerController Local; // helppo viittaus UI:lle

    public override void OnStartLocalPlayer()
    {
        base.OnStartLocalPlayer();
        Local = this;
    }

    // UI-nappi kutsuu tätä (vain local player)
    public void ClickEndTurn()
    {
        if (!isLocalPlayer) return;
        if (hasEndedThisTurn) return;
        if (NetTurnManager.Instance && NetTurnManager.Instance.phase != TurnPhase.Players) return;
        CmdEndTurn();
    }

    [Command(requiresAuthority = true)]
    void CmdEndTurn()
    {
        if (hasEndedThisTurn) return;
        hasEndedThisTurn = true;

        // Estä kaikki toiminnot clientillä
        TargetNotifyCanAct(connectionToClient, false);

        // Varmista myös että koordinaattori löytyy serveripuolelta:
        if (NetTurnManager.Instance == null)
        {
            Debug.LogWarning("[PC][SERVER] NetTurnManager.Instance is NULL on server!");
            return;
        }

        NetTurnManager.Instance.ServerPlayerEndedTurn(netIdentity.netId);
    }
    // Server kutsuu tämän kierroksen alussa nollatakseen tilan
    [Server]
    public void ServerSetHasEnded(bool v)
    {
        hasEndedThisTurn = v;
        TargetNotifyCanAct(connectionToClient, !v);
    }

    [TargetRpc]
    void TargetNotifyCanAct(NetworkConnectionToClient __, bool canAct)
    {

        // Update End Turn Button
        var ui = FindFirstObjectByType<TurnSystemUI>();
        if (ui != null)
            ui.SetCanAct(canAct);
        if (!canAct) ui.SetTeammateReady(false, null);

        // Lock/Unlock UnitActionSystem input
        if (UnitActionSystem.Instance != null)
        {
            if (canAct) UnitActionSystem.Instance.UnlockInput();
            else UnitActionSystem.Instance.LockInput();
        }

        // Set AP visibility in versus game
        PlayerLocalTurnGate.Set(canAct);
    }

}
```

Assets/scripts/GameLogic/Player/PlayerLocalTurnGate.cs

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

/// <summary>
/// Static gate that tracks whether the local player turn is. (e.g., enabling/disabling UI).
/// Other systems can subscribe to the <see cref="LocalPlayerTurnChanged"/> event to update their state
/// </summary>
///
public static class PlayerLocalTurnGate
{
    // public static int PlayerReady { get; private set; }

    // public static event Action<int> OnPlayerReadyChanged;
    /// <summary>
    /// Gets whether the local player can currently act.
    /// </summary>
    public static bool LocalPlayerTurn { get; private set; }

    /// <summary>
    /// Event fired whenever the <see cref="LocalPlayerTurn"/> state changes.
    /// The bool argument indicates the new state.
    /// </summary>
    public static event Action<bool> LocalPlayerTurnChanged;

    /// <summary>
    /// Updates the <see cref="LocalPlayerTurn"/> state.
    /// If the value changes, invokes <see cref="LocalPlayerTurnChanged"/> to notify listeners.
    /// </summary>
    /// <param name="canAct">True if the player may act; false otherwise.</param>
    public static void Set(bool canAct)
    {
        if (LocalPlayerTurn == canAct) return;
        LocalPlayerTurn = canAct;
        LocalPlayerTurnChanged?.Invoke(LocalPlayerTurn);
    }

    public static void SetCanAct(bool canAct)
    {
        LocalPlayerTurn = canAct;
        LocalPlayerTurnChanged?.Invoke(LocalPlayerTurn);
    }

}
```

Assets/scripts/GameModes/GameModeManager.cs

```
using UnityEngine;
using Utp;

/// <summary>
/// This class is responsible for managing the game mode
/// It checks if the game is being played online or offline and spawns units accordingly.
/// </summary>
public enum GameMode { SinglePlayer, CoOp, Versus }
public class GameModeManager : MonoBehaviour
{
    public static GameMode SelectedMode { get; private set; } = GameMode.SinglePlayer;

    public static void SetSinglePlayer() => SelectedMode = GameMode.SinglePlayer;
    public static void SetCoOp() => SelectedMode = GameMode.CoOp;
    public static void SetVersus() => SelectedMode = GameMode.Versus;

    void Start()
    {
        // if game is offline, spawn singleplayer units
        if (!GameNetworkManager.Instance.IsNetworkActive())
        {
            SpawnUnits();
        }
        else
        {
            Debug.Log("Game is online, waiting for host/client to spawn units.");
        }
    }

    private void SpawnUnits()
    {
        if (SelectedMode == GameMode.SinglePlayer)
        {

            SpawnUnitsCoordinator.Instance.SpwanSinglePlayerUnits();
            return;
        }
    }
}
```

Assets/scripts/GameModes/GameReset.cs

```
using UnityEngine.SceneManagement;

public static class GameReset
{
    public static void HardReloadSceneKeepMode()
    {
        // GameModeManager.SelectedMode säilyy, jos se on staattinen / DontDestroyOnLoad
        var scene = SceneManager.GetActiveScene().name;
        SceneManager.LoadScene(scene);
    }
}
```

Assets/scripts/GameObjects/DestructibleObject.cs

```
using System;
using Unity.Mathematics;
using UnityEngine;
using Mirror;
using System.Collections;

public class DestructibleObject : NetworkBehaviour
{
    public static event EventHandler OnAnyDestroyed;

    private GridPosition gridPosition;
    [SerializeField] private Transform objectDestroyPrefab;
    [SerializeField] private int health = 3;

    // To prevent multiple destruction events
    private bool isDestroyed;

    private bool _walkabilitySet;
    void Awake()
    {
        isDestroyed = false;
    }

    private void Start()
    {
        gridPosition = LevelGrid.Instance.GetGridPosition(transform.position);
        TryMarkBlocked();
    }

    /// <summary>
    /// Marks the grid position as blocked if not already set.
    /// </summary>
    private void TryMarkBlocked()
    {
        if (_walkabilitySet) return;

        if (PathFinding.Instance != null)
        {
            PathFinding.Instance.SetIsWalkableGridPosition(gridPosition, false);
            _walkabilitySet = true;
        }
        else
        {
            // jos PathFinding käynnistyy myöhemmin (scene-reload + spawn)
            StartCoroutine(DeferBlockOneFrame());
        }
    }

    private IEnumerator DeferBlockOneFrame()
    {
        yield return null; // 1 frame
        if (PathFinding.Instance != null)
        {
            Debug.Log("Later update: Deferring walkability set for destructible object at " + gridPosition);
            PathFinding.Instance.SetIsWalkableGridPosition(gridPosition, false);
            _walkabilitySet = true;
        }
    }

    public GridPosition GetGridPosition()
    {
        return gridPosition;
    }

    public void Damage(int damageAmount, Vector3 hitPosition)
    {
        if (isDestroyed) return;

        health -= damageAmount;
        if (health > 0) return;

        int overkill = math.abs(health) + 1;
        health = 0;
        isDestroyed = true;

        if (isServer)
        {
            RpcPlayDestroyFx(hitPosition, overkill);
            RpcSetSoftHidden(true);
            StartCoroutine(DestroyAfter(0.30f));
            return;
        }

        // Offline (ei serveriä eikä clienttia)
        if (!NetworkClient.active && !NetworkServer.active)
        {
            PlayDestroyFx(hitPosition, overkill);
            SetSoftHiddenLocal(true);
            StartCoroutine(DestroyAfter(0.30f));
        }
    }

    private void PlayDestroyFx(Vector3 hitPosition, int overkill)
    {
        var t = Instantiate(objectDestroyPrefab, transform.position, Quaternion.identity);
        ApplyPushForceToChildren(t, 10f * overkill, hitPosition, 10f);
        OnAnyDestroyed?.Invoke(this, EventArgs.Empty);
    }

    [ClientRpc] private void RpcPlayDestroyFx(Vector3 hitPosition, int overkill)
    {
        // Clientit: toista sama paikallisesti
        PlayDestroyFx(hitPosition, overkill);
    }

    private void ApplyPushForceToChildren(Transform root, float pushForce, Vector3 pushPosition, float PushRange)
    {
        foreach (Transform child in root)
        {
            if (child.TryGetComponent<Rigidbody>(out Rigidbody childRigidbody))
            {
                childRigidbody.AddExplosionForce(pushForce, pushPosition, PushRange);
            }

            ApplyPushForceToChildren(child, pushForce, pushPosition, PushRange);
        }
    }

    private IEnumerator DestroyAfter(float seconds)
    {
        yield return new WaitForSeconds(seconds);

        if (isServer) NetworkServer.Destroy(gameObject);
        else Destroy(gameObject);
        OnAnyDestroyed?.Invoke(this, EventArgs.Empty);
    }

    [ClientRpc]
    private void RpcSetSoftHidden(bool hidden)
    {
        SetSoftHiddenLocal(hidden);
    }

    private void SetSoftHiddenLocal(bool hidden)
    {
        foreach (var r in GetComponentsInChildren<Renderer>(true))
            r.enabled = !hidden;

        foreach (var c in GetComponentsInChildren<Collider>(true))
            c.enabled = !hidden;
    }
}
```

Assets/scripts/GameObjects/Door.cs

```
using UnityEngine;
using Mirror;
using System;

public class Door : NetworkBehaviour, IInteractable
{
    [Header("State")]
    [SyncVar(hook = nameof(OnIsOpenChanged))]
    [SerializeField] private bool isOpen = false;   // alkutila scene-objektille

    [SerializeField] string openParam = "IsOpen";
    [SerializeField] float interactDuration = 0.5f;


    private GridPosition gridPosition;
    private Animator animator;

    // Interact-viiveen hallinta (vain kutsujan koneella UI/turn-rytmitystä varten)
    private Action onInteractComplete;
    private bool isActive;
    private float timer;

    private static bool NetOffline => !NetworkClient.active && !NetworkServer.active;

    private void Awake()
    {
        animator = GetComponent<Animator>();

        // Pakota alkupose heti oikein (ei välähdyksiä)
        animator.SetBool("IsOpen", isOpen);
        animator.Play(isOpen ? "DoorOpen" : "DoorClose", 0, 1f);
        animator.Update(0f);
    }

    private void Start()
    {
        gridPosition = LevelGrid.Instance.GetGridPosition(transform.position);
        LevelGrid.Instance.SetInteractableAtGridPosition(gridPosition, this);

        // AINA: päivitä käveltävyys tämän hetken tilan mukaan
        if (PathFinding.Instance != null)
        {
            PathFinding.Instance.SetIsWalkableGridPosition(gridPosition, isOpen);
        }
    }

    private void Update()
    {
        if (!isActive) return;
        timer -= Time.deltaTime;
        if (timer <= 0f)
        {
            isActive = false;
            onInteractComplete?.Invoke();
            onInteractComplete = null;
        }
    }

    // KUTSUTAAN InteractActionista (sekä offline, host että puhdas client)
    public void Interact(Action onInteractComplete)
    {
        // Gate (estää spämmin)
        if (isActive) return;

        this.onInteractComplete = onInteractComplete;
        isActive = true;
        timer = interactDuration; // haluttu viive actionille

        if (NetOffline)
        {
            // SINGLEPLAYER: vaihda paikallisesti
            ToggleLocal();
        }
        else if (isServer)
        {
            // HOST / SERVER: vaihda suoraan serverillä
            ToggleServer();
        }
        else
        {
            // PUHDAS CLIENT: pyydä serveriä
            CmdToggleServer();
        }
    }

    [Command(requiresAuthority = false)]
    private void CmdToggleServer()
    {
        ToggleServer();
    }

    [Server]
    private void ToggleServer()
    {
        isOpen = !isOpen; // Tämä käynnistää hookin kaikilla
        // EI suoraa animator-kutsua täällä; hook hoitaa sen kauniisti
    }

    private void ToggleLocal()
    {
        // Offline-haara: päivitä animaatio ja pathfinding paikallisesti
        isOpen = !isOpen;
        ApplyAnimator(isOpen);
        PathFinding.Instance.SetIsWalkableGridPosition(gridPosition, isOpen);

    }

    // SyncVar hook – ajetaan kaikilla kun isOpen muuttuu serverillä
    private void OnIsOpenChanged(bool oldVal, bool newVal)
    {
        ApplyAnimator(newVal);

        // Pathfinding vain serverillä (tai offline Startissa/ToggleLocalissa)
        if (PathFinding.Instance != null)
            PathFinding.Instance.SetIsWalkableGridPosition(gridPosition, newVal);
    }

    private void ApplyAnimator(bool open)
    {
        animator.SetBool(openParam, open);
    }

    // Nämä jätetään jos muu koodi tarvitsee suoraviivaisia kutsuja
    public void OpenDoor()
    {
        if (NetOffline || NetworkServer.active)
        {
            isOpen = true; // käynnistää hookin vain serverillä; offline: päivitä itse
            if (NetOffline)
            {
                ApplyAnimator(true);
                PathFinding.Instance.SetIsWalkableGridPosition(gridPosition, true);
            }
        }
    }

    public void CloseDoor()
    {
        if (NetOffline || NetworkServer.active)
        {
            isOpen = false;
            if (NetOffline)
            {
                ApplyAnimator(false);
                PathFinding.Instance.SetIsWalkableGridPosition(gridPosition, false);
            }
        }
    }
}
```

Assets/scripts/GameObjects/IInreractable.cs

```
using System;
using UnityEngine;

public interface IInteractable
{
   void Interact(Action onInteractComplete);
}
```

Assets/scripts/GameObjects/InteractableItem.cs

```
using System;
using UnityEngine;
using Mirror;
public class InteractableItem : NetworkBehaviour, IInteractable
{
    [Header("State")]
    [SyncVar(hook = nameof(OnIsInteractChanged))]
    [SerializeField] private bool isGreen;

    [Header("Visuals")]
    [SerializeField] private Material greenMaterial;
    [SerializeField] private Material redMaterial;
    [SerializeField] private MeshRenderer meshRenderer;

    [Header("Interact")]
    [SerializeField] private float interactDuration = 0.5f;

    private GridPosition gridPosition;
    private Action onInteractComplete;
    private bool isActive;
    private float timer;

    private static bool NetOffline => !NetworkClient.active && !NetworkServer.active;

    void Awake()
    {
        // Pakota alkupose heti oikein (ei välähdyksiä)
        if (!meshRenderer) meshRenderer = GetComponentInChildren<MeshRenderer>();
        SetVisualFromState(isGreen);
    }
    private void Start()
    {
        gridPosition = LevelGrid.Instance.GetGridPosition(transform.position);
        LevelGrid.Instance.SetInteractableAtGridPosition(gridPosition, this);
       // SetColorRed();
    }
    private void Update()
    {
        if (!isActive) return;

        timer -= Time.deltaTime;
        if (timer <= 0f)
        {
            isActive = false;
            onInteractComplete?.Invoke();
            onInteractComplete = null;
        }
    }

    private void SetColorGreen()
    {
        isGreen = true;
        meshRenderer.material = greenMaterial;
    }

    private void SetColorRed()
    {
        isGreen = false;
        meshRenderer.material = redMaterial;
    }

    public void Interact(Action onInteractComplete)
    {
        this.onInteractComplete = onInteractComplete;
        isActive = true;
        timer = interactDuration;

        if (NetOffline)
        {
            // SINGLEPLAYER: vaihda paikallisesti
            ToggleLocal();
        }
        else if (isServer)
        {
            // HOST / SERVER: vaihda suoraan serverillä
            ToggleServer();
        }
        else
        {
            // PUHDAS CLIENT: pyydä serveriä
            CmdToggleServer();
        }
    }

    private void ToggleLocal()
    {
        isGreen = !isGreen;
        SetVisualFromState(isGreen);
    }

    [Server]
    private void ToggleServer()
    {
        // SERVER: muuta vain tila; visuaali päivittyy hookista kaikkialla
        isGreen = !isGreen;
        SetVisualFromState(isGreen); // valinnainen: tekee serverille välittömän visuaalin ilman uutta SyncVar-kirjoitusta
    }

    [Command(requiresAuthority = false)]
    void CmdToggleServer() => ToggleServer();

    private void OnIsInteractChanged(bool oldValue, bool newVal)
    {
        SetVisualFromState(newVal);
    }

    private void SetVisualFromState(bool state)
    {
        if (!meshRenderer) return;
        meshRenderer.material = state ? greenMaterial : redMaterial;
    }
}
```

Assets/scripts/GameObjects/ObjectSpawnPlaceHolder.cs

```
using Mirror;
using UnityEngine;
/// <summary>
/// This class is responsible for spawning objects in the game.
/// This object is only placeholder, which spawns the actual object and then destroys itself.
/// Because spawning must be done by the server, this object must exist on the server.
/// </summary>
public class ObjectSpawnPlaceHolder : MonoBehaviour
{
    [SerializeField] private GameObject objectPrefab;
    public GameObject Prefab => objectPrefab;

    private void Start()
    {
        // OFFLINE: ei verkkoa -> luo paikallisesti (näkyy heti)
        if (!NetworkClient.active && !NetworkServer.active)
        {
            Instantiate(objectPrefab, transform.position, transform.rotation);
            Destroy(gameObject);
        }

        // PUHDAS CLIENT: serveri spawnaa oikean → poista placeholder heti
        if (NetworkClient.active && !NetworkServer.active)
        {
            Destroy(gameObject);
            return;
        }

    }

    public void CreteObject()
    {
        // ONLINE: server luo ja spawnnaa
        if (NetworkServer.active)
        {
            Debug.Log($"[objectSpawnPoint] Spawning object at {transform.position}");
            var go = Instantiate(objectPrefab, transform.position, transform.rotation);
            NetworkServer.Spawn(go);
            Destroy(gameObject);
            return;
        }
    }
}
```

Assets/scripts/Grid/GridDebugObject.cs

```
using UnityEngine;
using TMPro;

// <summary>
// This script is used to display the grid object information in the scene view.
// </summary>

public class GridDebugObject : MonoBehaviour
{
    [SerializeField] private TextMeshPro textMeshPro;

    private object gridObject;
    public virtual void SetGridObject(object gridObject)
    {
        this.gridObject = gridObject;
    }
    protected virtual void Update()
    {
        textMeshPro.text = gridObject.ToString();
    }

}
```

Assets/scripts/Grid/GridObject.cs

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

// <summary>
// This class represents a grid object in the grid system.
// It contains a list of units that are present in the grid position.
// It also contains a reference to the grid system and the grid position.
// </summary>
public class GridObject
{
    private GridSystem<GridObject> gridSystem;
    private GridPosition gridPosition;
    private List<Unit> unitList;
    private IInteractable interactable;

    public GridObject(GridSystem<GridObject> gridSystem, GridPosition gridPosition)
    {
        this.gridSystem = gridSystem;
        this.gridPosition = gridPosition;
        unitList = new List<Unit>();
    }

    public override string ToString()
    {
        string unitListString = "";
        foreach (Unit unit in unitList)
        {
            unitListString += unit + "\n";
        }
        return gridPosition.ToString() + "\n" + unitListString;
    }

    public void AddUnit(Unit unit)
    {
        unitList.Add(unit);
    }

    public void RemoveUnit(Unit unit)
    {
        unitList.Remove(unit);
    }

    public List<Unit> GetUnitList()
    {
        unitList.RemoveAll(u => u == null);
        return unitList;
    }

    public bool HasAnyUnit()
    {
        // Poista tuhotut viitteet (Unity-null huomioiden)
        unitList.RemoveAll(u => u == null);
        return unitList.Count > 0;
    }

    public Unit GetUnit()
    {
        /*
        if (HasAnyUnit())
        {
            return unitList[0];
        }
        else
        {
            return null;
        }
        */
        // Siivoa ja palauta ensimmäinen elossa oleva

        for (int i = unitList.Count - 1; i >= 0; i--)
        {
            if (unitList[i] == null) { unitList.RemoveAt(i); continue; }
        }
        return unitList.Count > 0 ? unitList[0] : null;
    }

    public IInteractable GetInteractable()
    {
        return interactable;
    }

    public void SetInteractable(IInteractable interactable)
    {
        this.interactable = interactable;
    }
}
```

Assets/scripts/Grid/GridPosition.cs

```
using System;
using NUnit.Framework;

// <summary>
// This struct represents a position in a grid system.
// It contains two integer values, x and z, which represent the coordinates of the position in the grid.
// It also contains methods for comparing two GridPosition objects, adding and subtracting them, and converting them to a string representation.
// </summary>
public struct GridPosition:IEquatable<GridPosition>
{
    public int x;
    public int z;

    public int floor;

    public GridPosition(int x, int z, int floor)
    {
        this.x = x;
        this.z = z;
        this.floor = floor;
    }

    public override bool Equals(object obj)
    {
        return obj is GridPosition position &&
        x == position.x &&
        z == position.z &&
        floor == position.floor;
    }

    public bool Equals(GridPosition other)
    {
        return this == other;
    }

    public override int GetHashCode()
    {
        return HashCode.Combine(x, z, floor);
    }

    public override string ToString()
    {
        return $"(x:{x}, z:{z}, floor:{floor})";
    }

    public static bool operator ==(GridPosition a, GridPosition b)
    {
        return a.x == b.x && a.z == b.z && a.floor == b.floor;
    }
    public static bool operator !=(GridPosition a, GridPosition b)
    {
        return !(a == b);
    }

    public static GridPosition operator +(GridPosition a, GridPosition b)
    {
        return new GridPosition(a.x + b.x, a.z + b.z, a.floor + b.floor);
    }

    public static GridPosition operator -(GridPosition a, GridPosition b)
    {
        return new GridPosition(a.x - b.x, a.z - b.z, a.floor - b.floor);
    }

}
```

Assets/scripts/Grid/GridSystem.cs

```
using System;
using UnityEngine;

/// <summary>
/// This class represents a grid system in a 2D space.
/// It contains methods to create a grid, convert between grid and world coordinates,
/// and manage grid objects.
/// </summary>

public class GridSystem<TGridObject>
{
    private int width;
    private int height;
    private float cellSize;
    private int floor;

    private float floorHeigth;
    private TGridObject[,] gridObjectsArray;

    public GridSystem(int width, int height, float cellSize, int floor, float floorHeigth, Func<GridSystem<TGridObject>, GridPosition, TGridObject> createGridObject)
    {
        this.width = width;
        this.height = height;
        this.cellSize = cellSize;
        this.floor = floor;
        this.floorHeigth = floorHeigth;

        gridObjectsArray = new TGridObject[width, height];

        for (int x = 0; x < width; x++)
        {
            for (int z = 0; z < height; z++)
            {
                GridPosition gridPosition = new GridPosition(x, z, floor);
                gridObjectsArray[x, z] = createGridObject(this, gridPosition);
            }
        }
    }

/// Purpose: This method converts grid coordinates (x, z) to world coordinates.
/// It multiplies the grid coordinates by the cell size to get the world position.
    public Vector3 GetWorldPosition(GridPosition gridPosition)
    {

        return new Vector3(gridPosition.x, 0, gridPosition.z) * cellSize +
        new Vector3(0, gridPosition.floor, 0) * floorHeigth;
    }

/// Purpose: This is used to find the grid position of a unit in the grid system.
/// It is used to check if the unit is within the bounds of the grid system.
/// It converts the world position to grid coordinates by dividing the world position by the cell size.
    public GridPosition GetGridPosition(Vector3 worldPosition)
    {
        return new GridPosition( Mathf.RoundToInt(worldPosition.x/cellSize),
        Mathf.RoundToInt(worldPosition.z/cellSize),
        floor);
    }

/// Purpose: This method creates debug objects in the grid system for visualization purposes.
/// It instantiates a prefab at each grid position and sets the grid object for that position.
    public void CreateDebugObjects(Transform debugPrefab)
    {
        for (int x = 0; x< width; x++)
        {
            for(int z = 0; z < height; z++)
            {
                GridPosition gridPosition = new GridPosition(x, z, floor);
                Transform debugTransform = GameObject.Instantiate(debugPrefab, GetWorldPosition(gridPosition), Quaternion.identity);
                GridDebugObject gridDebugObject = debugTransform.GetComponent<GridDebugObject>();
                gridDebugObject.SetGridObject(GetGridObject(gridPosition));
            }
        }
    }

/// Purpose: This method returns the grid object at a specific grid position.
/// It is used to get the grid object for a specific position in the grid system.
    public TGridObject GetGridObject(GridPosition gridPosition)
    {
        return gridObjectsArray[gridPosition.x, gridPosition.z];
    }

/// Purpose: This method checks if a grid position is valid within the grid system.
/// It checks if the x and z coordinates are within the bounds of the grid width and height.
    public bool IsValidGridPosition(GridPosition gridPosition)
    {
        return gridPosition.x >= 0 &&
                gridPosition.x < width &&
                gridPosition.z >= 0 &&
                gridPosition.z < height &&
                gridPosition.floor == floor;
    }

    public int GetWidth()
    {
        return width;
    }
    public int GetHeight()
    {
        return height;
    }

}
```

Assets/scripts/Grid/GridSystemVisual.cs

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

/// <summary>
/// This class is responsible for visualizing the grid system in the game.
/// It creates a grid of visual objects that represent the grid positions.
/// </summary>
public class GridSystemVisual : MonoBehaviour
{

    public static GridSystemVisual Instance { get; private set; }

    [Serializable]
    public struct GridVisualTypeMaterial
    {
        public GridVisualType gridVisualType;
        public Material material;
    }
    public enum GridVisualType
    {
        white,
        Blue,
        Red,
        RedSoft,
        Yellow
    }

    /// Purpose: This prefab is used to create the visual representation of each grid position.
    [SerializeField] private Transform gridSystemVisualSinglePrefab;
    [SerializeField] private List<GridVisualTypeMaterial> gridVisualTypeMaterialList;

    /// Purpose: This array holds the visual objects for each grid position.
    private GridSystemVisualSingle[,,] gridSystemVisualSingleArray;

    private void Awake()
    {

        ///  Purpose: Ensure that there is only one instance in the scene
        if (Instance != null)
        {
            Debug.LogError("More than one GridSystemVisual in the scene!" + transform + " " + Instance);
            Destroy(gameObject);
            return;
        }

        Instance = this;
    }
    private void Start()
    {
        gridSystemVisualSingleArray = new GridSystemVisualSingle[
            LevelGrid.Instance.GetWidth(),
            LevelGrid.Instance.GetHeight(),
            LevelGrid.Instance.GetFloorAmount()
            ];

        /// Purpose: Create a grid of visual objects that represent the grid positions.
        /// It instantiates a prefab at each grid position and sets the grid object for that position.
        for (int x = 0; x < LevelGrid.Instance.GetWidth(); x++)
        {
            for (int z = 0; z < LevelGrid.Instance.GetHeight(); z++)
            {
                for (int floor = 0; floor < LevelGrid.Instance.GetFloorAmount(); floor++)
                {
                    GridPosition gridPosition = new(x, z, floor);
                    Transform gridSystemVisualSingleTransform = Instantiate(gridSystemVisualSinglePrefab, LevelGrid.Instance.GetWorldPosition(gridPosition), 
Quaternion.identity);
                    gridSystemVisualSingleArray[x, z, floor] = gridSystemVisualSingleTransform.GetComponent<GridSystemVisualSingle>();
                }
            }
        }

        UnitActionSystem.Instance.OnSelectedActionChanged += UnitActionSystem_OnSelectedActionChanged;
        UnitActionSystem.Instance.OnBusyChanged += UnitActionSystem_OnBusyChanged;
      // LevelGrid.Instance.onAnyUnitMoveGridPosition += LevelGrid_onAnyUnitMoveGridPosition;

        UpdateGridVisuals();
    }

    /*
    void OnEnable()
    {
        UnitActionSystem.Instance.OnSelectedActionChanged += UnitActionSystem_OnSelectedActionChanged;
        LevelGrid.Instance.onAnyUnitMoveGridPosition += LevelGrid_onAnyUnitMoveGridPosition;
    }
    */

    void OnDisable()
    {
        UnitActionSystem.Instance.OnSelectedActionChanged -= UnitActionSystem_OnSelectedActionChanged;
       // LevelGrid.Instance.onAnyUnitMoveGridPosition -= LevelGrid_onAnyUnitMoveGridPosition;
    }

    public void HideAllGridPositions()
    {
        for (int x = 0; x < LevelGrid.Instance.GetWidth(); x++)
        {
            for (int z = 0; z < LevelGrid.Instance.GetHeight(); z++)
            {
                for (int floor = 0; floor < LevelGrid.Instance.GetFloorAmount(); floor++)
                {
                    gridSystemVisualSingleArray[x, z, floor].Hide();
                }
            }
        }
    }

    private void ShowGridPositionRange(GridPosition gridPosition, int range, GridVisualType gridVisualType)
    {

        List<GridPosition> gridPositionsList = new List<GridPosition>();

        for (int x = -range; x <= range; x++)
        {
            for (int z = -range; z <= range; z++)
            {
                GridPosition testGridPosition = gridPosition + new GridPosition(x, z, 0);

                if (!LevelGrid.Instance.IsValidGridPosition(testGridPosition))
                {
                    continue;
                }

                int testDistance = Mathf.Abs(x) + Mathf.Abs(z);
                if (testDistance > range)
                {
                    continue;
                }

                gridPositionsList.Add(testGridPosition);
            }
        }

        ShowGridPositionList(gridPositionsList, gridVisualType);
    }

    private void ShowGridPositionRangeSquare(GridPosition gridPosition, int range, GridVisualType gridVisualType)
    {

        List<GridPosition> gridPositionsList = new List<GridPosition>();

        for (int x = -range; x <= range; x++)
        {
            for (int z = -range; z <= range; z++)
            {
                GridPosition testGridPosition = gridPosition + new GridPosition(x, z, 0);

                if (!LevelGrid.Instance.IsValidGridPosition(testGridPosition))
                {
                    continue;
                }
                gridPositionsList.Add(testGridPosition);
            }
        }

        ShowGridPositionList(gridPositionsList, gridVisualType);
    }

    public void ShowGridPositionList(List<GridPosition> gridPositionList, GridVisualType gridVisualType)
    {
        foreach (GridPosition gridPosition in gridPositionList)
        {
            gridSystemVisualSingleArray[gridPosition.x, gridPosition.z, gridPosition.floor].
            Show(GetGridVisualTypeMaterial(gridVisualType));
        }
    }

    private void UpdateGridVisuals()
    {
        HideAllGridPositions();
        Unit selectedUnit = UnitActionSystem.Instance.GetSelectedUnit();
        if (selectedUnit == null) return;

        BaseAction selectedAction = UnitActionSystem.Instance.GetSelectedAction();

        GridVisualType gridVisualType;

        switch (selectedAction)
        {
            default:
            case MoveAction moveAction:
                gridVisualType = GridVisualType.white;
                break;
            case TurnTowardsAction turnTowardsAction:
                gridVisualType = GridVisualType.Blue;
                break;
            case ShootAction shootAction:
                gridVisualType = GridVisualType.Red;
                ShowGridPositionRange(selectedUnit.GetGridPosition(), shootAction.GetMaxShootDistance(), GridVisualType.RedSoft);
                break;
            case GranadeAction granadeAction:
                gridVisualType = GridVisualType.Yellow;
                break;
            case MeleeAction meleeAction:
                gridVisualType = GridVisualType.Red;
                ShowGridPositionRangeSquare(selectedUnit.GetGridPosition(), 1, GridVisualType.RedSoft);
                break;
            case InteractAction interactAction:
                gridVisualType = GridVisualType.Blue;
                break;

        }
        ShowGridPositionList(
            selectedAction.GetValidGridPositionList(), gridVisualType);

    }

    private void UnitActionSystem_OnSelectedActionChanged(object sender, EventArgs e)
    {
        UpdateGridVisuals();
    }

    private void LevelGrid_onAnyUnitMoveGridPosition(object sender, EventArgs e)
    {
        UpdateGridVisuals();
    }

    private void UnitActionSystem_OnBusyChanged(object sender, bool e)
    {
        UpdateGridVisuals();
    }

    private Material GetGridVisualTypeMaterial(GridVisualType gridVisualType)
    {
        foreach (GridVisualTypeMaterial gridVisualTypeMaterial in gridVisualTypeMaterialList)
        {
            if (gridVisualTypeMaterial.gridVisualType == gridVisualType)
            {
                return gridVisualTypeMaterial.material;
            }
        }
        Debug.LogError("Cloud not find GridVisualTypeMaterial for GridVisualType" + gridVisualType);
        return null;
    }
}
```

Assets/scripts/Grid/GridSystemVisualSingle.cs

```
using UnityEngine;

/// <summary>
/// This class is responsible for visualizing a single grid position in the game.
/// It contains a MeshRenderer component that is used to show or hide the visual representation of the grid position.
/// </summary>
public class GridSystemVisualSingle : MonoBehaviour
{
    [SerializeField] private MeshRenderer meshRenderer;

    public void Show(Material material)
    {
        meshRenderer.enabled = true;
        meshRenderer.material = material;
    }
    public void Hide()
    {
        meshRenderer.enabled = false;
    }
}
```

Assets/scripts/Grid/LevelGrid.cs

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

/// @file LevelGrid.cs
/// @brief Core grid management system for RogueShooter.
///
/// The LevelGrid defines and manages the tactical grid used by all gameplay systems.
/// It stores spatial occupancy data, translates between world-space and grid-space coordinates,
/// and provides the structural backbone for the pathfinding and edge-baking systems.
///
/// ### Overview
/// Each level in RogueShooter is represented as one or more layered grids (floors).
/// Every grid cell corresponds to a physical area in the game world and may contain
/// references to units, obstacles, or other gameplay entities. The LevelGrid keeps
/// this data synchronized with the actual scene state and provides efficient lookup
/// and update operations.
///
/// ### System integration
/// - **LevelGrid** – Manages spatial layout, unit occupancy, and coordinate conversions.
/// - **EdgeBaker** – Uses LevelGrid data (width, height, cell size, floor count) to detect edge obstacles.
/// - **PathFinding** – Queries LevelGrid to determine walkable areas and world↔grid mapping for A* searches.
///
/// ### Key features
/// - Multi-floor grid architecture with configurable width, height, and cell size.
/// - Fast world↔grid coordinate conversion for unit and object placement.
/// - Real-time occupancy tracking of all units on the grid.
/// - Scene rebuild capability (`RebuildOccupancyFromScene`) for reinitializing unit positions after reload.
/// - Event-driven notifications for unit movement (`onAnyUnitMoveGridPosition`).
///
/// ### Why this exists in RogueShooter
/// - The game’s turn-based, tile-based design requires precise spatial logic independent of Unity’s physics.
/// - Provides a unified “source of truth” for spatial relationships used by both AI and player systems.
/// - Keeps the game’s tactical layer deterministic, debuggable, and efficient.
///
/// In summary, this file defines the foundational grid layer of RogueShooter’s tactical engine,
/// acting as the shared coordinate and occupancy system for all movement, visibility, and interaction logic.

/// <summary>
/// This class is responsible for managing the game's grid system.
/// It keeps track of the units on the grid and their positions.
/// It provides methods to add, remove, and move units on the grid.
/// Note: This class Script Execution Order is set to be executed after UnitManager.cs. High priority.
/// </summary>
public class LevelGrid : MonoBehaviour
{
    public static LevelGrid Instance { get; private set; }

    public const float FLOOR_HEIGHT = 4f;
    public event EventHandler onAnyUnitMoveGridPosition;
    [SerializeField] private Transform debugPrefab;
    // [SerializeField] private bool debugVisible = true;
    [SerializeField] private int width;
    [SerializeField] private int height;
    [SerializeField] private float cellSize;
    [SerializeField] private int floorAmount;

    private List<GridSystem<GridObject>> gridSystemList;

    private void Awake()
    {

        // Ensure that there is only one instance in the scene
        if (Instance != null)
        {
            Debug.LogError("LevelGrid: More than one LevelGrid in the scene!" + transform + " " + Instance);
            Destroy(gameObject);
            return;
        }
        Instance = this;

        gridSystemList = new List<GridSystem<GridObject>>(floorAmount);

        for (int floor = 0; floor < floorAmount; floor++)
        {
            var gridSystem = new GridSystem<GridObject>(
                width, height, cellSize, floor, FLOOR_HEIGHT,
                (GridSystem<GridObject> g, GridPosition gridPosition) => new GridObject(g, gridPosition)
                );
            //gridSystem.CreateDebugObjects(debugPrefab);
            gridSystemList.Add(gridSystem); // NullReferenceException: Object reference not set to an instance of an object!

        }
    }

    private void Start()
    {
        PathFinding.Instance.Setup(width, height, cellSize, floorAmount);
    }

    public GridSystem<GridObject> GetGridSystem(int floor)
    {
        if (floor < 0 || floor >= gridSystemList.Count) { Debug.LogError($"Invalid floor {floor}"); return null; }
        return gridSystemList[floor];
    }

    public int GetFloor(Vector3 worldPosition)
    {
        return Mathf.RoundToInt(worldPosition.y / FLOOR_HEIGHT);
    }
    public void AddUnitAtGridPosition(GridPosition gridPosition, Unit unit)
    {
        GridObject gridObject = GetGridSystem(gridPosition.floor).GetGridObject(gridPosition);
        gridObject.AddUnit(unit);
    }

    public List<Unit> GetUnitListAtGridPosition(GridPosition gridPosition)
    {
        GridObject gridObject = GetGridSystem(gridPosition.floor).GetGridObject(gridPosition);
        if (gridObject != null)
        {
            return gridObject.GetUnitList();
        }
        return null;
    }

    public IInteractable GetInteractableAtGridPosition(GridPosition gridPosition)
    {
        GridObject gridObject = GetGridSystem(gridPosition.floor).GetGridObject(gridPosition);
        if (gridObject != null)
        {
            return gridObject.GetInteractable();
        }
        return null;
    }

    public void SetInteractableAtGridPosition(GridPosition gridPosition, IInteractable interactable)
    {
        GridObject gridObject = GetGridSystem(gridPosition.floor).GetGridObject(gridPosition);
        gridObject?.SetInteractable(interactable);

    }

    public void RemoveUnitAtGridPosition(GridPosition gridPosition, Unit unit)
    {
        GridObject gridObject = GetGridSystem(gridPosition.floor).GetGridObject(gridPosition);
        gridObject.RemoveUnit(unit);
    }

    public void UnitMoveToGridPosition(GridPosition fromGridPosition, GridPosition toGridPosition, Unit unit)
    {
        RemoveUnitAtGridPosition(fromGridPosition, unit);
        AddUnitAtGridPosition(toGridPosition, unit);
        onAnyUnitMoveGridPosition?.Invoke(this, EventArgs.Empty);
    }

    public GridPosition GetGridPosition(Vector3 worldPosition)
    {
        int floor = GetFloor(worldPosition);
        return GetGridSystem(floor).GetGridPosition(worldPosition);
    }
    public Vector3 GetWorldPosition(GridPosition gridPosition)
    {
        return GetGridSystem(gridPosition.floor).GetWorldPosition(gridPosition);
    }

    public bool IsValidGridPosition(GridPosition gridPosition)
    {
        if (gridPosition.floor < 0 || gridPosition.floor >= floorAmount)
        {
            return false;
        }
        return GetGridSystem(gridPosition.floor).IsValidGridPosition(gridPosition);
    }

    public int GetWidth() => GetGridSystem(0).GetWidth();

    public int GetHeight() => GetGridSystem(0).GetHeight();

    public int GetFloorAmount() => floorAmount;

    public float GetCellSize() => cellSize;

    public bool HasAnyUnitOnGridPosition(GridPosition gridPosition)
    {
        GridObject gridObject = GetGridSystem(gridPosition.floor).GetGridObject(gridPosition);
        return gridObject.HasAnyUnit();
    }

    public Unit GetUnitAtGridPosition(GridPosition gridPosition)
    {
        GridObject gridObject = GetGridSystem(gridPosition.floor).GetGridObject(gridPosition);
        return gridObject.GetUnit();
    }

    public void ClearAllOccupancy()
    {
        if (gridSystemList == null) return;

        for (int floor = 0; floor < gridSystemList.Count; floor++)
        {
            var grid = gridSystemList[floor];
            if (grid == null) continue;

            for (int x = 0; x < grid.GetWidth(); x++)
            {
                for (int z = 0; z < grid.GetHeight(); z++)
                {
                    var gp = new GridPosition(x, z, floor);
                    var gridObj = grid.GetGridObject(gp);
                    gridObj?.GetUnitList()?.Clear();
                }
            }
        }
    }

    /// <summary>
    /// Rebuilds all grid occupancy data by scanning the current scene for active units.
    ///
    /// What it does:
    /// - Clears all existing unit occupancy from the <see cref="LevelGrid"/>.
    /// - Finds every active <see cref="Unit"/> in the scene.
    /// - Converts each unit’s world position into a grid position and re-registers it.
    ///
    /// Why this exists in RogueShooter:
    /// - Used after a scene or level is (re)loaded to ensure that the grid accurately reflects
    ///   the current in-scene unit placements.
    /// - Called by systems like <see cref="GameModeSelectUI"/> and <see cref="ServerBootstrap"/>
    ///   to synchronize game state after spawning or initialization events.
    ///
    /// Implementation notes:
    /// - Intended for runtime reinitialization, not per-frame updates.
    /// - Safe to call at any time; automatically rebuilds the occupancy layer from scratch.
    /// </summary>
    public void RebuildOccupancyFromScene()
    {
        ClearAllOccupancy();
        var units = FindObjectsByType<Unit>(FindObjectsSortMode.None);
        foreach (var u in units)
        {
            var gp = GetGridPosition(u.transform.position);
            AddUnitAtGridPosition(gp, u);
        }
    }
}
```

Assets/scripts/Grid/PathFindingDebugGridObject.cs

```
using TMPro;
using UnityEngine;

public class PathFindingDebugGridObject : GridDebugObject
{
    [SerializeField] private TextMeshPro gCostText;
    [SerializeField] private TextMeshPro hCostText;
    [SerializeField] private TextMeshPro fCostText;

    [SerializeField] private SpriteRenderer isWalkableSpriteRenderer;

    private PathNode pathNode;
    public override void SetGridObject(object gridObject)
    {
        base.SetGridObject(gridObject);
        pathNode = (PathNode)gridObject;

    }

    protected override void Update()
    {
        base.Update();
        gCostText.text = pathNode.GetGCost().ToString();
        hCostText.text = pathNode.GetHCost().ToString();
        fCostText.text = pathNode.GetFCost().ToString();
        isWalkableSpriteRenderer.color = pathNode.GetIsWalkable() ? Color.green : Color.red;

    }
}
```

Assets/scripts/Helpers/AllUnitsList.cs

```
using Mirror;
using UnityEngine;

[DisallowMultipleComponent]
public class FriendlyUnit : NetworkBehaviour {}

[DisallowMultipleComponent]
public class EnemyUnit : NetworkBehaviour {}
```

Assets/scripts/Helpers/AuthorityHelper.cs

```
using Mirror;

public static class AuthorityHelper
{
    /// <summary>
    /// Checks if the given NetworkBehaviour has local control.
    /// Prevents the player from controlling the object if they are not the owner.
    /// </summary>
    public static bool HasLocalControl(NetworkBehaviour netBehaviour)
    {
        return NetworkClient.isConnected && !netBehaviour.isOwned;
    }
}
```

Assets/scripts/Helpers/FieldCleaner.cs

```
using System.Linq;
using UnityEngine;
using UnityEngine.SceneManagement;
using Utp;

public class FieldCleaner : MonoBehaviour
{
    public static void ClearAll()
    {
        // Varmista: älä yritä siivota puhtaalta clientiltä verkossa
        if (GameNetworkManager.Instance != null &&
            GameNetworkManager.Instance.GetNetWorkClientConnected() &&
            !GameNetworkManager.Instance.GetNetWorkServerActive())
        {
            Debug.LogWarning("[FieldCleaner] Don't clear field from a pure client.");
            return;
        }

        // Find all friendly and enemy units (also inactive, just in case)
        var friendlies = Resources.FindObjectsOfTypeAll<FriendlyUnit>()
                          .Where(u => u != null && u.gameObject.scene.IsValid());
        var enemies = Resources.FindObjectsOfTypeAll<EnemyUnit>()
                          .Where(u => u != null && u.gameObject.scene.IsValid());

        foreach (var u in friendlies) Despawn(u.gameObject);
        foreach (var e in enemies) Despawn(e.gameObject);

        // Tyhjennä UnitManagerin listat (suojattu null-checkillä)
        UnitManager.Instance?.ClearAllUnitLists();

        // Nollaa myös ruudukon miehitys – sceneen jääneet objektit eivät jää kummittelemaan
        LevelGrid.Instance?.ClearAllOccupancy();
    }

    static void Despawn(GameObject go)
    {
        // if server is active, use Mirror's destroy; otherwise normal Unity Destroy
        if (GameNetworkManager.Instance.GetNetWorkServerActive())
        {
            GameNetworkManager.Instance.NetworkDestroy(go);
        }
        else
        {
            Destroy(go);
        }
    }

    public static void ReloadMap()
    {
        Debug.Log("[FieldCleaner] Reloading map.");
        SceneManager.LoadScene(SceneManager.GetActiveScene().name);
    }
}
```

Assets/scripts/LevelCreation/MapContentSpawner.cs

```
using Mirror;
using UnityEngine;

/// <summary>
/// Spawns map content such as destructible objects when the server starts.
/// </summary>
public class MapContentSpawner : NetworkBehaviour
{
    public override void OnStartServer()
    {
        base.OnStartServer();
        Debug.Log("[MapContentSpawner] OnStartServer - Spawning map content.");

        // Find all Destructibleobjects placeholders in the scene and spawn real destructible objects
        var spawnPoints = FindObjectsByType<ObjectSpawnPlaceHolder>(FindObjectsSortMode.None);
        foreach (var sp in spawnPoints)
        {
            sp.CreteObject();
        }
    }
}
```

Assets/scripts/LevelCreation/SpawnUnitsCoordinator.cs

```
using System.Linq;
using UnityEngine;
using Mirror;

public class SpawnUnitsCoordinator : MonoBehaviour
{
    public static SpawnUnitsCoordinator Instance { get; private set; }
    private bool enemiesSpawned;

    // --- Lisää luokan alkuun kentät ---
    [Header("Co-op squad prefabs")]
    public GameObject unitHostPrefab;      // -> UnitSolo
    public GameObject unitClientPrefab;    // -> UnitSolo Player 2

    [Header("Enemy spawn (Co-op)")]
    public GameObject enemyPrefab;

    [Header("Spawn positions (world coords on your grid)")]
    public Vector3[] hostSpawnPositions = {
            new Vector3(0, 0, 0),
            new Vector3(2, 0, 0),
        };
    public Vector3[] clientSpawnPositions = {
            new Vector3(0, 0, 6),
            new Vector3(2, 0, 6),
        };
    public Vector3[] enemySpawnPositions = {
            new Vector3(4, 0, 8),
            new Vector3(6, 0, 8),
        };

    void Awake()
    {
        if (Instance != null && Instance != this) { Destroy(gameObject); return; }
        Instance = this;
    }


    public GameObject[] SpawnPlayersForNetwork(NetworkConnectionToClient conn, bool isHost)
    {
        GameObject unitPrefab = GetUnitPrefabForPlayer(isHost);
        Vector3[] spawnPoints = GetSpawnPositionsForPlayer(isHost);

        if (unitPrefab == null)
        {
            Debug.LogError($"[NM] {(isHost ? "unitHostPrefab" : "unitClientPrefab")} puuttuu!");
            return null;
        }
        if (spawnPoints == null || spawnPoints.Length == 0)
        {
            Debug.LogError($"[NM] {(isHost ? "hostSpawnPositions" : "clientSpawnPositions")} ei ole asetettu!");
            return null;
        }

        var spawnedPlayersUnit = new GameObject[spawnPoints.Length];
        for (int i = 0; i < spawnPoints.Length; i++)
        {
            var playerUnit = Instantiate(unitPrefab, spawnPoints[i], Quaternion.identity);
            if (playerUnit.TryGetComponent<Unit>(out var u) && conn.identity != null)
                u.OwnerId = conn.identity.netId;
            spawnedPlayersUnit[i] = playerUnit;
        }

        return spawnedPlayersUnit;
    }

    public GameObject GetUnitPrefabForPlayer(bool isHost)
    {
        if (unitHostPrefab == null || unitClientPrefab == null)
        {
            Debug.LogError("Unit prefab references not set in SpawnUnitsCoordinator!");
            return null;
        }

        return isHost ? unitHostPrefab : unitClientPrefab;
    }

    public Vector3[] GetSpawnPositionsForPlayer(bool isHost)
    {
        if (hostSpawnPositions.Length == 0 || clientSpawnPositions.Length == 0)
        {
            Debug.LogError("Spawn position arrays not set in SpawnUnitsCoordinator!");
            return new Vector3[0];
        }

        return isHost ? hostSpawnPositions : clientSpawnPositions;
    }

    public GameObject[] SpawnEnemies()
    {
        var spawnedEnemies = new GameObject[enemySpawnPositions.Length];

        for (int i = 0; i < enemySpawnPositions.Length; i++)
        {
            var enemy = Instantiate(GetEnemyPrefab(), enemySpawnPositions[i], Quaternion.identity);
            spawnedEnemies[i] = enemy;
        }

        SetEnemiesSpawned(true);
        return spawnedEnemies;
    }
    public Vector3[] GetEnemySpawnPositions()
    {
        if (enemySpawnPositions.Length == 0)
        {
            Debug.LogError("Enemy spawn position array not set in SpawnUnitsCoordinator!");
            return new Vector3[0];
        }

        return enemySpawnPositions;
    }



    public void SetEnemiesSpawned(bool value)
    {
        enemiesSpawned = value;
    }
    public bool AreEnemiesSpawned()
    {
        return enemiesSpawned;
    }

    public GameObject GetEnemyPrefab()
    {
        if (enemyPrefab == null)
        {
            Debug.LogError("Enemy prefab reference not set in SpawnUnitsCoordinator!");
            return null;
        }
        return enemyPrefab;
    }

    public void SpwanSinglePlayerUnits()
    {
        SpawnPlayer1UnitsOffline();
        SpawnEnemyUnitsOffline();
    }

    // Singleplayer Gamemode Spawn units. hardcoded for now.
    // Later we can make it more generic with arrays and prefabs like in Co-op.
    private void SpawnPlayer1UnitsOffline()
    {
        Instantiate(unitHostPrefab, hostSpawnPositions[0], Quaternion.identity);
        Instantiate(unitHostPrefab, hostSpawnPositions[1], Quaternion.identity);
    }
    private void SpawnEnemyUnitsOffline()
    {
        Instantiate(enemyPrefab, enemySpawnPositions[0], Quaternion.identity);
        Instantiate(enemyPrefab, enemySpawnPositions[1], Quaternion.identity);
    }
}
```

Assets/scripts/MenuUI/BackButtonUI.cs

```
using UnityEngine;
using UnityEngine.UI;

public class BackButtonUI : MonoBehaviour
{

    // Serialized fields
    [Header("Canvas References")]
    [SerializeField] private GameObject connectCanvas; // this (self)
    [SerializeField] private GameObject gameModeSelectCanvas; // Hiden on start

    [Header("Buttons")]
    [SerializeField] private Button backButton;

    private void Awake()
    {

        // Add button listener
        backButton.onClick.AddListener(BackButton_OnClick);
    }

    private void BackButton_OnClick()
    {
        // Sign out the player from Unity Services
        Authentication authentication = connectCanvas.GetComponent<Authentication>();
        authentication.SignOutPlayerFromUnityServer();

        // Hide the connect canvas and show the game mode select canvas
        connectCanvas.SetActive(false);
        gameModeSelectCanvas.SetActive(true);
    }

}
```

Assets/scripts/MenuUI/GameModeSelectUI.cs

```
using UnityEngine;
using UnityEngine.UI;

public class GameModeSelectUI : MonoBehaviour
{
    // Serialized fields
    [Header("Canvas References")]
    [SerializeField] private GameObject gameModeSelectCanvas; // this (self)
    [SerializeField] private GameObject connectCanvas;        // Hiden on start

    // UI Elements
    [Header("Buttons")]
    [SerializeField] private Button coopButton;
    [SerializeField] private Button pvpButton;

    private void Awake()
    {
        // Ensure the game mode select canvas is active and connect canvas is inactive at start
        gameModeSelectCanvas.SetActive(true);
        connectCanvas.SetActive(false);

        // Add button listeners
        coopButton.onClick.AddListener(OnClickCoOp);
        pvpButton.onClick.AddListener(OnClickPvP);
    }

    public void OnClickCoOp()
    {
        GameModeManager.SetCoOp();
        OnSelected();
    }

    public void OnClickPvP()
    {
        GameModeManager.SetVersus();
        OnSelected();
    }

    public async void OnSelected()
    {
        Authentication authentication = connectCanvas.GetComponent<Authentication>();
        await authentication.SingInPlayerToUnityServerAsync();

        FieldCleaner.ClearAll();
        StartCoroutine(ResetGridNextFrame());
        gameModeSelectCanvas.SetActive(false);
        connectCanvas.SetActive(true);
    }

    private System.Collections.IEnumerator ResetGridNextFrame()
    {
        yield return new WaitForEndOfFrame();
        var lg = LevelGrid.Instance;
        if (lg != null) lg.RebuildOccupancyFromScene();
    }


    public void Reset()
    {
        // Pieni “siivous” ennen reloadia on ok, mutta ei pakollinen
        FieldCleaner.ClearAll();

        if (Mirror.NetworkServer.active)
        {
            ResetService.Instance.HardResetServerAuthoritative();
        }
        else if (Mirror.NetworkClient.active)
        {
            ResetService.Instance.CmdRequestHardReset();
        }
        else
        {
            // Yksinpeli
            GameReset.HardReloadSceneKeepMode();
        }
    }
}
```

Assets/scripts/Oneline/Authentication.cs

```
using System;
using System.Threading.Tasks;
using Unity.Services.Authentication;
using Unity.Services.Core;
using UnityEngine;

/// <summary>
/// This class is responsible for handling the authentication process.
/// It initializes the Unity Services and signs in the user anonymously.
/// Required when using Unity Relay, as it provides player authentication
/// and enables online multiplayer without port forwarding or direct IP connections.
/// </summary>
public class Authentication : MonoBehaviour
{
    public async Task SingInPlayerToUnityServerAsync()
    {
        try
        {
            await UnityServices.InitializeAsync();
            await AuthenticationService.Instance.SignInAnonymouslyAsync();
            Debug.Log("Logged into Unity, player ID: " + AuthenticationService.Instance.PlayerId);
        }
        catch (Exception e)
        {
            Debug.LogError(e);
        }
    }

    public void SignOutPlayerFromUnityServer()
    {
        if (AuthenticationService.Instance.IsSignedIn)
        {
            AuthenticationService.Instance.SignOut();
            Debug.Log("Player signed out of Unity Services");
        }
    }
}
```

Assets/scripts/Oneline/Connect.cs

```
using UnityEngine;
using TMPro;
using Mirror;
using Utp;
using UnityEngine.SceneManagement;

/// <summary>
/// This class is responsible for connecting to a game as a host or client.
///
/// NOTE: Button callbacks are set in the Unity Inspector.
/// </summary>
public class Connect : MonoBehaviour
{
    [SerializeField] private GameNetworkManager gameNetworkManager; // vedä tämä Inspectorissa
    [SerializeField] private TMP_InputField ipField;

    void Awake()
    {
        // find the NetworkManager in the scene if not set in Inspector
        if (!gameNetworkManager) gameNetworkManager = NetworkManager.singleton as GameNetworkManager;
        if (!gameNetworkManager) gameNetworkManager = FindFirstObjectByType<GameNetworkManager>();
        if (!gameNetworkManager) Debug.LogError("[Connect] GameNetworkManager not found in scene.");
    }


    public void HostLAN()
    {

        LoadSceneToAllHostLAN();
    }


    public void ClientLAN()
    {
        // Jos syötekenttä puuttuu/tyhjä → oletus localhost (sama kone)
        string ip = (ipField != null && !string.IsNullOrWhiteSpace(ipField.text))
                      ? ipField.text.Trim()
                      : "localhost"; // tai 127.0.0.1

        gameNetworkManager.networkAddress = ip;   // <<< TÄRKEIN KOHTA
        gameNetworkManager.JoinStandardServer();  // useRelay=false ja StartClient()
    }

    public void Host()
    {
        if (!gameNetworkManager)
        {
            Debug.LogError("[Connect] GameNetworkManager not found in scene.");
            return;
        }
        LoadSceneToAllHost();
    }

    public void Client()
    {
        if (!gameNetworkManager)
        {
            Debug.LogError("[Connect] GameNetworkManager not found in scene.");
            return;
        }

        gameNetworkManager.JoinRelayServer();
    }

    /// <summary>
    /// Starts a LAN host and loads the current scene for all clients.
    /// </summary>
    public void LoadSceneToAllHostLAN()
    {
        gameNetworkManager.StartStandardHost();
        var sceneName = SceneManager.GetActiveScene().name;
        NetworkManager.singleton.ServerChangeScene(sceneName);
    }

    /// <summary>
    /// Starts a relay host and loads the current scene for all clients.
    /// </summary>
    public void LoadSceneToAllHost()
    {
        gameNetworkManager.StartRelayHost(2, null);
        var sceneName = SceneManager.GetActiveScene().name;
        NetworkManager.singleton.ServerChangeScene(sceneName);
    }
}
```

Assets/scripts/Oneline/CoopTurnCoordinator.cs

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

public class CoopTurnCoordinator : NetworkBehaviour
{
    public static CoopTurnCoordinator Instance { get; private set; }

    void Awake()
    {
        if (Instance != null && Instance != this) { Destroy(gameObject); return; }
        Instance = this;
    }


    [Server]
    public void TryAdvanceIfReady()
    {
        if (NetTurnManager.Instance.phase == TurnPhase.Players && NetTurnManager.Instance.endedPlayers.Count >= Mathf.Max(1, NetTurnManager.Instance.requiredCount))
        {
            StartCoroutine(ServerEnemyTurnThenNextPlayers());
        }
    }

    [Server]
    private IEnumerator ServerEnemyTurnThenNextPlayers()
    {
        // Asettaa vihollisen WordUI: (Action Points) näkyviin.
        UnitUIBroadcaster.Instance.BroadcastUnitWorldUIVisibility(true);

        // 1) Vihollisvuoro alkaa
        RpcTurnPhaseChanged(NetTurnManager.Instance.phase = TurnPhase.Enemy, NetTurnManager.Instance.turnNumber, false);

        // Silta unit/AP-logiikalle (sama kuin nyt)
        if (TurnSystem.Instance != null)
        {
            TurnSystem.Instance.ForcePhase(isPlayerTurn: false, incrementTurnNumber: false);
        }

        // Aja AI
        yield return RunEnemyAI();

        // 2) Paluu pelaajille + turn-numero + resetit
        NetTurnManager.Instance.turnNumber++;
        NetTurnManager.Instance.ResetTurnState();

        if (TurnSystem.Instance != null)
        {
            TurnSystem.Instance.ForcePhase(isPlayerTurn: true, incrementTurnNumber: false);
        }

        // 3) Lähetä *kaikille* (host + clientit) HUD-päivitys SP-logiikan kautta
        RpcTurnPhaseChanged(NetTurnManager.Instance.phase = TurnPhase.Players, NetTurnManager.Instance.turnNumber, true);

        // Asettaa pelaajien WordUI: (Action Points) näkyviin.
        UnitUIBroadcaster.Instance.BroadcastUnitWorldUIVisibility(false);
    }

    [Server]
    IEnumerator RunEnemyAI()
    {
        if (EnemyAI.Instance != null)
            yield return EnemyAI.Instance.RunEnemyTurnCoroutine();
        else
            yield return null; // fallback, ettei ketju katkea
    }

    // ---- Client-notifikaatiot UI:lle ----
    [ClientRpc]
    public void RpcTurnPhaseChanged(TurnPhase newPhase, int newTurnNumber, bool isPlayersPhase)
    {
        // Päivitä paikallinen SP-UI-luuppi (ei Mirror-kutsuja)
        if (TurnSystem.Instance != null)
            TurnSystem.Instance.SetHudFromNetwork(newTurnNumber, isPlayersPhase);

        // Vaihe vaihtui → varmuuden vuoksi piilota mahdollinen "READY" -teksti
        var ui = FindFirstObjectByType<TurnSystemUI>();
        if (ui != null) ui.SetTeammateReady(false, null);
    }


    // Näyttää toiselle pelaajalle "Player X READY"
    [ClientRpc]
    public void RpcUpdateReadyStatus(int[] whoEndedIds, string[] whoEndedLabels)
    {
        var ui = FindFirstObjectByType<TurnSystemUI>();
        if (ui == null) return;

        // Selvitä oma netId
        uint localId = 0;
        if (NetworkClient.connection != null && NetworkClient.connection.identity)
            localId = NetworkClient.connection.identity.netId;

        bool show = false;
        string label = null;

        // Jos joku muu kuin minä on valmis → näytä hänen labelinsa
        for (int i = 0; i < whoEndedIds.Length; i++)
        {
            if ((uint)whoEndedIds[i] != localId)
            {
                show = true;
                label = (i < whoEndedLabels.Length) ? whoEndedLabels[i] : "Teammate";
                break;
            }
        }

        ui.SetTeammateReady(show, label);
    }

    // ---- Server-apurit ----
    [Server] string GetLabelByNetId(uint id)
    {
        foreach (var kvp in NetworkServer.connections)
        {
            var conn = kvp.Value;
            if (conn != null && conn.identity && conn.identity.netId == id)
                return conn.connectionId == 0 ? "Player 1" : "Player 2";
        }
        return "Teammate";
    }

    [Server]
    public string[] BuildEndedLabels()
    {
        // HashSetin järjestys ei ole merkityksellinen, näytetään mikä tahansa toinen
        return NetTurnManager.Instance.endedPlayers.Select(id => GetLabelByNetId(id)).ToArray();
    }
}
```

Assets/scripts/Oneline/GameNetworkManager.cs

```
using System;
using System.Collections.Generic;
using Mirror;
using UnityEngine;
using Unity.Services.Relay.Models;

namespace Utp
{
 [RequireComponent(typeof(UtpTransport))]
 public class GameNetworkManager : NetworkManager
 {
  public static GameNetworkManager Instance { get; private set; }
  private UtpTransport utpTransport;

  /// <summary>
  /// Server's join code if using Relay.
  /// </summary>
  public string relayJoinCode = "";


  public override void Awake()
  {
   if (Instance != null && Instance != this)
   {
    Destroy(gameObject);
    return;
   }
   Instance = this;

   base.Awake();
   autoCreatePlayer = false;

   utpTransport = GetComponent<UtpTransport>();

   string[] args = Environment.GetCommandLineArgs();
   for (int key = 0; key < args.Length; key++)
   {
    if (args[key] == "-port")
    {
     if (key + 1 < args.Length)
     {
      string value = args[key + 1];

      try
      {
       utpTransport.Port = ushort.Parse(value);
      }
      catch
      {
       UtpLog.Warning($"Unable to parse {value} into transport Port");
      }
     }
    }
   }
  }

  public override void OnStartServer()
  {
   base.OnStartServer();
   SpawnUnitsCoordinator.Instance.SetEnemiesSpawned(false);

   if (GameModeManager.SelectedMode == GameMode.CoOp)
   {
    ServerSpawnEnemies();
   }

  }

  /// <summary>
  /// Get the port the server is listening on.
  /// </summary>
  /// <returns>The port.</returns>
  public ushort GetPort()
  {
   return utpTransport.Port;
  }

  /// <summary>
  /// Get whether Relay is enabled or not.
  /// </summary>
  /// <returns>True if enabled, false otherwise.</returns>
  public bool IsRelayEnabled()
  {
   return utpTransport.useRelay;
  }

  /// <summary>
  /// Ensures Relay is disabled. Starts the server, listening for incoming connections.
  /// </summary>
  public void StartStandardServer()
  {
   utpTransport.useRelay = false;
   StartServer();
  }

  /// <summary>
  /// Ensures Relay is disabled. Starts a network "host" - a server and client in the same application
  /// </summary>
  public void StartStandardHost()
  {
   utpTransport.useRelay = false;
   StartHost();
```

```
}

/// <summary>
/// Gets available Relay regions.
/// </summary>
///
public void GetRelayRegions(Action<List<Region>> onSuccess, Action onFailure)
{
 utpTransport.GetRelayRegions(onSuccess, onFailure);
}

/// <summary>
/// Ensures Relay is enabled. Starts a network "host" - a server and client in the same application
/// </summary>
public void StartRelayHost(int maxPlayers, string regionId = null)
{
 utpTransport.useRelay = true;
 utpTransport.AllocateRelayServer(maxPlayers, regionId,
 (string joinCode) =>
 {
  relayJoinCode = joinCode;
  Debug.LogError($"Relay join code: {joinCode}");
  StartHost();
 },
 () =>
 {
  UtpLog.Error($"Failed to start a Relay host.");
 });
}

/// <summary>
/// Ensures Relay is disabled. Starts the client, connects it to the server with networkAddress.
/// </summary>
public void JoinStandardServer()
{
 utpTransport.useRelay = false;
 StartClient();
}

/// <summary>
/// Ensures Relay is enabled. Starts the client, connects to the server with the relayJoinCode.
/// </summary>
public void JoinRelayServer()
{
 utpTransport.useRelay = true;
 utpTransport.ConfigureClientWithJoinCode(relayJoinCode,
 () =>
 {
  StartClient();
 },
 () =>
 {
  UtpLog.Error($"Failed to join Relay server.");
 });
}

public override void OnValidate()
{
 base.OnValidate();
}

bool addPlayerRequested;

/// <summary>
/// Make sure that the clien sends a AddPlayer request once the scene is loaded.
/// </summary>
public override void OnClientSceneChanged()
{
 base.OnClientSceneChanged();

 if (!NetworkClient.ready) NetworkClient.Ready();

 // Send AddPlayer message only once
 if (NetworkClient.connection != null &&
  NetworkClient.connection.identity == null &&
  !addPlayerRequested)
 {
  addPlayerRequested = true;
  NetworkClient.AddPlayer();
 }
}

public override void OnStopClient()
{
 base.OnStopClient();
 addPlayerRequested = false; // nollaa vartija disconnectissa
}

public override void OnClientDisconnect()
{
 base.OnClientDisconnect();
 addPlayerRequested = false;
}


/// <summary>
/// Tämä metodi spawnaa jokaiselle clientille oman Unitin ja tekee siitä heidän ohjattavan yksikkönsä.
/// </summary>
public override void OnServerAddPlayer(NetworkConnectionToClient conn)
{

 if (playerPrefab == null)
 {
  Debug.LogError("[NM] Player Prefab (EmptySquad) puuttuu!");
  return;
 }
 base.OnServerAddPlayer(conn);

 // 2) päätä host vs client
 bool isHost = conn.connectionId == 0;

 // 3) spawnaa pelaajan yksiköt ja anna authority niihin
 var units = SpawnUnitsCoordinator.Instance.SpawnPlayersForNetwork(conn, isHost);
 foreach (var unit in units)
 {
  Debug.Log($"[NM] Spawning player unit {unit.name} for connection {conn.connectionId}, isHost={isHost}");
  NetworkServer.Spawn(unit, conn); // authority tälle pelaajalle
 }

 // päivitä pelaajamäärä koordinaattorille
 var coord = NetTurnManager.Instance;
 //var coord = CoopTurnCoordinator.Instance;
 if (coord != null)
  coord.ServerUpdateRequiredCount(NetworkServer.connections.Count);

 //  Jos nyt on Players-vuoro, avaa toiminta tälle uudelle clientille
 if (NetTurnManager.Instance && NetTurnManager.Instance.phase == TurnPhase.Players)
 {
  var pc = conn.identity ? conn.identity.GetComponent<PlayerController>() : null;
  if (pc != null) pc.ServerSetHasEnded(false);   // -> TargetRpc avaa UI:n
 }

 // Asettaa pelaajan UI.n pelaajan vuoroksi.
 if (CoopTurnCoordinator.Instance && NetTurnManager.Instance)
 {
  CoopTurnCoordinator.Instance.RpcTurnPhaseChanged(
   NetTurnManager.Instance.phase,
   NetTurnManager.Instance.turnNumber,
   true
  );
 }

 // --- VERSUS (PvP) – host aloittaa ---
 if (GameModeManager.SelectedMode == GameMode.Versus)
 {
  var pc = conn.identity != null ? conn.identity.GetComponent<PlayerController>() : null;
  if (pc != null && PvPTurnCoordinator.Instance != null)
  {
   // Rekisteröi pelaaja PvP-vuoroon (host saa aloitusvuoron PvPTurnCoordinatorissa)
   PvPTurnCoordinator.Instance.ServerRegisterPlayer(pc);
  }
  else
  {
   Debug.LogWarning("[NM] PvP rekisteröinti epäonnistui: PlayerController tai PvPTurnCoordinator puuttuu.");
  }
 }
```

```
 }

 [Server]
 public void ServerSpawnEnemies()
 {
  // Pyydä SpawnUnitsCoordinatoria luomaan viholliset
  var enemies = SpawnUnitsCoordinator.Instance.SpawnEnemies();

  // Synkronoi viholliset verkkoon Mirrorin avulla
  foreach (var enemy in enemies)
  {
   if (enemy != null)
   {
    NetworkServer.Spawn(enemy);
   }
  }
 }


 public override void OnServerDisconnect(NetworkConnectionToClient conn)
 {
  base.OnServerDisconnect(conn);
  // päivitä pelaajamäärä koordinaattorille
  var coord = NetTurnManager.Instance;
  //var coord = CoopTurnCoordinator.Instance;
  if (coord != null)
   coord.ServerUpdateRequiredCount(NetworkServer.connections.Count);
 }

 public bool IsNetworkActive()
 {
  return GetNetWorkServerActive() || GetNetWorkClientConnected();
 }

 public bool GetNetWorkServerActive()
 {
  return NetworkServer.active;
 }

 public bool GetNetWorkClientConnected()
 {
  return NetworkClient.isConnected;
 }

 public NetworkConnection NetWorkClientConnection()
 {
  return NetworkClient.connection;
 }

 public void NetworkDestroy(GameObject go)
 {
  NetworkServer.Destroy(go);
  }

  public void SetEnemies()
  {
   SpawnUnitsCoordinator.Instance.SetEnemiesSpawned(false);

   if (GameModeManager.SelectedMode == GameMode.CoOp)
   {
    ServerSpawnEnemies();
   }
  }
 }
}
```

Assets/scripts/Oneline/NetSceneReload.cs

```
using Mirror;
using UnityEngine.SceneManagement;

public static class NetSceneReload {
    public static void ReloadForAll()
    {
        string sceneName = SceneManager.GetActiveScene().name;
        NetworkManager.singleton.ServerChangeScene(sceneName);
    }
}
```

Assets/scripts/Oneline/NetTurnManager.cs

```
using UnityEngine;
using Mirror;
using System.Collections.Generic;
using System.Collections;
using System.Linq;
///<sumary>
/// NetTurnManager coordinates turn phases in a networked multiplayer game.
/// It tracks which players have ended their turns and advances the game phase accordingly.
///</sumary>
public enum TurnPhase { Players, Enemy }
public class NetTurnManager : NetworkBehaviour
{
    public static NetTurnManager Instance { get; private set; }
    [SyncVar] public TurnPhase phase = TurnPhase.Players;
    [SyncVar] public int turnNumber = 1;

    // Seurannat (server)
    [SyncVar] public int endedCount = 0;
    [SyncVar] public int requiredCount = 0; // päivitetään kun pelaajia liittyy/lähtee

    public readonly HashSet<uint> endedPlayers = new();

    void Awake()
    {
        if (Instance != null && Instance != this) { Destroy(gameObject); return; }
        Instance = this;
    }

    public override void OnStartServer()
    {
        base.OnStartServer();
        // jos haluat lukita kahteen pelaajaan protoa varten:
        if (GameModeManager.SelectedMode == GameMode.CoOp) requiredCount = 2;
        StartCoroutine(DeferResetOneFrame());
    }

    [Server]
    private IEnumerator DeferResetOneFrame()
    {
        yield return null;                  // odota että SpawnObjects on valmis
        ResetTurnState();                   // nyt RpcUpdateReadyStatus on turvallinen
    }

    [Server]
    public void ResetTurnState()
    {

        phase = TurnPhase.Players;
        endedPlayers.Clear();
        endedCount = 0;
        SetPlayerStartState();
    }

    [Server]
    public void ServerPlayerEndedTurn(uint playerNetId)
    {
        // PvP: siirrä vuoro heti vastustajalle
        if (GameModeManager.SelectedMode == GameMode.Versus)
        {
            if (PvPTurnCoordinator.Instance)
                PvPTurnCoordinator.Instance.ServerHandlePlayerEndedTurn(playerNetId);
            return;
        }

        if (phase != TurnPhase.Players) return;          // ei lasketa jos ei pelaajavuoro
        if (!endedPlayers.Add(playerNetId)) return;      // älä laske tuplia

        endedCount = endedPlayers.Count;

        // Ilmoita kaikille, KUKA on valmis → UI näyttää "Player X READY" toisella pelaajalla. Käytössä vain Co-opissa
        if (GameModeManager.SelectedMode == GameMode.CoOp)
        {
            // Asettaa yksikoiden UI Näkyvyydet
            UnitUIBroadcaster.Instance.BroadcastUnitWorldUIVisibility(false);

            CoopTurnCoordinator.Instance.
            RpcUpdateReadyStatus(
            endedPlayers.Select(id => (int)id).ToArray(),
            CoopTurnCoordinator.Instance.BuildEndedLabels()
            );

            CoopTurnCoordinator.Instance.TryAdvanceIfReady();
        }
    }

    [Server]
    public void ServerUpdateRequiredCount(int playersNow)
    {
        requiredCount = Mathf.Max(1, playersNow); // Co-opissa yleensä 2
                                              // jos yksi poistui kesken odotuksen, tarkista täyttyikö ehto nyt

        if (GameModeManager.SelectedMode == GameMode.CoOp)
        {
            CoopTurnCoordinator.Instance.TryAdvanceIfReady();
        }
    }

    public void SetPlayerStartState()
    {
        // Asettaa pelaajan tilan pelaajan vuoroksi.
        foreach (var kvp in NetworkServer.connections)
        {
            var id = kvp.Value.identity;
            if (!id) continue;
            var pc = id.GetComponent<PlayerController>();
            if (pc) pc.ServerSetHasEnded(false);  // <<< TÄRKEIN RIVI
        }
    }
}
```

Assets/scripts/Oneline/NetVisibility.cs

```
using Mirror;
using UnityEngine;

public class NetVisibility : NetworkBehaviour
{
    [SerializeField] private GameObject target; // se esine jonka näkyvyyttä halutaan ohjata

    [SyncVar(hook = nameof(OnChanged))]
    private bool isVisible;

    void OnChanged(bool _, bool now) => Apply(now);

    public override void OnStartClient() => Apply(isVisible);

    private void Apply(bool now)
    {
        if (target) target.SetActive(now);
    }

    // --- SERVER-API ---
    [Server] public void ServerShow()            { isVisible = true;  Apply(true);  }
    [Server] public void ServerHide()            { isVisible = false; Apply(false); }
    [Server] public void ServerSetVisible(bool v){ isVisible = v;     Apply(v);     }

    // --- CLIENT-API (authority) ---
    [Command] private void CmdSetVisible(bool v) => ServerSetVisible(v);

    /// Kutsu tätä mistä tahansa: hoitaa sekä server- että client-puolen.
    public void SetVisibleAny(bool v)
    {
        if (isServer) ServerSetVisible(v);
        else          CmdSetVisible(v);  // vaatii client authorityn tälle objektille
    }
}
```

Assets/scripts/Oneline/PvpClientState.cs

```
using UnityEngine;
using System;
public class PvpClientState : MonoBehaviour
{
    public static bool IsMyTurn { get; set; }
}

public static class PvpClientEvents
{
    public static event Action<uint, int> OnTurnChanged;

    public static void RaiseTurnChanged(uint turnOwnerNetId, int turnNo)
        => OnTurnChanged?.Invoke(turnOwnerNetId, turnNo);
}
```

Assets/scripts/Oneline/PvpPerception.cs

```
using System.Reflection;
using Mirror;
using UnityEngine;

public class PvpPerception : MonoBehaviour
{
    // Kutsu tätä aina kun vuoro vaihtuu (ja bootstrapissa)
    public static void ApplyEnemyFlagsLocally(bool isMyTurn)
    {
        var units = FindObjectsByType<Unit>(FindObjectsSortMode.None);

        foreach (var u in units)
        {
            var ni = u.GetComponent<NetworkIdentity>();
            if (!ni) continue;

            // Onko tämä yksikkö minun (tässä clientissä)?
            bool unitIsMine = ni.isOwned || ni.isLocalPlayer;

            // Vuorologiikka:
            // - Jos on MINUN vuoro: vastustajan yksiköt ovat enemy
            // - Jos EI ole minun vuoro: MINUN omat yksiköt ovat enemy
            bool enemy = isMyTurn ? !unitIsMine : unitIsMine;

            SetUnitEnemyFlag(u, enemy);
        }
    }

    static void SetUnitEnemyFlag(Unit u, bool enemy)
    {
        // Unitissa on [SerializeField] private bool isEnemy; -> käytä BindingFlagsia! :contentReference[oaicite:1]{index=1}
        var field = typeof(Unit).GetField("isEnemy",
            BindingFlags.Instance | BindingFlags.NonPublic | BindingFlags.Public);
        if (field != null) { field.SetValue(u, enemy); return; }

        // Varalle, jos joskus lisäät setterin
        var m = typeof(Unit).GetMethod("SetEnemy",
            BindingFlags.Instance | BindingFlags.Public | BindingFlags.NonPublic,
            null, new[] { typeof(bool) }, null);
        if (m != null) { m.Invoke(u, new object[] { enemy }); return; }

        Debug.LogWarning("[PvP] Unitilta puuttuu isEnemy/SetEnemy(bool). Lisää jompikumpi.");
    }
}
```

Assets/scripts/Oneline/PvPTurnCoordinator.cs

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

public class PvPTurnCoordinator : NetworkBehaviour
{
    public static PvPTurnCoordinator Instance { get; private set; }

    [SyncVar] private uint currentOwnerNetId; // kumman pelaajan vuoro on

    void Awake()
    {
        if (Instance != null && Instance != this) { Destroy(gameObject); return; }
        Instance = this;
    }

    // Kutsutaan, kun pelaaja liittyy. Hostista tehdään aloitusvuoron omistaja.
    [Server]
    public void ServerRegisterPlayer(PlayerController pc)
    {
        // Host (connectionId == 0) asettaa aloitusvuoron, jos ei vielä asetettu
        if (currentOwnerNetId == 0 && pc.connectionToClient != null && pc.connectionToClient.connectionId == 0)
        {
            currentOwnerNetId = pc.netId;
            pc.ServerSetHasEnded(false);     // host saa toimia
            foreach (var other in GetAllPlayers().Where(p => p != pc))
                other.ServerSetHasEnded(true); // muut lukkoon varmuudeksi

            RpcTurnChanged(GetTurnNumber(), currentOwnerNetId);
        }
        else
        {
            // Myöhemmin liittynyt (client) – lukitaan kunnes hänen vuoronsa alkaa
            pc.ServerSetHasEnded(true);
            RpcTurnChanged(GetTurnNumber(), currentOwnerNetId);
        }
    }

    // Kutsutaan, kun joku painaa End Turn
    [Server]
    public void ServerHandlePlayerEndedTurn(uint whoEndedNetId)
    {
        var players = GetAllPlayers().ToList();
        var ended = players.FirstOrDefault(p => p.netId == whoEndedNetId);
        var next = players.FirstOrDefault(p => p.netId != whoEndedNetId);
        if (next == null) return; // ei vastustajaa vielä

        // Nosta vuorolaskuria (kierrätetään olemassaolevaa turnNumberia)
        if (NetTurnManager.Instance) NetTurnManager.Instance.turnNumber++;
        currentOwnerNetId = next.netId;

        // Anna seuraavalle vuoro
        next.ServerSetHasEnded(false);   // avaa syötteen ja nappulan
        // ended pysyy lukossa (hasEndedThisTurn = true)
        RpcTurnChanged(GetTurnNumber(), currentOwnerNetId);
    }

    int GetTurnNumber() => NetTurnManager.Instance ? NetTurnManager.Instance.turnNumber : 1;

    [ClientRpc]
    void RpcTurnChanged(int newTurnNumber, uint ownerNetId)
    {
        // Päivitä paikallinen HUD “player/enemy turn” -logiikalla
        bool isMyTurn = false;
        if (NetworkClient.connection != null && NetworkClient.connection.identity != null)
            isMyTurn = NetworkClient.connection.identity.netId == ownerNetId;

        PvpPerception.ApplyEnemyFlagsLocally(isMyTurn);

        if (TurnSystem.Instance != null)
            TurnSystem.Instance.SetHudFromNetwork(newTurnNumber, isMyTurn);

    }

    [Server]
    IEnumerable<PlayerController> GetAllPlayers()
    {
        foreach (var kvp in NetworkServer.connections)
        {
            var id = kvp.Value.identity;
            if (!id) continue;
            var pc = id.GetComponent<PlayerController>();
            if (pc) yield return pc;
        }
    }
}
```

Assets/scripts/Oneline/ResetService.cs

```
using System.Collections;
using Mirror;
using UnityEngine.SceneManagement;

public class ResetService : NetworkBehaviour
{
    public static ResetService Instance;

    // LIPPU: ajetaan post-reset -alustus, kun uusi scene on valmis
    public static bool PendingHardReset;

    void Awake() => Instance = this;

    [Command(requiresAuthority = false)]
    public void CmdRequestHardReset()
    {
        if (!NetworkServer.active) return;
        HardResetServerAuthoritative();
    }

    [Server]
    public void HardResetServerAuthoritative()
    {
        PendingHardReset = true; // <-- vain lippu päälle
        var nm = (NetworkManager)NetworkManager.singleton;
        var scene = SceneManager.GetActiveScene().name;
        nm.ServerChangeScene(scene);
        // ÄLÄ tee mitään tähän enää
    }

    [ClientRpc]
    public void RpcPostResetClientInit(int turnNumber)
    {
        // odota 1 frame että UI-komponentit ovat ehtineet OnEnable/subscribe
        StartCoroutine(_ClientInitCo(turnNumber));
    }

    private IEnumerator _ClientInitCo(int turnNumber)
    {
        yield return null;

        // 1) Avaa paikallinen “saa toimia” -portti (triggaa LocalPlayerTurnChanged)
        PlayerLocalTurnGate.SetCanAct(true);

        // 2) Päivitä HUD (näyttää "Players turn", aktivoi End Turn -napin logiikkaasi vasten)
        TurnSystem.Instance?.SetHudFromNetwork(turnNumber, true);
    }
}
```

Assets/scripts/Oneline/ServerBootstrap.cs

```
using System.Collections;
using Mirror;
using UnityEngine;
using Utp;
/// <summary>
/// This ensures that the server starts correctly and in the correct order.
/// </summary>

[DefaultExecutionOrder(10000)]               // aja myöhään
[DisallowMultipleComponent]
public class ServerBootstrap : NetworkBehaviour
{
    public override void OnStartServer()
    {
        // varmistaa että tämä ei ajaudu clientillä
        StartCoroutine(Bootstrap());
    }

    private IEnumerator Bootstrap()
    {
        // 1) Odota että Mirror on spawnannut scene-identiteetit
        //    (2 frameä riittää, mutta odotetaan lisäksi koordinaattorit)
        yield return null;
        yield return null;

        // Odota kunnes koordinaattori(t) ovat varmasti olemassa ja spawned
        yield return new WaitUntil(() =>
            CoopTurnCoordinator.Instance &&
            CoopTurnCoordinator.Instance.netIdentity &&
            CoopTurnCoordinator.Instance.netIdentity.netId != 0
        );

        // 2) Nollaa vuorologiikka vain serverillä
        NetTurnManager.Instance.ResetTurnState();   // EI UI-RPC:itä täällä

        // 3) Spawnaa viholliset vain Co-opissa ja vain jos tarvitaan
        if (GameModeManager.SelectedMode == GameMode.CoOp &&
            !SpawnUnitsCoordinator.Instance.AreEnemiesSpawned())
        {
            GameNetworkManager.Instance.SetEnemies();
        }

        // 4) Rakenna occupancy nykyisestä scenestä (unitit/esteet)
        LevelGrid.Instance?.RebuildOccupancyFromScene();

        // 5) Pakota aloitus Players turniin ja turnNumber = 1
        NetTurnManager.Instance.turnNumber = 1;
        NetTurnManager.Instance.phase = TurnPhase.Players;
        TurnSystem.Instance?.ForcePhase(isPlayerTurn: true, incrementTurnNumber: false);
        // 6) Nyt on turvallista lähettää UI/RPC:t kaikille
        var endedIds = System.Array.Empty<int>();
        var endedLabels = CoopTurnCoordinator.Instance.BuildEndedLabels();

        CoopTurnCoordinator.Instance.RpcUpdateReadyStatus(endedIds, endedLabels);
        CoopTurnCoordinator.Instance.RpcTurnPhaseChanged(
            NetTurnManager.Instance.phase,
            NetTurnManager.Instance.turnNumber,
            true // isPlayersPhase
        );

        // (valinnainen) piilota enemy-WorldUI tms. alussa
        UnitUIBroadcaster.Instance?.BroadcastUnitWorldUIVisibility(false);

        // (valinnainen) client-init, jos sinulla on tällainen
        ResetService.Instance?.RpcPostResetClientInit(NetTurnManager.Instance.turnNumber);

        NetTurnManager.Instance.SetPlayerStartState();
    }
}
```

Assets/scripts/Oneline/Sync/NetworkSync.cs

```
using Mirror;
using Mirror.Examples.CharacterSelection;
using UnityEngine;

/// <summary>
/// NetworkSync is a static helper class that centralizes all network-related actions.
///
/// Responsibilities:
/// - Provides a single entry point for spawning and synchronizing networked effects and objects.
/// - Decides whether the game is running in server/host mode, client mode, or offline mode.
/// - In online play:
///     - If running on the server/host, spawns objects directly with NetworkServer.Spawn.
///     - If running on a client, forwards the request to the local NetworkSyncAgent, which relays it to the server via Command.
/// - In offline/singleplayer mode, simply instantiates objects locally with Instantiate.
///
/// Usage:
/// Call the static methods from gameplay code (e.g. UnitAnimator, Actions) instead of
/// directly instantiating or spawning prefabs. This ensures consistent behavior in all game modes.
///
/// Example:
/// NetworkSync.SpawnBullet(bulletPrefab, shootPoint.position, targetPosition);
/// </summary>
public static class NetworkSync
{
    /// <summary>
    /// Spawns a bullet projectile in the game world.
    /// Handles both offline (local Instantiate) and online (NetworkServer.Spawn) scenarios.
    ///
    /// In server/host:
    ///     - Instantiates and spawns the bullet directly with NetworkServer.Spawn.
    /// In client:
    ///     - Forwards the request to NetworkSyncAgent.Local, which executes a Command.
    /// In offline:
    ///     - Instantiates the bullet locally.
    /// </summary>
    /// <param name="bulletPrefab">The bullet prefab to spawn (must have NetworkIdentity if used online).</param>
    /// <param name="spawnPos">The starting position of the bullet (usually weapon muzzle).</param>
    /// <param name="targetPos">The target world position the bullet should travel towards.</param>
    public static void SpawnBullet(GameObject bulletPrefab, Vector3 spawnPos, Vector3 targetPos)
    {
        if (NetworkServer.active) // Online: server or host
        {
            var bullet = Object.Instantiate(bulletPrefab, spawnPos, Quaternion.identity);
            if (bullet.TryGetComponent<BulletProjectile>(out var bulletProjectile))
                bulletProjectile.Setup(targetPos);
            NetworkServer.Spawn(bullet);
            return;
        }
        if (NetworkClient.active) // Online: client
        {
            if (NetworkSyncAgent.Local != null)
            {
                NetworkSyncAgent.Local.CmdSpawnBullet(spawnPos, targetPos);
            }
            else
            {
                // fallback if no local agent found (shouldn't happen in a correct setup)
                Debug.LogWarning("[NetworkSync] No Local NetworkSyncAgent found, falling back to local Instantiate.");
                var bullet = Object.Instantiate(bulletPrefab, spawnPos, Quaternion.identity);
                if (bullet.TryGetComponent<BulletProjectile>(out var bulletProjectile))
                    bulletProjectile.Setup(targetPos);
            }
        }
        else
        {
            // Offline / Singleplayer: just instantiate locally
            var bullet = Object.Instantiate(bulletPrefab, spawnPos, Quaternion.identity);
            if (bullet.TryGetComponent<BulletProjectile>(out var bulletProjectile))
                bulletProjectile.Setup(targetPos);
        }
    }

    // HUOM: käytä tätä myös AE:stä (UnitAnimatorista)
    public static void SpawnGrenade(GameObject grenadePrefab, Vector3 spawnPos, Vector3 targetPos)
    {
        if (NetworkServer.active) // Online: server tai host
        {
            var go = Object.Instantiate(grenadePrefab, spawnPos, Quaternion.identity);
            if (go.TryGetComponent<GrenadeProjectile>(out var gp))
                gp.Setup(targetPos);                 // ASETUS ENNEN spawnia
            NetworkServer.Spawn(go);
            return;
        }

        if (NetworkClient.active) // Online: client
        {
            if (NetworkSyncAgent.Local != null)
            {
                NetworkSyncAgent.Local.CmdSpawnGrenade(spawnPos, targetPos);
            }
            else
            {
                // Sama fallback kuin luodeissa (jos näin haluat)
                Debug.LogWarning("[NetworkSync] No Local NetworkSyncAgent found, falling back to local Instantiate.");
                var go = Object.Instantiate(grenadePrefab, spawnPos, Quaternion.identity);
                if (go.TryGetComponent<GrenadeProjectile>(out var gp))
                    gp.Setup(targetPos);
            }
        }
        else
        {
            // Offline / Singleplayer
            var go = Object.Instantiate(grenadePrefab, spawnPos, Quaternion.identity);
            if (go.TryGetComponent<GrenadeProjectile>(out var gp))
                gp.Setup(targetPos);
        }
    }


    /// <summary>
    /// Apply damage to a Unit in SP/Host/Client modes.
    /// - Server/Host: call HealthSystem.Damage directly (authoritative).
    /// - Client: send a Command via NetworkSyncAgent to run on server.
    /// - Offline: call locally.
    /// </summary>
    public static void ApplyDamageToUnit(Unit target, int amount, Vector3 hitPosition)
    {
        if (target == null) return;

        if (NetworkServer.active) // Online: server or host
        {
            var healthSystem = target.GetComponent<HealthSystem>();
            if (healthSystem == null) return;

            healthSystem.Damage(amount, hitPosition);
            UpdateHealthBarUI(healthSystem, target);
            return;
        }

        if (NetworkClient.active) // Online: client
        {
            var ni = target.GetComponent<NetworkIdentity>();
            if (ni && NetworkSyncAgent.Local != null)
            {
                NetworkSyncAgent.Local.CmdApplyDamage(ni.netId, amount, hitPosition);
                return;
            }
        }

        // Offline fallback
        target.GetComponent<HealthSystem>()?.Damage(amount, hitPosition);
    }

    public static void ApplyDamageToObject(DestructibleObject target, int amount, Vector3 hitPosition)
    {
        if (target == null) return;

        if (NetworkServer.active) // Online: server or host
        {
            target.Damage(amount, hitPosition);
            return;
        }
        if (NetworkClient.active) // Online: client
        {
            var ni = target.GetComponent<NetworkIdentity>();
            if (ni && NetworkSyncAgent.Local != null)
            {
                NetworkSyncAgent.Local.CmdApplyDamageToObject(ni.netId, amount, hitPosition);
                return;
            }
        }

        // Offline fallback
        target.Damage(amount, hitPosition);
    }

    private static void UpdateHealthBarUI(HealthSystem healthSystem, Unit target)
    {
        // → ilmoita kaikille clienteille, jotta UnitWorldUI saa eventin
        if (NetworkSyncAgent.Local == null)
        {
            // haetaan mikä tahansa agentti serveriltä (voi olla erillinen manageri)
            var agent = Object.FindFirstObjectByType<NetworkSyncAgent>();
            if (agent != null)
                agent.ServerBroadcastHp(target, healthSystem.GetHealth(), healthSystem.GetHealthMax());
        }
        else
        {
            NetworkSyncAgent.Local.ServerBroadcastHp(target, healthSystem.GetHealth(), healthSystem.GetHealthMax());
        }
    }

    /// <summary>
    /// Server: Control when Pleyers can see own and others Unit stats,
    /// Like only active player AP(Action Points) are visible.
    /// When is Enemy turn only Enemy Units Action points are visible.
    /// Solo and Versus mode handle this localy becouse there is no need syncronisation.
    /// </summary>
    public static void BroadcastActionPoints(Unit unit, int apValue)
    {
        if (unit == null) return;

        if (NetworkServer.active)
        {
            var agent = Object.FindFirstObjectByType<NetworkSyncAgent>();
            if (agent != null)
                agent.ServerBroadcastAp(unit, apValue);
            return;
        }

        // CLIENT-haara: lähetä peilauspyyntö serverille
        if (NetworkClient.active && NetworkSyncAgent.Local != null)
        {
            var ni = unit.GetComponent<NetworkIdentity>();
            if (ni) NetworkSyncAgent.Local.CmdMirrorAp(ni.netId, apValue);
        }
    }

    public static void SpawnRagdoll(GameObject prefab, Vector3 pos, Quaternion rot, uint sourceUnitNetId, Transform originalRootBone, Vector3 lastHitPosition, int overkill)
    {

        if (NetworkServer.active)
        {
            var go = Object.Instantiate(prefab, pos, rot);

            if (go.TryGetComponent<UnitRagdoll>(out var rg))
            {
                rg.SetOverkill(overkill);
                rg.SetLastHitPosition(lastHitPosition);
            }

            // Set sourceUnitNetId so that clients can find the original unit
            if (go.TryGetComponent<RagdollPoseBinder>(out var ragdollBinder))
            {
                ragdollBinder.sourceUnitNetId = sourceUnitNetId;
                ragdollBinder.lastHitPos = lastHitPosition;
                ragdollBinder.overkill = overkill;
            }

            else
            {
                Debug.LogWarning("[Ragdoll] Ragdoll prefab lacks RagdollPoseBinder component.");
            }

            NetworkServer.Spawn(go);
            return;
        }

        // offline fallback
        var off = Object.Instantiate(prefab, pos, rot);
        if (off.TryGetComponent<UnitRagdoll>(out var unitRagdoll))
        {
            unitRagdoll.SetOverkill(overkill);
            unitRagdoll.SetLastHitPosition(lastHitPosition);
            unitRagdoll.Setup(originalRootBone);
        }
    }
}
```

Assets/scripts/Oneline/Sync/NetworkSyncAgent.cs

```
using System;
using Mirror;
using UnityEngine;
/// <summary>
/// NetworkSyncAgent is a helper NetworkBehaviour to relay Commands from clients to the server.
/// Each client should have exactly one instance of this script in the scene, usually attached to the PlayerController GameObject.
///
/// Responsibilities:
/// - Receives local calls from NetworkSync (static helper).
/// - Sends Commands to the server when the local player performs an action (e.g. shooting).
/// - On the server, instantiates and spawns networked objects (like projectiles).
/// </summary>
public class NetworkSyncAgent : NetworkBehaviour
{
    public static NetworkSyncAgent Local;   // Easy access for NetworkSync static helper
    [SerializeField] private GameObject bulletPrefab; // Prefab for the bullet projectile
    [SerializeField] private GameObject grenadePrefab;

    public override void OnStartLocalPlayer()
    {
        base.OnStartLocalPlayer();
        Local = this;
    }

    /// <summary>
    /// Command from client → server.
    /// The client requests the server to spawn a bullet at the given position.
    /// The server instantiates the prefab, sets it up, and spawns it to all connected clients.
    /// </summary>
    /// <param name="spawnPos">World position where the bullet starts (usually weapon muzzle).</param>
    /// <param name="targetPos">World position the bullet is travelling towards.</param>
    [Command(requiresAuthority = true)]
    public void CmdSpawnBullet(Vector3 spawnPos, Vector3 targetPos)
    {
        if (bulletPrefab == null) { Debug.LogWarning("[NetSync] bulletPrefab missing"); return; }

        // Instantiate on the server
        var go = Instantiate(bulletPrefab, spawnPos, Quaternion.identity);

        // Setup target on the projectile
        if (go.TryGetComponent<BulletProjectile>(out var bp))
        {
            bp.Setup(targetPos);
        }

        // Spawn across the network
        NetworkServer.Spawn(go);
    }

    [Command(requiresAuthority = true)]
    public void CmdSpawnGrenade(Vector3 spawnPos, Vector3 targetPos)
    {
        if (grenadePrefab == null) { Debug.LogWarning("[NetSync] grenadePrefab missing"); return; }

        var go = Instantiate(grenadePrefab, spawnPos, Quaternion.identity);
        if (go.TryGetComponent<GrenadeProjectile>(out var gp))
            gp.Setup(targetPos); // tärkeää: ennen Spawnia

        NetworkServer.Spawn(go);
    }

    /// <summary>
    /// Client → Server: resolve target by netId and apply damage on server.
    /// then broadcast the new HP to all clients for UI.
    /// </summary>
    [Command(requiresAuthority = true)]
    public void CmdApplyDamage(uint targetNetId, int amount, Vector3 hitPosition)
    {
        if (!NetworkServer.spawned.TryGetValue(targetNetId, out var targetNi) || targetNi == null)
            return;

        var unit = targetNi.GetComponent<Unit>();
        var hs = targetNi.GetComponent<HealthSystem>();
        if (unit == null || hs == null)
            return;

        // 1) Server tekee damagen (kuten ennenkin)
        hs.Damage(amount, hitPosition);

        // 2) Heti perään broadcast → kaikki clientit päivittävät oman UI:nsa
        //    (ServerBroadcastHp kutsuu RpcNotifyHpChanged → hs.ApplyNetworkHealth(..) clientillä)
        ServerBroadcastHp(unit, hs.GetHealth(), hs.GetHealthMax());
    }

    [Command(requiresAuthority = true)]
    public void CmdApplyDamageToObject(uint targetNetId, int amount, Vector3 hitPosition)
    {
        if (!NetworkServer.spawned.TryGetValue(targetNetId, out var targetNi) || targetNi == null)
            return;

        var obj = targetNi.GetComponent<DestructibleObject>();
        if (obj == null)
            return;

        obj.Damage(amount, hitPosition);
    }

    // ---- SERVER-puolen helperit: kutsu näitä palvelimelta
    [Server]
    public void ServerBroadcastHp(Unit unit, int current, int max)
    {
        var ni = unit.GetComponent<NetworkIdentity>();
        if (ni) RpcNotifyHpChanged(ni.netId, current, max);
    }

    [Server]
    public void ServerBroadcastAp(Unit unit, int ap)
    {
        var ni = unit.GetComponent<NetworkIdentity>();
        if (ni) RpcNotifyApChanged(ni.netId, ap);
    }

    // ---- SERVER → ALL CLIENTS: HP-muutos ilmoitus
    [ClientRpc]
    void RpcNotifyHpChanged(uint unitNetId, int current, int max)
    {
        if (!NetworkClient.spawned.TryGetValue(unitNetId, out var id) || id == null) return;

        var hs = id.GetComponent<HealthSystem>();
        if (hs == null) return;

        hs.ApplyNetworkHealth(current, max);
    }

    // ---- SERVER → ALL CLIENTS: AP-muutos ilmoitus
    [ClientRpc]
    void RpcNotifyApChanged(uint unitNetId, int ap)
    {
        ApplyApClient(unitNetId, ap);
    }

    [Command]
    public void CmdMirrorAp(uint unitNetId, int ap)
    {
        RpcNotifyApChanged(unitNetId, ap);
    }

    void ApplyApClient(uint unitNetId, int ap)
    {
        if (!NetworkClient.spawned.TryGetValue(unitNetId, out var id) || id == null) return;
        var unit = id.GetComponent<Unit>();
        if (!unit) return;

        unit.ApplyNetworkActionPoints(ap); // päivittää arvon + triggaa eventin
    }
}
```

Assets/scripts/Oneline/WeaponVisibilitySync.cs

```
using Mirror;
using UnityEngine;

public class WeaponVisibilitySync : NetworkBehaviour
{
    [Header("Unit Weapons Refs")]
    [SerializeField] private Transform rifleRightHandTransform;
    [SerializeField] private Transform rifleLeftHandTransform;
    [SerializeField] private Transform meleeLeftHandTransform;
    [SerializeField] private Transform grenadeRightHandTransform;


    private NetVisibility rifleRightVis, rifleLeftVis ,meleeLeftVis, grenadeRightVis;

    void Awake()
    {
        if (rifleRightHandTransform) rifleRightVis = rifleRightHandTransform.GetComponent<NetVisibility>();
        if (rifleLeftHandTransform) rifleLeftVis= rifleLeftHandTransform.GetComponent<NetVisibility>();
        if (meleeLeftHandTransform) meleeLeftVis = meleeLeftHandTransform.GetComponent<NetVisibility>();
        if (grenadeRightHandTransform) grenadeRightVis = grenadeRightHandTransform.GetComponent<NetVisibility>();
    }

    // --- OWNER kutsuu tätä (esim. AE:ssä) ---
    public void OwnerRequestSet(bool rifleRight,bool rifleLeft, bool meleeLeft, bool grenade)
    {
        // Offline: suoraan paikalliset
        if (!NetworkClient.active && !NetworkServer.active)
        {
            SetLocal(rifleRight, rifleLeft, meleeLeft, grenade);
            return;
        }

        // Online: vain omistaja saa pyytää
        var ni = GetComponent<NetworkIdentity>();
        if (isClient && ni && ni.isOwned)
        {
            CmdSet(rifleRight, rifleLeft,meleeLeft, grenade);
        }
    }

    [Command(requiresAuthority = true)]
    private void CmdSet(bool rifleRight, bool rifleLeft ,bool meleeLeft, bool grenade)
    {
        // Serverissä voi halutessa käyttää server-authoritatiivista NetVisibilityä:
        // jos käytössä, aseta serverillä -> SyncVar/RPC hoitaa muille
        if (rifleRightVis)   rifleRightVis.ServerSetVisible(rifleRight);
        if (rifleLeftVis)   rifleLeftVis.ServerSetVisible(rifleLeft);
        if (meleeLeftVis) meleeLeftVis.ServerSetVisible(meleeLeft);
        if (grenadeRightVis) grenadeRightVis.ServerSetVisible(grenade);
        // Lisäksi varma ClientRpc (jos NetVisibility ei kata kaikkea):
        RpcSet(rifleRight, rifleLeft ,meleeLeft, grenade);
    }

    [ClientRpc]
    private void RpcSet(bool rifleRight, bool rifleLeft ,bool meleeLeft, bool grenade)
    {
        SetLocal(rifleRight, rifleLeft ,meleeLeft, grenade);
    }

    private void SetLocal(bool rifleRight,bool rifleLeft, bool meleeLeft, bool grenade)
    {
        // Jos sinulla on NetVisibility, käytä sen "pehmeää" piilotusta,
        // muuten pelkkä SetActive/renderer.enabled
        if (rifleRightHandTransform) rifleRightHandTransform.gameObject.SetActive(rifleRight);
        if (rifleLeftHandTransform) rifleLeftHandTransform.gameObject.SetActive(rifleLeft);
        if (meleeLeftHandTransform) meleeLeftHandTransform.gameObject.SetActive(meleeLeft);
        if (grenadeRightHandTransform) grenadeRightHandTransform.gameObject.SetActive(grenade);

        // Esim. renderer-tason piilotus:
        // ToggleRenderers(rifleTransform, rifle);
        // ToggleRenderers(meleeTransform, melee);
        // ToggleRenderers(grenadeTransform, grenade);
    }

    private static void ToggleRenderers(Transform t, bool visible)
    {
        if (!t) return;
        foreach (var r in t.GetComponentsInChildren<Renderer>(true))
            r.enabled = visible;
    }
}
```

Assets/scripts/PriorityQueue.cs

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

/// <summary>
/// A lightweight, generic min-heap–based Priority Queue implementation used internally for game logic,
/// especially pathfinding and AI decision-making.
///
/// This class provides a simple and efficient way to retrieve the next element with the lowest priority value.
/// It avoids external dependencies for performance and maintainability within Unity builds.
///
/// Design notes specific to RogueShooter:
/// - Used by the pathfinding and tactical AI systems to determine optimal movement and action order.
/// - Provides deterministic and garbage-free priority management during runtime (no LINQ or heap allocations).
/// - Does not support key priority updates (“decrease-key”) — instead, updated items are re-enqueued,
///   and outdated entries are safely ignored by the higher-level game logic.
///
/// In short, this queue enables efficient and predictable priority handling for all turn-based tactical calculations,
/// without relying on .NET’s built-in PriorityQueue (which is unavailable in some Unity versions).
/// </summary>
public sealed class PriorityQueue<T>
{
    private (T item, int priority)[] _heap;
    private int _count;

    public int Count => _count;

    public PriorityQueue(int initialCapacity = 64)
    {
        if (initialCapacity < 1) initialCapacity = 1;
        _heap = new (T, int)[initialCapacity];
        _count = 0;
    }

    public void Clear()
    {
        Array.Clear(_heap, 0, _count);
        _count = 0;
    }

    public void Enqueue(T item, int priority)
    {
        if (_count == _heap.Length) Array.Resize(ref _heap, _heap.Length * 2);
        _heap[_count] = (item, priority);
        SiftUp(_count++);
    }

    public T Dequeue()
    {
        if (_count == 0) throw new InvalidOperationException("PriorityQueue is empty");
        T result = _heap[0].item;
        _heap[0] = _heap[--_count];
        _heap[_count] = default;
        if (_count > 0) SiftDown(0);
        return result;
    }

    public bool TryDequeue(out T item)
    {
        if (_count == 0)
        {
            item = default;
            return false;
        }
        item = Dequeue();
        return true;
    }

    public T Peek()
    {
        if (_count == 0) throw new InvalidOperationException("PriorityQueue is empty");
        return _heap[0].item;
    }

    public int PeekPriority()
    {
        if (_count == 0) throw new InvalidOperationException("PriorityQueue is empty");
        return _heap[0].priority;
    }

    private void SiftUp(int idx)
    {
        while (idx > 0)
        {
            int parent = (idx - 1) >> 1;
            if (_heap[parent].priority <= _heap[idx].priority) break;
            (_heap[parent], _heap[idx]) = (_heap[idx], _heap[parent]);
            idx = parent;
        }
    }

    private void SiftDown(int idx)
    {
        while (true)
        {
            int left = (idx << 1) + 1;
            if (left >= _count) break;
            int right = left + 1;
            int smallest = (right < _count && _heap[right].priority < _heap[left].priority) ? right : left;
            if (_heap[idx].priority <= _heap[smallest].priority) break;
            (_heap[idx], _heap[smallest]) = (_heap[smallest], _heap[idx]);
            idx = smallest;
        }
    }
}
```

Assets/scripts/Units/EmptySquad.cs

```
using UnityEngine;

/// <summary>
/// GameNetorkManager is required to have a NetworkManager component.
/// This is an empty class just to satisfy that requirement.
/// </summary>
public class EmptySquad : MonoBehaviour
{

}
```

Assets/scripts/Units/HealthSystem.cs

```
using System;
using UnityEngine;

public class HealthSystem : MonoBehaviour
{
    public event EventHandler OnDead;
    public event EventHandler OnDamaged;

    [SerializeField] private int health = 100;
    private int healthMax;

    // To prevent multiple death events
    private bool isDead;
    private Vector3 lastHitPosition;
    public Vector3 LastHitPosition => lastHitPosition;

    private int overkill;
    public int Overkill => overkill;

    void Awake()
    {
        healthMax = health;
        isDead = false;
    }

    public void Damage(int damageAmount, Vector3 hitPosition)
    {
        if (isDead) return;

        health -= damageAmount;
        if (health <= 0)
        {
            overkill = Math.Abs(health) + 1;
            health = 0;

            if (!isDead)
            {
                lastHitPosition = hitPosition;
                isDead = true;
                Die();
            }
        }

        OnDamaged?.Invoke(this, EventArgs.Empty);
    }

    private void Die()
    {
        OnDead?.Invoke(this, EventArgs.Empty);
    }
    public float GetHealthNormalized()
    {
        return (float)health / healthMax;
    }

    public int GetHealth()
    {
        return health;
    }

    public int GetHealthMax()
    {
        return healthMax;
    }


    public void ApplyNetworkHealth(int current, int max)
    {
        healthMax = Mathf.Max(1, max);
        health    = Mathf.Clamp(current, 0, healthMax);
        OnDamaged?.Invoke(this, EventArgs.Empty);
    }
}
```

Assets/scripts/Units/Unit.cs

```
using Mirror;
using System;
using System.Collections;
using UnityEngine;

/// <summary>
///     This class represents a unit in the game.
///     Actions can be called on the unit to perform various actions like moving or shooting.
///     The class inherits from NetworkBehaviour to support multiplayer functionality.
/// </summary>
[RequireComponent(typeof(HealthSystem))]
[RequireComponent(typeof(MoveAction))]
[RequireComponent(typeof(TurnTowardsAction))]
public class Unit : NetworkBehaviour
{

    private const int ACTION_POINTS_MAX = 100;

    [SyncVar] public uint OwnerId;

    public static event EventHandler OnAnyActionPointsChanged;
    public static event EventHandler OnAnyUnitSpawned;
    public static event EventHandler OnAnyUnitDead;

    public event Action<bool> OnHiddenChangedEvent;

    [SerializeField] public bool isEnemy;

    private GridPosition gridPosition;
    private HealthSystem healthSystem;

    private BaseAction[] baseActionsArray;

    private int actionPoints = ACTION_POINTS_MAX;

    private int maxMoveDistance;

    [SyncVar(hook = nameof(OnHiddenChanged))]
    private bool isHidden;

    private Renderer[] renderers;
    private Collider[] colliders;
    private Animator anim;

    private void Awake()
    {
        renderers = GetComponentsInChildren<Renderer>(true);
        colliders = GetComponentsInChildren<Collider>(true);
        TryGetComponent(out anim);
        healthSystem = GetComponent<HealthSystem>();
        baseActionsArray = GetComponents<BaseAction>();
        maxMoveDistance = GetComponent<MoveAction>().GetMaxMoveDistance();
    }

    private void Start()
    {
        if (LevelGrid.Instance != null)
        {
            gridPosition = LevelGrid.Instance.GetGridPosition(transform.position);
            LevelGrid.Instance.AddUnitAtGridPosition(gridPosition, this);
        }

        TurnSystem.Instance.OnTurnChanged += TurnSystem_OnTurnChanged;

        healthSystem.OnDead += HealthSystem_OnDead;

        OnAnyUnitSpawned?.Invoke(this, EventArgs.Empty);
    }

    private void Update()
    {
        GridPosition newGridPosition = LevelGrid.Instance.GetGridPosition(transform.position);
        if (newGridPosition != gridPosition)
        {
            GridPosition oldGridposition = gridPosition;
            gridPosition = newGridPosition;
            LevelGrid.Instance.UnitMoveToGridPosition(oldGridposition, newGridPosition, this);
        }
    }

    /// <summary>
    ///     When unit get destroyed, this clears grid system under destroyed unit.
    ///
    /// </summary>
    void OnDestroy()
    {
        if (LevelGrid.Instance != null)
        {
            gridPosition = LevelGrid.Instance.GetGridPosition(transform.position);
            LevelGrid.Instance.RemoveUnitAtGridPosition(gridPosition, this);
        }
    }

    public T GetAction<T>() where T : BaseAction
    {
        foreach (BaseAction baseAction in baseActionsArray)
        {
            if (baseAction is T t)
            {
                return t;
            }
        }
        return null;
    }

    public GridPosition GetGridPosition()
    {
        return gridPosition;
    }

    public Vector3 GetWorldPosition()
    {
        return transform.position;
    }

    public BaseAction[] GetBaseActionsArray()
    {
        return baseActionsArray;
    }

    public bool TrySpendActionPointsToTakeAction(BaseAction baseAction)
    {
        if (CanSpendActionPointsToTakeAction(baseAction))
        {
            SpendActionPoints(baseAction.GetActionPointsCost());
            return true;
        }
        return false;
    }

    public bool CanSpendActionPointsToTakeAction(BaseAction baseAction)
    {
        if (actionPoints >= baseAction.GetActionPointsCost())
        {
            return true;
        }
        return false;
    }

    private void SpendActionPoints(int amount)
    {
        actionPoints -= amount;

        OnAnyActionPointsChanged?.Invoke(this, EventArgs.Empty);
        NetworkSync.BroadcastActionPoints(this, actionPoints);
    }

    public int GetActionPoints()
    {
        return actionPoints;
    }

    /// <summary>
    ///     This method is called when the turn changes. It resets the action points to the maximum value.
    /// </summary>
    private void TurnSystem_OnTurnChanged(object sender, EventArgs e)
    {
        actionPoints = ACTION_POINTS_MAX;
        OnAnyActionPointsChanged?.Invoke(this, EventArgs.Empty);
    }

    /// <summary>
    ///    Online: Updating ActionPoints usage to otherplayers.
    /// </summary>
    public void ApplyNetworkActionPoints(int ap)
    {
        if (actionPoints == ap) return;
        actionPoints = ap;
        OnAnyActionPointsChanged?.Invoke(this, EventArgs.Empty);
    }

    public bool IsEnemy()
    {
        return isEnemy;
    }

    private void HealthSystem_OnDead(object sender, System.EventArgs e)
    {
        OnAnyUnitDead?.Invoke(this, EventArgs.Empty);
        if (!NetworkServer.active)
        {
            // OFFLINE: suoraan tuho
            if (!NetworkClient.active) { Destroy(gameObject); return; }
            return;
        }

        // Piilota jotta client ehtii kopioida omaan ragdolliin tiedot
        isHidden = true;
        SetSoftHiddenLocal(true);
        StartCoroutine(DestroyAfter(0.30f));

    }

    private IEnumerator DestroyAfter(float seconds)
    {
        yield return new WaitForSeconds(seconds);
        NetworkServer.Destroy(gameObject);
    }

    private void SetSoftHiddenLocal(bool hidden)
    {
        bool visible = !hidden;
        foreach (var r in renderers) if (r) r.enabled = visible;
        foreach (var c in colliders) if (c) c.enabled = visible;
        if (anim) anim.enabled = visible;
    }

    public float GetHealthNormalized()
    {
        return healthSystem.GetHealthNormalized();
    }

    private void OnHiddenChanged(bool oldVal, bool newVal)
    {
        OnHiddenChangedEvent?.Invoke(newVal);
    }

    public bool IsHidden()
    {
        return isHidden;
    }

    public int GetMaxMoveDistance()
    {
        return maxMoveDistance;
    }
}
```

Assets/scripts/Units/UnitActions/Actions/BaseAction.cs

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


/// <summary>
/// Base class for all unit actions in the game.
/// This class inherits from NetworkBehaviour and provides common functionality for unit actions.
/// </summary>
[RequireComponent(typeof(Unit))]
public abstract class BaseAction : NetworkBehaviour
{
    public static event EventHandler OnAnyActionStarted;
    public static event EventHandler OnAnyActionCompleted;


    protected Unit unit;
    protected bool isActive;
    protected Action onActionComplete;

    protected virtual void Awake()
    {
        unit = GetComponent<Unit>();
    }

    // Defines the action button text for the Unit UI.
    public abstract string GetActionName();

    // Executes the action at the specified grid position and invokes the callback upon completion.
    public abstract void TakeAction(GridPosition gridPosition, Action onActionComplete);

    // Checks if the specified grid position is valid for the action, when mouse is over a grid position.
    public virtual bool IsValidGridPosition(GridPosition gridPosition)
    {
        List<GridPosition> validGridPositionsList = GetValidGridPositionList();
        return validGridPositionsList.Contains(gridPosition);
    }

    // Returns a list of valid grid positions for the action.
    public abstract List<GridPosition> GetValidGridPositionList();

    // Returns the action points cost for performing the action.
    public virtual int GetActionPointsCost()
    {
        return 1;
    }

    // Called when the action starts, sets the action as active and stores the completion callback.
    // Prevents the player from performing multiple actions at the same time.
    protected void ActionStart(Action onActionComplete)
    {
        isActive = true;
        this.onActionComplete = onActionComplete;

        OnAnyActionStarted?.Invoke(this, EventArgs.Empty);
    }

    // Called when the action is completed, sets the action as inactive and invokes the completion callback.
    // Allows the player to perform new actions.
    protected void ActionComplete()
    {
        isActive = false;
        onActionComplete();

        OnAnyActionCompleted?.Invoke(this, EventArgs.Empty);
    }

    public Unit GetUnit()
    {
        return unit;
    }

    public void MakeDamage(int damage, Unit targetUnit)
    {
        // Peruspaikat (world-space)
        Vector3 attacerPos = unit.GetWorldPosition() + Vector3.up * 1.6f;   // silmä/rinta
        Vector3 targetPos  = targetUnit.GetWorldPosition() + Vector3.up * 1.2f;

        // Suunta
        Vector3 dir = targetPos - attacerPos;
        if (dir.sqrMagnitude < 0.0001f) dir = targetUnit.transform.forward;  // fallback
        dir.Normalize();

        // Siirrä osumakeskus hieman kohti hyökkääjää (0.5–1.0 m toimii yleensä hyvin)
        float backOffset = 0.7f;
        Vector3 hitPosition = targetPos - dir * backOffset;

        // (valinnainen) pieni satunnainen sivuttaisjitter, ettei kaikki näytä identtiseltä
        Vector3 side = Vector3.Cross(dir, Vector3.up).normalized;
        hitPosition += side * UnityEngine.Random.Range(-0.1f, 0.1f);

        NetworkSync.ApplyDamageToUnit(targetUnit, damage, hitPosition);
    }

    public enum RotateTargetType
    {
        Unit,
        GridPosition
    }

    public bool RotateTowards(Vector3 targetPosition, float rotationSpeed = 10f)
    {
         // Suuntavektori
        Vector3 aimDirection = (targetPosition - unit.GetWorldPosition()).normalized;
        aimDirection.y = 0f;

        transform.forward = Vector3.Slerp(transform.forward, aimDirection, Time.deltaTime * rotationSpeed);

        // Kääntyminen on suoritettu.
        float tolerance = 0.99f;
        float dot = Vector3.Dot(transform.forward.normalized, aimDirection);
        return dot > tolerance;
    }

    // -------------- ENEMY AI ACTIONS -------------

    /// <summary>
    /// ENEMY AI:
    /// Empty ENEMY AI ACTIONS abstract class.
    /// Every Unit action like MoveAction.cs, ShootAction.cs and so on defines this differently
    /// Contains gridposition and action value
    /// </summary>
    public abstract EnemyAIAction GetEnemyAIAction(GridPosition gridPosition);

    /// <summary>
    /// ENEMY AI:
    /// Making a list all possible actions an enemy Unit can take, and shorting them
    /// based on highest action value.(Gives the enemy the best outcome)
    /// The best Action is in the enemyAIActionList[0]
    /// </summary>
    public EnemyAIAction GetBestEnemyAIAction()
    {
        List<EnemyAIAction> enemyAIActionList = new();

        List<GridPosition> validActionGridPositionList = GetValidGridPositionList();


        foreach (GridPosition gridPosition in validActionGridPositionList)
        {
            // All actions have own EnemyAIAction to set griposition and action value.
            EnemyAIAction enemyAIAction = GetEnemyAIAction(gridPosition);
            enemyAIActionList.Add(enemyAIAction);
        }

        if (enemyAIActionList.Count > 0)
        {
            enemyAIActionList.Sort((a, b) => b.actionValue - a.actionValue);
            return enemyAIActionList[0];
        }
        else
        {
            // No possible Enemy AI Actions
            return null;
        }
    }
}
```

Assets/scripts/Units/UnitActions/Actions/GranadeAction.cs

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

public class GranadeAction : BaseAction
{
    public event EventHandler ThrowGranade;

    public event EventHandler ThrowReady;

    public Vector3 TargetWorld { get; private set; }

    [SerializeField] private Transform grenadeProjectilePrefab;

    private int maxThrowDistance = 7;

    private void Update()
    {
        if (!isActive)
        {
            return;
        }
    }

    public override string GetActionName()
    {
        return "Granade";
    }

    public override EnemyAIAction GetEnemyAIAction(GridPosition gridPosition)
    {
        return new EnemyAIAction
        {
            gridPosition = gridPosition,
            actionValue = 0,

        };
    }

    public override List<GridPosition> GetValidGridPositionList()
    {

        List<GridPosition> validGridPositionList = new();

        GridPosition unitGridPosition = unit.GetGridPosition();

        for (int x = -maxThrowDistance; x <= maxThrowDistance; x++)
        {
            for (int z = -maxThrowDistance; z <= maxThrowDistance; z++)
            {
                GridPosition offsetGridPosition = new(x, z, 0);
                GridPosition testGridPosition = unitGridPosition + offsetGridPosition;

                // Check if the test grid position is within the valid range
                if (!LevelGrid.Instance.IsValidGridPosition(testGridPosition)) continue;
                int testDistance = Mathf.Abs(x) + Mathf.Abs(z);
                if (testDistance > maxThrowDistance) continue;

                validGridPositionList.Add(testGridPosition);
            }

        }

        return validGridPositionList;
    }

    public override void TakeAction(GridPosition gridPosition, Action onActionComplete)
    {

        ActionStart(onActionComplete);
        TargetWorld = LevelGrid.Instance.GetWorldPosition(gridPosition);
        StartCoroutine(TurnAndThrow(.5f, TargetWorld));


    }

    private IEnumerator TurnAndThrow(float delay, Vector3 targetWorld)
    {
        // Odotetaan kunnes RotateTowards palaa true
        float waitAfterAligned = 0.1f; // pienen odotuksen verran
        float alignedTime = 0f;

        while (true)
        {
            bool aligned = RotateTowards(targetWorld);

            if (aligned)
            {
                alignedTime += Time.deltaTime;
                if (alignedTime >= waitAfterAligned)
                    break; // ollaan kohdistettu ja odotettu tarpeeksi
            }
            else
            {
                alignedTime = 0f; // resetoi jos ei vielä kohdallaan
            }

            yield return null;
        }

        ThrowGranade?.Invoke(this, EventArgs.Empty);
    }

    public void OnGrenadeBehaviourComplete()
    {
        ThrowReady?.Invoke(this, EventArgs.Empty);
        ActionComplete();
    }
}
```

Assets/scripts/Units/UnitActions/Actions/InteractAction.cs

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

public class InteractAction : BaseAction
{
    private void Update()
    {
        if (!isActive)
        {
            return;
        }
    }

    public override string GetActionName()
    {
        return "Interact";
    }

    public override EnemyAIAction GetEnemyAIAction(GridPosition gridPosition)
    {
        return new EnemyAIAction
        {
            gridPosition = gridPosition,
            actionValue = 0,
        };
    }

    public override List<GridPosition> GetValidGridPositionList()
    {
        List<GridPosition> validGridPositionList = new();

        GridPosition unitGridPosition = unit.GetGridPosition();

        for (int x = -1; x <= 1; x++)
        {
            for (int z = -1; z <= 1; z++)
            {
                GridPosition offsetGridPosition = new(x, z, 0);
                GridPosition testGridPosition = unitGridPosition + offsetGridPosition;

                if (!LevelGrid.Instance.IsValidGridPosition(testGridPosition)) continue;
                IInteractable interactable = LevelGrid.Instance.GetInteractableAtGridPosition(testGridPosition);
                if (interactable == null) continue;
                validGridPositionList.Add(testGridPosition);
            }
        }

        return validGridPositionList;
    }
    public override void TakeAction(GridPosition gridPosition, Action onActionComplete)
    {
        IInteractable interactable = LevelGrid.Instance.GetInteractableAtGridPosition(gridPosition);
        interactable.Interact(OnInteractComplete);
        ActionStart(onActionComplete);
    }

    private void OnInteractComplete()
    {
        ActionComplete();
    }
}
```

Assets/scripts/Units/UnitActions/Actions/MeleeAction.cs

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

public class MeleeAction : BaseAction
{
    public static event EventHandler OnAnyMeleeActionHit;

    public event EventHandler OnMeleeActionStarted;
    public event EventHandler OnMeleeActionCompleted;
    [SerializeField] private int damage = 100;

    private enum State
    {
        MeleeActionBeforeHit,
        MeleeActionAfterHit,
    }
    private int maxMeleedDistance = 1;
    private State state;
    private float stateTimer;
    private Unit targetUnit;

    private void Update()
    {
        if (!isActive)
        {
            return;
        }
        stateTimer -= Time.deltaTime;
        switch (state)
        {
            case State.MeleeActionBeforeHit:
                if (targetUnit != null)
                {
                    if (RotateTowards(targetUnit.GetWorldPosition()))
                    {
                        stateTimer = Mathf.Min(stateTimer, 0.4f);
                    }
                }
                break;
            case State.MeleeActionAfterHit:
                break;
        }

        if (stateTimer <= 0f)
        {
            NextState();
        }
    }
    private void NextState()
    {
        switch (state)
        {
            case State.MeleeActionBeforeHit:
                state = State.MeleeActionAfterHit;
                float afterHitStateTime = 1f;
                stateTimer = afterHitStateTime;
                MakeDamage(damage, targetUnit);
                OnAnyMeleeActionHit?.Invoke(this, EventArgs.Empty);
                break;
            case State.MeleeActionAfterHit:
                OnMeleeActionCompleted?.Invoke(this, EventArgs.Empty);
                ActionComplete();
                break;
        }
    }

    public override string GetActionName()
    {
        return "Melee";
    }

    public override List<GridPosition> GetValidGridPositionList()
    {
        List<GridPosition> validGridPositionList = new();

        GridPosition unitGridPosition = unit.GetGridPosition();

        for (int x = -maxMeleedDistance; x <= maxMeleedDistance; x++)
        {
            for (int z = -maxMeleedDistance; z <= maxMeleedDistance; z++)
            {
                GridPosition offsetGridPosition = new(x, z, 0);
                GridPosition testGridPosition = unitGridPosition + offsetGridPosition;

                if (!LevelGrid.Instance.HasAnyUnitOnGridPosition(testGridPosition)) continue;

                Unit targetUnit = LevelGrid.Instance.GetUnitAtGridPosition(testGridPosition);
                // Make sure we don't include friendly units.
                if (targetUnit.IsEnemy() == unit.IsEnemy()) continue;
                // Check if the test grid position is within the valid range
                if (!LevelGrid.Instance.IsValidGridPosition(testGridPosition)) continue;

                validGridPositionList.Add(testGridPosition);
            }
        }

        return validGridPositionList;
    }

    public override void TakeAction(GridPosition gridPosition, Action onActionComplete)
    {
        targetUnit = LevelGrid.Instance.GetUnitAtGridPosition(gridPosition);

        state = State.MeleeActionBeforeHit;
        float beforeHitStateTime = 0.7f;
        stateTimer = beforeHitStateTime;
        OnMeleeActionStarted?.Invoke(this, EventArgs.Empty);
        ActionStart(onActionComplete);
    }

    //-------------- ENEMY AI ACTIONS -------------
    public override EnemyAIAction GetEnemyAIAction(GridPosition gridPosition)
    {
        return new EnemyAIAction
        {
            gridPosition = gridPosition,
            actionValue = 200,
        };
    }
}
```

Assets/scripts/Units/UnitActions/Actions/MoveAction.cs

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


/// <summary>
/// The MoveAction class is responsible for handling the movement of a unit in the game.
/// It allows the unit to move to a target position, and it calculates valid move grid positions based on the unit's current position.
/// </summary>
public class MoveAction : BaseAction
{

    public event EventHandler OnStartMoving;
    public event EventHandler OnStopMoving;
    [SerializeField] private int maxMoveDistance = 4;
    private List<Vector3> positionList;
    private int currentPositionIndex;

    private bool isChangingFloors;
    private float differentFloorsTeleportTimer;
    private float differentFloorsTeleportTimerMax = .5f;


    private void Update()
    {
        if (!isActive) return;

        Vector3 targetPosition = positionList[currentPositionIndex];

        if (isChangingFloors)
        {
            Vector3 targetSameFloorPosition = targetPosition;
            targetSameFloorPosition.y = transform.position.y;
            Vector3 rotateDirection = (targetSameFloorPosition - transform.position).normalized;

            float rotationSpeed = 10f;
            transform.forward = Vector3.Slerp(transform.forward, rotateDirection, Time.deltaTime * rotationSpeed);
            differentFloorsTeleportTimer -= Time.deltaTime;
            if (differentFloorsTeleportTimer < 0f)
            {
                isChangingFloors = false;
                transform.position = targetPosition;
            }
        }
        else
        {

            Vector3 moveDirection = (targetPosition - transform.position).normalized;

            // Rotate towards the target position
            float rotationSpeed = 10f;
            transform.forward = Vector3.Slerp(transform.forward, moveDirection, Time.deltaTime * rotationSpeed);

            // Move towards the target position
            float moveSpeed = 6f;
            transform.position += moveSpeed * Time.deltaTime * moveDirection;
        }


        float stoppingDistance = 0.2f;
        if (Vector3.Distance(transform.position, targetPosition) < stoppingDistance)
        {
            currentPositionIndex++;
            if (currentPositionIndex >= positionList.Count)
            {
                OnStopMoving?.Invoke(this, EventArgs.Empty);
                ActionComplete();
            }
            else
            {
                targetPosition = positionList[currentPositionIndex];
                GridPosition targetGridPosition = LevelGrid.Instance.GetGridPosition(targetPosition);
                GridPosition unitGridPosition = LevelGrid.Instance.GetGridPosition(transform.position);

                if (targetGridPosition.floor != unitGridPosition.floor)
                {
                    //Different floors
                    isChangingFloors = true;
                    differentFloorsTeleportTimer = differentFloorsTeleportTimerMax;
                }
            }
        }
    }

    public override void TakeAction(GridPosition gridPosition, Action onActionComplete)
    {
        List<GridPosition> pathGridPositionsList = PathFinding.Instance.FindPath(unit.GetGridPosition(), gridPosition, out int pathLeght, maxMoveDistance);

        currentPositionIndex = 0;
        positionList = new List<Vector3>();

        foreach (GridPosition pathGridPosition in pathGridPositionsList)
        {
            positionList.Add(LevelGrid.Instance.GetWorldPosition(pathGridPosition));

        }

        OnStartMoving?.Invoke(this, EventArgs.Empty);
        ActionStart(onActionComplete);
    }

    public override List<GridPosition> GetValidGridPositionList()
    {
        var valid = new List<GridPosition>();
        var candidates = new HashSet<GridPosition>(); // estää duplikaatit

        GridPosition unitPos = unit.GetGridPosition();
        int startFloor = unitPos.floor;

        // Jos maxMoveDistance on RUUTUJA, kustannusbudjetti on *10 per ruutu*
        const int COST_PER_TILE = 10;
        int moveBudgetCost = maxMoveDistance * COST_PER_TILE;

        // --- 1) Nykyisen kerroksen ruudut (perus-offsetit) ---
        for (int dx = -maxMoveDistance; dx <= maxMoveDistance; dx++)
        {
            for (int dz = -maxMoveDistance; dz <= maxMoveDistance; dz++)
            {
                var test = new GridPosition(unitPos.x + dx, unitPos.z + dz, startFloor);
                candidates.Add(test);
            }
        }

        // --- 2) Linkkien kautta saavutettavat kerrokset (hybridi) ---
        var links = PathFinding.Instance.GetPathfindingLinks();
        if (links != null && links.Count > 0)
        {
            foreach (var link in links)
            {
                // A -> B
                if (link.gridPositionA.floor == startFloor)
                {
                    int lbToA = PathFinding.Instance.CalculateDistance(unitPos, link.gridPositionA);
                    if (lbToA <= moveBudgetCost)
                    {
                        int remaining = moveBudgetCost - lbToA;
                        int radiusTiles = Mathf.Max(0, remaining / COST_PER_TILE);

                        for (int dx = -radiusTiles; dx <= radiusTiles; dx++)
                        {
                            for (int dz = -radiusTiles; dz <= radiusTiles; dz++)
                            {
                                var aroundB = new GridPosition(
                                    link.gridPositionB.x + dx,
                                    link.gridPositionB.z + dz,
                                    link.gridPositionB.floor
                                );
                                candidates.Add(aroundB);
                            }
                        }
                    }
                }

                // B -> A
                if (link.gridPositionB.floor == startFloor)
                {
                    int lbToB = PathFinding.Instance.CalculateDistance(unitPos, link.gridPositionB);
                    if (lbToB <= moveBudgetCost)
                    {
                        int remaining = moveBudgetCost - lbToB;
                        int radiusTiles = Mathf.Max(0, remaining / COST_PER_TILE);

                        for (int dx = -radiusTiles; dx <= radiusTiles; dx++)
                        {
                            for (int dz = -radiusTiles; dz <= radiusTiles; dz++)
                            {
                                var aroundA = new GridPosition(
                                    link.gridPositionA.x + dx,
                                    link.gridPositionA.z + dz,
                                    link.gridPositionA.floor
                                );
                                candidates.Add(aroundA);
                            }
                        }
                    }
                }
            }
        }

        // --- 3) Suodata & tee vain yksi A* per kandidaatti (välimuistilla) ---
        foreach (var test in candidates)
        {
            // Perusvalidoinnit
            if (!LevelGrid.Instance.IsValidGridPosition(test)) continue;
            if (test == unitPos) continue;
            if (LevelGrid.Instance.HasAnyUnitOnGridPosition(test)) continue;
            if (!PathFinding.Instance.IsWalkableGridPosition(test)) continue;

            // Heuristiikkakarsinta (Manhattan*10): jos edes optimistinen kustannus > budjetti, skip
            int lowerBound = PathFinding.Instance.CalculateDistance(unitPos, test);
            if (lowerBound > moveBudgetCost) continue;

            // *** VAIN YKSI A* per ruutu (mutta nyt cachetettuna saman framen sisällä) ***
            if (!TryGetPathCostCached(unitPos, test, out int pathCost)) continue; // ei polkua
            if (pathCost > moveBudgetCost) continue;

            valid.Add(test);
        }

        return valid;
    }

    public override string GetActionName()
    {
        return "Move";
    }
    // --- Per-frame pathfinding cache ---
    private struct PathQuery : IEquatable<PathQuery> {
        public GridPosition start;
        public GridPosition end;
        public bool Equals(PathQuery other) => start == other.start && end == other.end;
        public override bool Equals(object obj) => obj is PathQuery pq && Equals(pq);
        public override int GetHashCode() => (start.GetHashCode() * 397) ^ end.GetHashCode();
    }

    private struct PathCacheEntry {
        public bool exists;
        public int cost;
        // Jos joskus haluat itse polun, voit lisätä: public List<GridPosition> path;
    }

    // Yhteinen cache tälle actionille (voisi olla myös static jos haluat jakaa yli instanssien)
    private Dictionary<PathQuery, PathCacheEntry> _pathCache = new Dictionary<PathQuery, PathCacheEntry>(256);
    private int _cacheFrame = -1;

    private bool TryGetPathCostCached(GridPosition start, GridPosition end, out int cost)
    {
        // Nollaa cache kerran per frame
        int frame = Time.frameCount;
        if (_cacheFrame != frame) {
            _pathCache.Clear();
            _cacheFrame = frame;
        }

        var key = new PathQuery { start = start, end = end };
        if (_pathCache.TryGetValue(key, out var entry)) {
            cost = entry.cost;
            return entry.exists;
        }

        // Ei ollut välimuistissa -> laske kerran
        var path = PathFinding.Instance.FindPath(start, end, out int pathCost, maxMoveDistance);
        bool exists = path != null;
        _pathCache[key] = new PathCacheEntry { exists = exists, cost = pathCost };

        cost = pathCost;
        return exists;
    }

    public int GetMaxMoveDistance()
    {
        return maxMoveDistance;
    }

    /// <summary>
    /// ENEMY AI:
    /// Move toward to Player unit to make shoot action.
    /// </summary>
    public override EnemyAIAction GetEnemyAIAction(GridPosition gridPosition)
    {
        int targetCountAtGridPosition = unit.GetAction<ShootAction>().GetTargetCountAtPosition(gridPosition);

        return new EnemyAIAction
        {
            gridPosition = gridPosition,
            actionValue = targetCountAtGridPosition * 10,
        };
    }
}
```

Assets/scripts/Units/UnitActions/Actions/ShootAction.cs

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

public class ShootAction : BaseAction
{

    public static event EventHandler<OnShootEventArgs> OnAnyShoot;

    public event EventHandler<OnShootEventArgs> OnShoot;

    public class OnShootEventArgs : EventArgs
    {
        public Unit targetUnit;
        public Unit shootingUnit;
    }

    private enum State
    {
        Aiming,
        Shooting,
        Cooloff
    }

    [SerializeField] private LayerMask obstaclesLayerMask;
    private State state;
    [SerializeField] private int maxShootDistance = 7;
    [SerializeField] private int damage = 30;

    private float stateTimer;
    private Unit targetUnit;
    private bool canShootBullet;

    // Update is called once per frame
    void Update()
    {
        if (!isActive) return;

        stateTimer -= Time.deltaTime;
        switch (state)
        {
            case State.Aiming:
                if (targetUnit != null)
                {
                    if (RotateTowards(targetUnit.GetWorldPosition()))
                    {
                        stateTimer = Mathf.Min(stateTimer, 0.4f);
                    }
                 }
                break;
            case State.Shooting:
                if (canShootBullet)
                {
                    Shoot();
                    canShootBullet = false;

                }
                break;
            case State.Cooloff:
                break;
        }

        if (stateTimer <= 0f)
        {
            NextState();
        }
    }

    private void NextState()
    {
        switch (state)
        {
            case State.Aiming:
                state = State.Shooting;
                float shootingStateTime = 0.1f;
                stateTimer = shootingStateTime;
                break;
            case State.Shooting:
                state = State.Cooloff;
                float cooloffStateTime = 0.5f;
                stateTimer = cooloffStateTime;
                break;
            case State.Cooloff:
                ActionComplete();
                break;
        }
    }

    private void Shoot()
    {
        OnAnyShoot?.Invoke(this, new OnShootEventArgs
        {
            targetUnit = targetUnit,
            shootingUnit = unit
        });

        OnShoot?.Invoke(this, new OnShootEventArgs
        {
            targetUnit = targetUnit,
            shootingUnit = unit
        });
        MakeDamage(damage, targetUnit);

    }

    public override int GetActionPointsCost()
    {
        return 1;
    }

    public override string GetActionName()
    {
        return "Shoot";
    }

    public  List<GridPosition> GetValidActionGridPositionList(GridPosition unitGridPosition)
    {
        List<GridPosition> validGridPositionList = new();

        for (int x = -maxShootDistance; x <= maxShootDistance; x++)
        {
            for (int z = -maxShootDistance; z <= maxShootDistance; z++)
            {
                for (int floor = -maxShootDistance; floor <= maxShootDistance; floor++)
                {
                    GridPosition offsetGridPosition = new(x, z, floor);
                    GridPosition testGridPosition = unitGridPosition + offsetGridPosition;

                    // Check if the test grid position is within the valid range and not occupied by another unit
                    if (!LevelGrid.Instance.IsValidGridPosition(testGridPosition)) continue;
                    int testDistance = Mathf.Abs(x) + Mathf.Abs(z);
                    if (testDistance > maxShootDistance) continue;
                    if (!LevelGrid.Instance.HasAnyUnitOnGridPosition(testGridPosition)) continue;

                    Unit targetUnit = LevelGrid.Instance.GetUnitAtGridPosition(testGridPosition);
                    if (targetUnit == null) continue;
                    // Make sure we don't include friendly units.
                    if (targetUnit.IsEnemy() == unit.IsEnemy()) continue;

                    Vector3 unitWorldPosition = LevelGrid.Instance.GetWorldPosition(unitGridPosition);
                    Vector3 shootDir = (targetUnit.GetWorldPosition() - unitWorldPosition).normalized;
                    float unitShoulderHeight = 2.5f;
                    if (Physics.Raycast(
                        unitWorldPosition + Vector3.up * unitShoulderHeight,
                        shootDir,
                        Vector3.Distance(unitWorldPosition, targetUnit.GetWorldPosition()),
                        obstaclesLayerMask))
                    {
                        //Target Unit is Blocked by an Obstacle
                        continue;
                    }

                    validGridPositionList.Add(testGridPosition);
                }
            }

        }

        return validGridPositionList;
    }

    public override void TakeAction(GridPosition gridPosition, Action onActionComplete)
    {

        targetUnit = LevelGrid.Instance.GetUnitAtGridPosition(gridPosition);

        state = State.Aiming;
        float aimingStateTime = 1f;
        stateTimer = aimingStateTime;

        canShootBullet = true;

        ActionStart(onActionComplete);
    }

    public Unit GetTargetUnit()
    {
        return targetUnit;
    }

    public int GetMaxShootDistance()
    {
        return maxShootDistance;
    }

    /// ---------------- AI ----------------
    /// <summary>
    /// ENEMY AI: Make a list about Player Units what Enemy Unit can shoot.
    /// </summary>
    public override List<GridPosition> GetValidGridPositionList()
    {
        GridPosition unitGridPosition = unit.GetGridPosition();
        return GetValidActionGridPositionList(unitGridPosition);
    }

    /// <summary>
    /// ENEMY AI: How "good" target is. Target who have a lowest health, gets a higher actionvalue
    /// </summary>
    public override EnemyAIAction GetEnemyAIAction(GridPosition gridPosition)
    {
        Unit targetUnit = LevelGrid.Instance.GetUnitAtGridPosition(gridPosition);

        return new EnemyAIAction
        {
            gridPosition = gridPosition,
            actionValue = 100 + Mathf.RoundToInt((1 - targetUnit.GetHealthNormalized()) * 100f), //Take at target who have a lowest health.
        };
    }

    public int GetTargetCountAtPosition(GridPosition gridPosition)
    {
        return GetValidActionGridPositionList(gridPosition).Count;
    }
}
```

Assets/scripts/Units/UnitActions/Actions/TurnTowardsAction.cs

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




/// <summary>
///     This class is responsible for spinning a unit around its Y-axis.
/// </summary>
/// remarks>
///     Change to turn towards the direction the mouse is pointing
/// </remarks>

public class TurnTowardsAction : BaseAction
{
    private enum State
    {
        StartTurning,
        EndTurning,
    }
     private State state;
    public Vector3 TargetWorld { get; private set; }

    private float stateTimer;
    GridPosition gridPosition;

    private void Update()
    {
        if (!isActive)
        {
            return;
        }
        stateTimer -= Time.deltaTime;
        switch (state)
        {
            case State.StartTurning:
                if (RotateTowards(TargetWorld))
                {
                    stateTimer = 0;
                }
                break;
            case State.EndTurning:
                break;
        }

        if (stateTimer <= 0f)
        {
            NextState();
        }

    }

    private void NextState()
    {
        switch (state)
        {
            case State.StartTurning:
                state = State.EndTurning;
                float afterTurnStateTime = 0.5f;
                stateTimer = afterTurnStateTime;

                break;
            case State.EndTurning:
                ActionComplete();
                break;
        }
    }
    public override void TakeAction(GridPosition gridPosition, Action onActionComplete)
    {
        TargetWorld = LevelGrid.Instance.GetWorldPosition(gridPosition);
        this.gridPosition = gridPosition;
        state = State.StartTurning;
        float beforeTurnStateTime = 0.7f;
        stateTimer = beforeTurnStateTime;
        ActionStart(onActionComplete);
    }

    public override string GetActionName()
    {
        return "Turn";
    }

    public override List<GridPosition> GetValidGridPositionList()
    {
        List<GridPosition> validGridPositionList = new();

        GridPosition unitGridPosition = unit.GetGridPosition();

        for (int x = -1; x <= 1; x++)
        {
            for (int z = -1; z <= 1; z++)
            {
                GridPosition offsetGridPosition = new(x, z, 0);
                GridPosition testGridPosition = unitGridPosition + offsetGridPosition;
                validGridPositionList.Add(testGridPosition);
            }
        }

        return validGridPositionList;
    }
    public override int GetActionPointsCost()
    {
        return 100;
    }

    /// <summary>
    /// ENEMY AI:
    /// Currently this action has no value. Just testing!
    /// </summary>

    public override EnemyAIAction GetEnemyAIAction(GridPosition gridPosition)
    {
        return new EnemyAIAction
        {
            gridPosition = gridPosition,
            actionValue = 0,

        };
    }

}
```

Assets/scripts/Units/UnitActions/ScreenShakeActions.cs

```
using System;
using UnityEngine;

public class ScreenShakeActions : MonoBehaviour
{
    private void Start()
    {
        ShootAction.OnAnyShoot += ShootAction_OnAnyShoot;
        GrenadeProjectile.OnAnyGranadeExploded += GrenadeProjectile_OnAnyGranadeExploded;
        MeleeAction.OnAnyMeleeActionHit += MeleeAction_OnAnyMeleeActionHit;
    }

    private void OnDisable()
    {
        ShootAction.OnAnyShoot -= ShootAction_OnAnyShoot;
        GrenadeProjectile.OnAnyGranadeExploded -= GrenadeProjectile_OnAnyGranadeExploded;
    }

    private void ShootAction_OnAnyShoot(object sender, ShootAction.OnShootEventArgs e)
    {
        ScreenShake.Instance.RecoilCameraShake(1f);
    }

    private void GrenadeProjectile_OnAnyGranadeExploded(object sender, EventArgs e)
    {
        ScreenShake.Instance.ExplosiveCameraShake(2f);
    }

    private void MeleeAction_OnAnyMeleeActionHit(object sender, EventArgs e)
    {
        ScreenShake.Instance.RecoilCameraShake(3f);
    }

}
```

Assets/scripts/Units/UnitActions/UnitActionSystem.cs

```
using System;
using UnityEngine;
using UnityEngine.EventSystems;

/// <summary>
/// This script handles the unit action system in the game.
/// It allows the player to select units and perform actions on them, such as moving or shooting.
/// It also manages the state of the selected unit and action, and prevents the player from performing multiple actions at the same time.
/// Note: This class Script Execution Order is set to be executed before UnitManager.cs. High priority.
/// </summary>
public class UnitActionSystem : MonoBehaviour
{
    public static UnitActionSystem Instance { get; private set; }

    public event EventHandler OnSelectedUnitChanged;
    public event EventHandler OnSelectedActionChanged;
    public event EventHandler<bool> OnBusyChanged;
    public event EventHandler OnActionStarted;

    // This allows the script to only interact with objects on the specified layer
    [SerializeField] private LayerMask unitLayerMask;
    [SerializeField] private Unit selectedUnit;

    private BaseAction selectedAction;

    // Prevents the player from performing multiple actions at the same time
    private bool isBusy;

    private void Awake()
    {
        selectedUnit = null;
        // Ensure that there is only one instance in the scene
        if (Instance != null)
        {
            Debug.LogError("UnitActionSystem: More than one UnitActionSystem in the scene!" + transform + " " + Instance);
            Destroy(gameObject);
            return;
        }
        Instance = this;
    }

    private void Start()
    {

    }
    private void Update()
    {
//        Debug.Log(LevelGrid.Instance.GetGridPosition(MouseWorld.GetMouseWorldPosition()));
        // Prevents the player from performing multiple actions at the same time
        if (isBusy) return;
        // if is not the player's turn, ignore input
        if (!TurnSystem.Instance.IsPlayerTurn()) return;

        // Ignore input if the mouse is over a UI element
        if (EventSystem.current.IsPointerOverGameObject()) return;

        // Check if the player is trying to select a unit or move the selected unit
        if (TryHandleUnitSelection()) return;

        HandleSelectedAction();
    }

    private void HandleSelectedAction()
    {
        // Jos ei ole valittua yksikköä. Ei edes yritetä tehdä mitään.
        if (selectedUnit == null || selectedAction == null) return;

        if (InputManager.Instance.IsMouseButtonDownThisFrame())
        {
            GridPosition mouseGridPosition = LevelGrid.Instance.GetGridPosition(MouseWorld.GetPositionOnlyHitVisible());

            // Ei yritetä tehdä niitä toimintoja johon valittun Unitin liike ei riitä
            int steps = selectedUnit.GetMaxMoveDistance();
            int moveBudgetCost = PathFinding.CostFromSteps(steps);
            int estCost = PathFinding.Instance.CalculateDistance(selectedUnit.GetGridPosition(),
            mouseGridPosition);
            if (estCost > moveBudgetCost * 10) return;

            if (!selectedAction.IsValidGridPosition(mouseGridPosition)||
            !selectedUnit.TrySpendActionPointsToTakeAction(selectedAction)) return;

            SetBusy();
            selectedAction.TakeAction(mouseGridPosition, ClearBusy);
            OnActionStarted?.Invoke(this, EventArgs.Empty);
        }
    }

    /// <summary>
    //      Prevents the player from performing multiple actions at the same time
    /// </summary>
    private void SetBusy()
    {
        isBusy = true;
        OnBusyChanged?.Invoke(this, isBusy);
    }

    /// <summary>
    ///     This method is called when the action is completed.
    /// </summary>
    private void ClearBusy()
    {
        isBusy = false;
        OnBusyChanged?.Invoke(this, isBusy);
    }

    /// <summary>
    ///     This method is called when the player clicks on a unit in the game world.
    ///     Check if the mouse is over a unit
    ///     If so, select the unit and return
    ///     If not, move the selected unit to the mouse position
    /// </summary>
    private bool TryHandleUnitSelection()
    {
        if (InputManager.Instance.IsMouseButtonDownThisFrame())
        {
            Ray ray = Camera.main.ScreenPointToRay(InputManager.Instance.GetMouseScreenPosition());
            if (Physics.Raycast(ray, out RaycastHit hit, float.MaxValue, unitLayerMask))
            {
                if (hit.transform.TryGetComponent<Unit>(out Unit unit))
                {
                    if (AuthorityHelper.HasLocalControl(unit) || unit == selectedUnit) return false;
                    SetSelectedUnit(unit);
                    return true;
                }
            }
        }

        return false;
    }

    /// <summary>
    ///     Sets the selected unit and triggers the OnSelectedUnitChanged event.
    ///     By defaults set the selected action to the unit's move action. The most common action.
    /// </summary>
    private void SetSelectedUnit(Unit unit)
    {
        if (unit.IsEnemy()) return;
        selectedUnit = unit;
     //   SetSelectedAction(unit.GetMoveAction());
        SetSelectedAction(unit.GetAction<MoveAction>());
        OnSelectedUnitChanged?.Invoke(this, EventArgs.Empty);
    }

    /// <summary>
    ///     Sets the selected action and triggers the OnSelectedActionChanged event.
    ///  </summary>
    public void SetSelectedAction(BaseAction baseAction)
    {
        selectedAction = baseAction;
        OnSelectedActionChanged?.Invoke(this, EventArgs.Empty);
    }

    public Unit GetSelectedUnit()
    {
        return selectedUnit;
    }

    public BaseAction GetSelectedAction()
    {
        return selectedAction;
    }

    // Lock/Unlock input methods for PlayerController when playing online
    public void LockInput() { if (!isBusy) SetBusy(); }
    public void UnlockInput() { if (isBusy)  ClearBusy(); }
}
```

Assets/scripts/Units/UnitAnimator.cs

```
using UnityEngine;
using System;
using Mirror;

[RequireComponent(typeof(MoveAction))]
public class UnitAnimator : NetworkBehaviour
{
    [Header("UnitWeaponVisibilitySync")]
    [SerializeField] private WeaponVisibilitySync weaponVis;

    [Header("Animators")]
    [SerializeField] private Animator animator;
    [SerializeField] private NetworkAnimator netAnim;

    [Header("Projectiles")]
    [SerializeField] private GameObject bulletProjectilePrefab;
    [SerializeField] private GameObject granadeProjectilePrefab;

    [Header("Spawnpoints")]
    [SerializeField] private Transform shootPointTransform;
    [SerializeField] private Transform rightHandTransform;

    private static bool IsNetworkActive() => NetworkClient.active || NetworkServer.active;

    private void Awake()
    {
        if (TryGetComponent<MoveAction>(out MoveAction moveAction))
        {
            moveAction.OnStartMoving += MoveAction_OnStartMoving;
            moveAction.OnStopMoving += MoveAction_OnStopMoving;
        }

        if (TryGetComponent<ShootAction>(out ShootAction shootAction))
        {
            shootAction.OnShoot += ShootAction_OnShoot;
        }

        if (TryGetComponent<GranadeAction>(out GranadeAction granadeAction))
        {
            granadeAction.ThrowGranade += GrenadeAction_ThrowGranade;
            granadeAction.ThrowReady += GrenadeAction_ThrowReady;
        }

        if (TryGetComponent<MeleeAction>(out MeleeAction meleeAction))
        {
            meleeAction.OnMeleeActionStarted += MeleeAction_OnMeleeActionStarted;
            meleeAction.OnMeleeActionCompleted += MeleeAction_OnMeleeActionCompleted;
        }
    }
    private void Start()
    {
        EquipRifle();
    }

    void OnDisable()
    {
        if (TryGetComponent<MoveAction>(out MoveAction moveAction))
        {
            moveAction.OnStartMoving -= MoveAction_OnStartMoving;
            moveAction.OnStopMoving -= MoveAction_OnStopMoving;
        }

        if (TryGetComponent<ShootAction>(out ShootAction shootAction))
        {
            shootAction.OnShoot -= ShootAction_OnShoot;
        }

        if (TryGetComponent<GranadeAction>(out GranadeAction granadeAction))
        {
            granadeAction.ThrowGranade -= GrenadeAction_ThrowGranade;
            granadeAction.ThrowReady -= GrenadeAction_ThrowReady;
        }

        if (TryGetComponent<MeleeAction>(out MeleeAction meleeAction))
        {
            meleeAction.OnMeleeActionStarted -= MeleeAction_OnMeleeActionStarted;
            meleeAction.OnMeleeActionCompleted -= MeleeAction_OnMeleeActionCompleted;
        }
    }

    private void MoveAction_OnStartMoving(object sender, EventArgs e)
    {
        animator.SetBool("IsRunning", true);
    }
    private void MoveAction_OnStopMoving(object sender, EventArgs e)
    {
        animator.SetBool("IsRunning", false);
    }

    private void ShootAction_OnShoot(object sender, ShootAction.OnShootEventArgs e)
    {
        if (!IsNetworkActive())
        {
            animator.SetTrigger("Shoot");
        }
        else
        {
            netAnim.SetTrigger("Shoot");
        }

        Vector3 target = e.targetUnit.GetWorldPosition();
        float unitShoulderHeight = 2.5f;
        target.y += unitShoulderHeight;
        NetworkSync.SpawnBullet(bulletProjectilePrefab, shootPointTransform.position, target);
    }

    private void MeleeAction_OnMeleeActionStarted(object sender, EventArgs e)
    {
        EquipMelee();
        if (!IsNetworkActive())
        {
            animator.SetTrigger("Melee");
        }
        else
        {
            netAnim.SetTrigger("Melee");
        }
    }
    private void MeleeAction_OnMeleeActionCompleted(object sender, EventArgs e)
    {
        EquipRifle();
    }

    private void GranadeActionStart()
    {
        weaponVis.OwnerRequestSet(rifleRight: false, rifleLeft: true, meleeLeft: false, grenade: false);
    }
    private Vector3 pendingGrenadeTarget;
    private GranadeAction pendingGrenadeAction;
    private void GrenadeAction_ThrowGranade(object sender, EventArgs e)
    {
        pendingGrenadeAction = (GranadeAction)sender;
        pendingGrenadeTarget = pendingGrenadeAction.TargetWorld;
        GranadeActionStart();
        if (!IsNetworkActive())
        {
            animator.SetTrigger("ThrowGrenade");
        }
        else
        {
            netAnim.SetTrigger("ThrowGrenade");
        }
    }

    // --------- START Grenade Animation events START -----------------------
    // Event marks is set in animation. UnitAnimations -> Throw Grenade Stand
    public void AE_PickGrenadeStand()
    {
        EguipGranade();
    }
    public void AE_ThrowGrenadeStandRelease()
    {
        // --- GUARD: jos pending on jo käytetty, älä tee mitään (estää tuplan samalta koneelta)
        if (pendingGrenadeAction == null) return;

        // --- GATE: onlinessa vain omistaja-client saa jatkaa (server ja ei-ownerit return)
        if (NetworkClient.active || NetworkServer.active)
        {
            var ni = GetComponentInParent<NetworkIdentity>();
            if (!(isClient && ni && ni.isOwned)) return;
        }

        // Mistä kranaatti lähtee (sama logiikka kuin luodeilla)
        Vector3 origin = rightHandTransform.position;

        // Kutsu keskitettyä synkkaa (täsmälleen kuin luodeissa)
        NetworkSync.SpawnGrenade(granadeProjectilePrefab, origin, pendingGrenadeTarget);

        // Siivous kuten ennen
        pendingGrenadeAction?.OnGrenadeBehaviourComplete();
        pendingGrenadeAction = null;
    }
    public void AE_OnGrenadeThrowStandFinished()
    {
        EquipRifle();
    }
    //--------------- END Grenade Animation events END ---------------
    private void GrenadeAction_ThrowReady(object sender, EventArgs e)
    {
       weaponVis.OwnerRequestSet(rifleRight: false, rifleLeft: true, meleeLeft: false, grenade: false);
    }

    private void EquipRifle()
    {
        weaponVis.OwnerRequestSet(rifleRight: true, rifleLeft: false, meleeLeft: false, grenade: false);
    }
    private void EquipMelee()
    {
        weaponVis.OwnerRequestSet(rifleRight: true, rifleLeft: false, meleeLeft: true, grenade: false);
    }
    private void EguipGranade()
    {
        weaponVis.OwnerRequestSet(rifleRight: false, rifleLeft: true, meleeLeft: false, grenade: true);
    }
}
```

Assets/scripts/Units/UnitAnimatorEventRelay.cs

```
using UnityEngine;

/// <summary>
/// This is needed so that animation event-bound functions in UnitAnimator can be used. Such as AE_Throw Grenade Stand Release()
/// </summary>
public class AnimationEventRelay : MonoBehaviour
{
    [SerializeField] private UnitAnimator unitAnimator;

    void Awake()
    {
        // Etsi parentista jos ei asetettu Inspectorissa
        if (!unitAnimator) unitAnimator = GetComponentInParent<UnitAnimator>();
    }

    // Täsmälleen sama nimi kuin Animation Eventin Function-kentässä
    public void AE_ThrowGrenadeStandRelease()
    {
        unitAnimator?.AE_ThrowGrenadeStandRelease();
    }

    public void AE_PickGrenadeStand()
    {
        unitAnimator?.AE_PickGrenadeStand();
    }
}
```

Assets/scripts/Units/UnitManager.cs

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

/// <summary>
/// This class is responsible for managing all units in the game.
/// It keeps track of all units, friendly units, and enemy units.
/// It listens to unit spawn and death events to update its lists accordingly.
/// Note: This class Script Script Execution Order is set to be executed after UnitActionSystem.cs. High priority.
/// </summary>
public class UnitManager : MonoBehaviour
{
    public static UnitManager Instance { get; private set; }
    private List<Unit> unitList;
    private List<Unit> friendlyUnitList;
    private List<Unit> enemyUnitList;

    private void Awake()
    {
        if (Instance != null)
        {
            Debug.LogError("There's more than one UnitManager! " + transform + " - " + Instance);
            Destroy(gameObject);
            return;
        }
        Instance = this;

        unitList = new List<Unit>();
        friendlyUnitList = new List<Unit>();
        enemyUnitList = new List<Unit>();
    }

    private void Start()
    {
        Unit.OnAnyUnitSpawned += Unit_OnAnyUnitSpawned;
        Unit.OnAnyUnitDead += Unit_OnAnyUnitDead;
    }

    void OnEnable()
    {
        Unit.OnAnyUnitSpawned += Unit_OnAnyUnitSpawned;
        Unit.OnAnyUnitDead += Unit_OnAnyUnitDead;
    }

    void OnDisable()
    {
        Unit.OnAnyUnitSpawned -= Unit_OnAnyUnitSpawned;
        Unit.OnAnyUnitDead -= Unit_OnAnyUnitDead;
    }
    private void Unit_OnAnyUnitSpawned(object sender, EventArgs e)
    {

        Unit unit = sender as Unit;
        unitList.Add(unit);

        if (unit.IsEnemy())
        {
            enemyUnitList.Add(unit);
        }
        else
        {
            friendlyUnitList.Add(unit);
        }
    }

    private void Unit_OnAnyUnitDead(object sender, EventArgs e)
    {
        Unit unit = sender as Unit;
        unitList.Remove(unit);

        if (unit.IsEnemy())
        {
            enemyUnitList.Remove(unit);
        }
        else
        {
            friendlyUnitList.Remove(unit);
        }
    }

    public List<Unit> GetUnitList()
    {
        return unitList;
    }

    public List<Unit> GetFriendlyUnitList()
    {
        return friendlyUnitList;
    }

    public List<Unit> GetEnemyUnitList()
    {
        return enemyUnitList;
    }

    public void ClearAllUnitLists()
    {
        unitList.Clear();
        friendlyUnitList.Clear();
        enemyUnitList.Clear();
    }
}
```

Assets/scripts/Units/UnitPathFinding/EdgeBaker.cs

```
using UnityEngine;

[DefaultExecutionOrder(500)] // After Pathfindingin
[DisallowMultipleComponent]

/// @file EdgeBaker.cs
/// @brief Edge-based obstacle detection and wall baking system for RogueShooter.
///
/// The EdgeBaker scans the environment to detect narrow obstacles (walls, fences, railings, doorframes)
/// between adjacent grid cells and encodes them as edge-wall flags in the pathfinding data.
/// This ensures that unit movement and line-of-sight calculations align precisely with physical geometry.
///
/// ### Overview
/// EdgeBaker operates immediately after walkability baking has been performed by the `PathFinding` system.
/// It iterates through all walkable cells and performs four narrow physics checks (north, east, south, west)
/// to detect thin colliders lying between grid borders. Any detected obstacle is stored as an `EdgeMask`
/// flag on both affected nodes to maintain symmetric connectivity.
///
/// ### System integration
/// - **LevelGrid** – Provides spatial dimensions and world↔grid coordinate mapping for each cell.
/// - **PathFinding** – Supplies the `PathNode` data structure where edge walls are stored and queried.
/// - **EdgeBaker** – Bridges the physical Unity scene and the logical pathfinding layer by detecting edge blockers.
///
/// ### Key features
/// - Detects fine-grained edge blockers that are smaller than a full grid cell.
/// - Writes edge-wall data symmetrically to adjacent nodes (no “one-way walls”).
/// - Supports incremental rebaking after runtime geometry changes (doors opening, walls destroyed).
/// - Uses Physics.CheckBox for reliable thin-edge detection with adjustable thickness and scan height.
/// - Operates deterministically and independently of Unity’s NavMesh system.
///
/// ### Why this exists in RogueShooter
/// - The game’s tactical combat requires accurate cover and movement restrictions based on geometry.
/// - Standard per-cell walkability alone cannot capture small barriers or partial walls.
/// - This system creates a precise “micro-collision” layer between cells, allowing units to interact
///   with the environment in a realistic and strategically meaningful way.
///
/// In summary, this file defines the edge-detection system that enhances the grid-based pathfinding
/// with sub-cell precision, ensuring that RogueShooter’s movement, visibility, and cover mechanics
/// reflect the actual physical layout of each combat environment.

/// <summary>
/// Automatically detects and marks impassable edges between walkable grid cells,
/// based on physical obstacles present in the scene (walls, fences, railings, doorframes, etc.).
///
/// This component “bakes” thin collision lines along cell borders using Physics.CheckBox tests,
/// writing wall data directly into the PathFinding grid nodes (via EdgeMask flags).
/// It ensures that movement and line-of-sight calculations align with the actual environment geometry.
///
/// Design notes specific to RogueShooter:
/// - Used right after walkability baking to identify fine-grained obstacles between adjacent cells.
/// - Prevents units from moving or shooting through narrow environmental blockers
///   that don't occupy a full cell (e.g., half-walls, railings, or destroyed doorframes).
/// - Enables more realistic tactical cover and movement logic without relying on Unity's full NavMesh system.
/// - Automatically rebakes affected areas when dynamic obstacles (like doors or destructible walls) change state.
/// </summary>
public class EdgeBaker : MonoBehaviour
{
    public static EdgeBaker Instance { get; private set; }

    [Header("References")]
    [SerializeField] private PathFinding pathfinding;   // Jos jätät tyhjäksi, etsitään automaattisesti
    [SerializeField] private LevelGrid levelGrid;       // Jos jätät tyhjäksi, käytetään LevelGrid.Instance

    [Header("When to run")]
    [SerializeField] private bool autoBakeOnStart = true;

    [Header("Edge scan")]
    [Tooltip("Layerit, jotka edustavat RUUTUJEN VÄLISIÄ, ohuita liikkumista estäviä juttuja (kaiteet, seinäviivat, ovenpielet, tms.)")]
    [SerializeField] private LayerMask edgeBlockerMask;

    [Tooltip("Reunan skannauksen 'nauhan' paksuus suhteessa cellSizeen (0.05-0.2 on tyypillinen).")]
    [Range(0.01f, 0.5f)]
    [SerializeField] private float edgeStripThickness = 0.1f;

    [Tooltip("Kuinka korkealta skannataan (metreinä). Yleensä hieman ukkelin pään korkeuden yläpuolelle.")]
    [SerializeField] private float edgeScanHeight = 2.0f;

    // ---- Lyhyet aliasit, ettei tarvitse arvailla mistä mikäkin tulee ----
    private PathFinding PF => pathfinding != null ? pathfinding : (pathfinding = FindFirstObjectByType<PathFinding>());
    private LevelGrid LG => levelGrid != null ? levelGrid : (levelGrid = LevelGrid.Instance);

    private int Width;
    private int Height;
    private int FloorAmount;
    private float CellSize;

    private void Awake()
    {
        if (Instance != null && Instance != this) { Destroy(gameObject); return; }
        Instance = this;

        if (pathfinding == null) pathfinding = FindFirstObjectByType<PathFinding>();
        if (levelGrid == null) levelGrid = LevelGrid.Instance;

        Width = levelGrid.GetWidth();
        Height = levelGrid.GetHeight();
        FloorAmount = levelGrid.GetFloorAmount();
        CellSize = levelGrid.GetCellSize();


    }
    private void Start()
    {
        if (autoBakeOnStart) BakeAllEdges();
    }

    // ------------------------- PUBLIC API -------------------------

    /// <summary>
    /// Performs a full edge bake across the entire grid.
    ///
    /// Clears all previously marked walls, then scans every walkable cell
    /// in all floors to detect thin obstacles (edges) between neighboring cells.
    ///
    /// Design notes specific to RogueShooter:
    /// - This is typically called once at level initialization, right after walkability checks.
    /// - It ensures that all cell borders reflect real physical blockers,
    ///   so units cannot move or shoot through walls, fences, or other narrow obstacles.
    /// - Provides the foundation for accurate tactical pathfinding and cover detection.
    /// </summary>
    public void BakeAllEdges()
    {
        if (!Preflight()) return;

        // 1) Clear all existing wall data from every node in every floor
        for (int f = 0; f < FloorAmount; f++)
            for (int x = 0; x < Width; x++)
                for (int z = 0; z < Height; z++)
                {
                    var node = PF.GetNode(x, z, f);
                    if (node != null) node.ClearWalls();
                }

        // 2) Scan each walkable cell and bake its N/E/S/W edge data
        for (int f = 0; f < FloorAmount; f++)
            for (int x = 0; x < Width; x++)
                for (int z = 0; z < Height; z++)
                {
                    var gp = new GridPosition(x, z, f);
                    if (!IsWalkable(gp)) continue;

                    BakeEdgesForCell(gp);
                }
    }

    /// <summary>
    /// Rebuilds edge data locally around a given grid position.
    ///
    /// Used when the environment changes dynamically — for example,
    /// when a door opens or closes, or when a wall is destroyed.
    /// This function rescans a small area instead of rebaking the entire map,
    /// keeping pathfinding and cover data up to date with minimal performance cost.
    ///
    /// Design notes specific to RogueShooter:
    /// - Ensures that tactical movement and line-of-sight stay accurate
    ///   after real-time map changes during combat.
    /// - Called automatically by interactive elements like doors or destructible props.
    /// </summary>
    public void RebakeEdgesAround(GridPosition center, int radius = 1)
    {
        if (!Preflight()) return;

        // Loop through a square area centered on the target grid position
        for (int dx = -radius; dx <= radius; dx++)
            for (int dz = -radius; dz <= radius; dz++)
            {
                var gp = new GridPosition(center.x + dx, center.z + dz, center.floor);
                if (!IsValidGridPosition(gp)) continue;

                var node = PF.GetNode(gp.x, gp.z, gp.floor);
                if (node == null) continue;

                // 1) Clear old wall data
                node.ClearWalls();

                // 2) Rescan and rebuild edge data for this cell
                BakeEdgesForCell(gp);
            }
    }

    // ------------------------- CORE -------------------------
    /// <summary>
    /// Scans the four borders (N/E/S/W) of a single walkable grid cell and writes edge-wall flags.
    ///
    /// What it does:
    /// - Builds four thin, axis-aligned 3D “strips” (AABBs) that sit exactly on the cell borders.
    /// - Uses Physics.CheckBox to detect narrow blockers (rails, thin walls, door frames) at a chosen height.
    /// - For every detected blocker, sets the matching EdgeMask flag on the current node
    ///   and mirrors the opposite flag on the neighboring node to keep graph connectivity symmetric.
    ///
    /// Why this exists in RogueShooter:
    /// - Our levels contain many obstacles that do NOT fill the whole cell but still block movement/LOS across an edge.
    /// - Baking per-edge blockers yields more faithful tactical movement and cover behavior than cell-only walkability.
    /// - Keeping the data symmetric (both sides of the shared edge agree) avoids pathfinding inconsistencies.
    ///
    /// Implementation notes:
    /// - Each cell does a constant amount of physics work (4 × Physics.CheckBox).
    /// - The strip thickness is a fraction of the cell size (edgeStripThickness), tuned to “catch” thin geometry
    ///   without overlapping neighboring interiors.
    /// - The scan runs at edgeScanHeight (centered at Y = edgeScanHeight * 0.5), typically around head-height,
    ///   so low floor clutter doesn’t cause false positives while walls/rails are still detected.
    /// </summary>
    private void BakeEdgesForCell(GridPosition gp)
    {
        // World-space center of this cell (at floor level)
        Vector3 center = LG.GetWorldPosition(gp);
        float s = CellSize;

        // Define half-extents for the thin scanning strips:
        // - North/South strips are long along Z, thin along X.
        // - East/West strips are long along X, thin along Z.
        // Height half-extent is half of edgeScanHeight (so total box height == edgeScanHeight).
        Vector3 halfNorthSouth = new(s * edgeStripThickness * 0.5f, edgeScanHeight * 0.5f, s * 0.45f);
        Vector3 halfEastWest = new(s * 0.45f, edgeScanHeight * 0.5f, s * edgeStripThickness * 0.5f);

        // Place the four strip centers exactly on the cell borders and lift to mid-scan height.
        float y = edgeScanHeight * 0.5f;
        Vector3 north = center + new Vector3(0f, y, +s * 0.5f);
        Vector3 south = center + new Vector3(0f, y, -s * 0.5f);
        Vector3 east = center + new Vector3(+s * 0.5f, y, 0f);
        Vector3 west = center + new Vector3(-s * 0.5f, y, 0f);

        var node = PF.GetNode(gp.x, gp.z, gp.floor);

        // Probe NORTH edge; if blocked, mark N on this node and S on the northern neighbor.
        if (HasEdgeBlock(north, halfNorthSouth, Quaternion.identity))
        {
            node.AddWall(EdgeMask.N);
            MarkOpposite(gp, +0, +1, EdgeMask.S);
        }
        // Probe SOUTH edge; mirror to the southern neighbor.
        if (HasEdgeBlock(south, halfNorthSouth, Quaternion.identity))
        {
            node.AddWall(EdgeMask.S);
            MarkOpposite(gp, +0, -1, EdgeMask.N);
        }
        // Probe EAST edge; mirror to the eastern neighbor.
        if (HasEdgeBlock(east, halfEastWest, Quaternion.identity))
        {
            node.AddWall(EdgeMask.E);
            MarkOpposite(gp, +1, +0, EdgeMask.W);
        }
        // Probe WEST edge; mirror to the western neighbor.
        if (HasEdgeBlock(west, halfEastWest, Quaternion.identity))
        {
            node.AddWall(EdgeMask.W);
            MarkOpposite(gp, -1, +0, EdgeMask.E);
        }
    }

    /// <summary>
    /// Checks whether a physical obstacle exists along a specific cell edge.
    ///
    /// Uses Physics.CheckBox with the configured <see cref="edgeBlockerMask"/> to detect
    /// any geometry that should prevent movement or line-of-sight across that border.
    ///
    /// Why this exists in RogueShooter:
```

```
    /// - We rely on thin colliders (walls, railings, doorframes) placed between grid cells.
    /// - Detecting those lets the pathfinding system respect scene geometry more accurately
    ///   than simple per-cell walkability checks.
    /// - Called four times per cell (once for each direction) during edge baking.
    ///
    /// Implementation notes:
    /// - Returns true if *any* collider in the given layer mask overlaps the test volume.
    /// - QueryTriggerInteraction.Ignore avoids false positives from trigger colliders.
    /// </summary>
    private bool HasEdgeBlock(Vector3 center, Vector3 halfExtents, Quaternion rot)
    {
        return Physics.CheckBox(center, halfExtents, rot, edgeBlockerMask, QueryTriggerInteraction.Ignore);
    }

    /// <summary>
    /// Mirrors an edge-wall flag to the neighboring grid cell so both sides of the shared border agree.
    ///
    /// What it does:
    /// - Computes the neighbor position by offset (dx, dz) on the same floor.
    /// - If the neighbor node exists, adds the opposite direction wall flag to it.
    ///
    /// Why this exists in RogueShooter:
    /// - Keeps pathfinding data consistent between adjacent nodes.
    /// - Prevents “one-way walls,” where one node thinks the edge is blocked
    ///   but its neighbor does not — a common cause of desyncs in tactical grids.
    ///
    /// Implementation notes:
    /// - This method assumes edge baking is done in grid order, so each pair
    ///   of adjacent cells will eventually synchronize their shared edge data.
    /// </summary>
    private void MarkOpposite(GridPosition a, int dx, int dz, EdgeMask oppositeDir)
    {
        var b = new GridPosition(a.x + dx, a.z + dz, a.floor);
        if (!IsValidGridPosition(b)) return;

        var nb = PF.GetNode(b.x, b.z, b.floor);
        if (nb == null) return;

        // Add the mirrored wall flag to the neighbor node
        nb.AddWall(oppositeDir);
    }

    // ------------------------- HELPERS -------------------------
    /// <summary>
    /// Performs a quick validation before baking begins.
    ///
    /// Checks that references to <see cref="PathFinding"/> and <see cref="LevelGrid"/> are valid,
    /// either through serialized fields or automatic runtime lookup.
    ///
    /// Why this exists in RogueShooter:
    /// - Prevents null-reference errors during scene startup.
    /// - Ensures that the grid and pathfinding systems are fully initialized
    ///   before attempting any edge scanning or node modification.
    ///
    /// Implementation notes:
    /// - Logs descriptive errors to help diagnose missing scene references.
    /// - Returns false if any critical dependency is missing, stopping the bake safely.
    /// </summary>
    private bool Preflight()
    {
        if (PF == null)
        {
            Debug.LogError("[EdgeBaker] Pathfinding reference missing (and not found automatically).");
            return false;
        }
        if (LG == null)
        {
            Debug.LogError("[EdgeBaker] LevelGrid reference missing (and not found automatically).");
            return false;
        }
        return true;
    }

    /// <summary>
    /// Determines whether the specified grid position corresponds to a walkable node.
    ///
    /// Why this exists in RogueShooter:
    /// - Edge baking should only occur on cells that units can actually occupy.
    /// - Avoids unnecessary physics checks for blocked or void cells (improves performance).
    ///
    /// Implementation notes:
    /// - Fetches the node from PathFinding and queries its <c>GetIsWalkable()</c> flag.
    /// </summary>
    private bool IsWalkable(GridPosition gp)
    {
        var node = PF.GetNode(gp.x, gp.z, gp.floor);
        return node != null && node.GetIsWalkable();
    }

    /// <summary>
    /// Validates that a given grid position exists within the bounds of the level grid.
    ///
    /// Why this exists in RogueShooter:
    /// - Edge baking frequently queries neighboring cells (±1 in X/Z).
    /// - Ensures that no out-of-range indices are accessed, preventing runtime errors.
    ///
    /// Implementation notes:
    /// - Uses LevelGrid’s built-in <c>IsValidGridPosition()</c> if available for the current floor.
    /// - Falls back to manual bounds checking if no grid system reference is found.
    /// </summary>
    private bool IsValidGridPosition(GridPosition gp)
    {
        var gridSystem = LG.GetGridSystem(gp.floor);
        if (gridSystem != null) return gridSystem.IsValidGridPosition(gp);
        return gp.x >= 0 && gp.z >= 0 && gp.x < Width && gp.z < Height && gp.floor >= 0 && gp.floor < FloorAmount;
    }
}
```

Assets/scripts/Units/UnitPathFinding/PathFinding.cs

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

/// @file PathFinding.cs
/// @brief Core pathfinding system for RogueShooter.
///
/// This component implements the game’s grid-based navigation logic using a custom A* algorithm
/// with full support for multi-floor environments, movement budgets, and edge-based wall detection.
///
/// ### Overview
/// The pathfinding system converts Unity scene geometry into an abstract tactical grid used
/// by both player and AI units. Each cell is represented by a `PathNode` containing walkability,
/// cost, and edge-wall information. The system supports 8-directional movement (N, NE, E, SE, S, SW, W, NW)
/// and dynamically links multiple floors through designer-placed `PathfindingLink` components.
///
/// ### System integration
/// - **LevelGrid** – Defines grid dimensions and provides world↔grid coordinate conversions.
/// - **EdgeBaker** – Scans scene colliders to detect thin obstacles between cells and marks walls accordingly.
/// - **PathFinding** – Performs A* searches using the processed node and edge data.
///
/// ### Key features
/// - Fully deterministic and allocation-free per search (generation-ID based node reuse).
/// - Accurate obstacle handling using edge blockers (no corner clipping or one-way walls).
/// - Move-budget based path truncation for tactical range queries and AI planning.
/// - Extensible multi-floor connectivity via `PathfindingLink` objects.
/// - Optional runtime diagnostics through `PathfindingDiagnostics` (profiling search times and expansions).
///
/// ### Why this exists in RogueShooter
/// - The game’s tactical, turn-based design requires predictable and grid-aligned movement.
/// - Unity’s built-in NavMesh system is unsuitable for deterministic tile-based combat logic.
/// - Custom A* implementation allows tight integration with game-specific mechanics such as
///   cover, destructible walls, and limited-range actions.
///
/// In summary, this file defines the core pathfinding logic that powers all unit movement
/// and AI navigation in RogueShooter, ensuring consistency between physical scene geometry
/// and tactical gameplay rules.

/// <summary>
/// Grid-based A* pathfinding for 8-directional movement (N, NE, E, SE, S, SW, W, NW) across multiple floors.
///
/// What it does:
/// - Builds and queries a per-floor grid of PathNodes and computes shortest paths using A* with an octile heuristic.
/// - Respects fine-grained edge blockers (walls/rails/doorframes) baked by <see cref="EdgeBaker"/> so units can’t
///   cut corners or move/shoot through narrow obstacles.
/// - Supports optional move budgets (in “steps”) for tactical range queries and AI decisions.
/// - Supports explicit inter-cell “links” (stairs/elevators/hatches) that connect arbitrary cells and floors.
///
/// Why this exists in RogueShooter:
/// - The game is turn-based and tile-based; we need deterministic, frame-stable paths that match tactical rules,
///   not freeform NavMesh paths.
/// - Edge-aware movement prevents diagonal corner-cutting and enforces cover/door behavior consistent with combat.
/// - Budgeted pathfinding enables fast “reachable area” calculations for UI previews and AI planning.
///
/// Design notes:
/// - Uses a lightweight custom PriorityQueue and generation IDs to avoid per-search allocations and stale scores.
/// - Movement costs: straight = 10, diagonal = 20 (octile distance for heuristic and step costs).
/// - Runs after <see cref="LevelGrid"/> initialization; floor walkability is raycasted once, edges baked next,
///   then A* queries can safely rely on up-to-date node/edge data.
/// - Optional debug visualizations can create grid debug objects for inspection in the editor.
/// </summary>
public class PathFinding : MonoBehaviour
{
    public static PathFinding Instance { get; private set; }

    private const int MOVE_STRAIGHT_COST = 10;
    private const int MOVE_DIAGONAL_COST = 20;

    [Header("Debug")]
    [SerializeField] private bool showDebug = false;
    [SerializeField] private Transform gridDebugPrefab;

    [Header("Layers")]
    [SerializeField] private LayerMask obstaclesLayerMask;
    [SerializeField] private LayerMask floorLayerMask;

    [Header("Links")]
    [SerializeField] private Transform pathfindingLinkContainer;

    private int width;
    private int height;
    private int currentGenerationID = 0;

    private List<GridSystem<PathNode>> gridSystemList;
    private List<PathfindingLink> pathfindingLinkList;

    private void Awake()
    {
        if (Instance != null)
        {
            Debug.LogError("PathFinding: More than one PathFinding in the scene! " + transform + " - " + Instance);
            Destroy(gameObject);
            return;
        }
        Instance = this;
    }

    /// <summary>
    /// Initializes the pathfinding system and builds all per-floor grid data.
    ///
    /// What it does:
    /// - Creates a <see cref="GridSystem{PathNode}"/> for each floor with the given dimensions.
    /// - Performs raycast-based walkability detection for every grid cell using floor and obstacle layers.
    /// - Invokes <see cref="EdgeBaker"/> to detect thin edge blockers between walkable cells.
    /// - Collects any explicit <see cref="PathfindingLink"/> connections (stairs, elevators, etc.) from the scene.
    ///
    /// Why this exists in RogueShooter:
    /// - Converts the 3D scene geometry into a grid-based navigation map used by all AI and tactical systems.
    /// - Ensures that units move on valid walkable surfaces and respect real physical barriers.
    /// - Keeps the runtime logic deterministic and self-contained without relying on Unity’s NavMesh.
    ///
    /// Implementation notes:
    /// - Should be called once during level initialization (by LevelGrid or GameManager).
    /// - Automatically performs full edge baking after walkability setup.
    /// - Uses layer masks for flexibility: <c>floorLayerMask</c> defines valid surfaces, <c>obstaclesLayerMask</c> blocks them.
    /// </summary>
    public void Setup(int width, int height, float cellSize, int floorAmount)
    {
        this.width = width;
        this.height = height;

        gridSystemList = new List<GridSystem<PathNode>>();

        // 1) Create one grid per floor
        for (int floor = 0; floor < floorAmount; floor++)
        {
            GridSystem<PathNode> gridSystem = new GridSystem<PathNode>(
                width, height, cellSize, floor, LevelGrid.FLOOR_HEIGHT,
                (GridSystem<PathNode> g, GridPosition gridPosition) => new PathNode(gridPosition)
            );

            // Optional: visualize grid in editor for debugging
            if (showDebug && gridDebugPrefab != null)
            {
                gridSystem.CreateDebugObjects(gridDebugPrefab);
            }

            gridSystemList.Add(gridSystem);
        }

        // 2) Raycast: determine which cells are walkable or blocked
        float raycastOffsetDistance = 1f;
        float raycastDistance = raycastOffsetDistance * 2f;

        for (int x = 0; x < width; x++)
        {
            for (int z = 0; z < height; z++)
            {
                for (int floor = 0; floor < floorAmount; floor++)
                {
                    GridPosition gridPosition = new GridPosition(x, z, floor);
                    Vector3 worldPosition = LevelGrid.Instance.GetWorldPosition(gridPosition);

                    // Default to non-walkable
                    GetNode(x, z, floor).SetIsWalkable(false);

                    // Downward ray: detect if a valid floor exists under this cell
                    if (Physics.Raycast(
                            worldPosition + Vector3.up * raycastOffsetDistance,
                            Vector3.down,
                            raycastDistance,
                            floorLayerMask))
                    {
                        GetNode(x, z, floor).SetIsWalkable(true);
                    }

                    // Upward ray: short check for obstacles blocking this space
                    if (Physics.Raycast(
                            worldPosition + Vector3.down * raycastOffsetDistance,
                            Vector3.up,
                            raycastDistance,
                            obstaclesLayerMask))
                    {
                        GetNode(x, z, floor).SetIsWalkable(false);
                    }
                }
            }
        }

        // 3) Bake edges between cells (walls, rails, etc.)
        EdgeBaker.Instance.BakeAllEdges();

        // 4) Gather explicit pathfinding links (stairs, lifts, portals)
        pathfindingLinkList = new List<PathfindingLink>();
        if (pathfindingLinkContainer != null)
        {
            foreach (Transform linkTf in pathfindingLinkContainer)
            {
                if (linkTf.TryGetComponent(out PathfindingLinkMonoBehaviour linkMb))
                {
                    pathfindingLinkList.Add(linkMb.GetPathfindingLink());
                }
            }
        }
    }

    /// <summary>
    /// Finds a path between two grid positions using the A* algorithm with an optional move budget.
    ///
    /// What it does:
    /// - Serves as the public entry point for pathfinding queries.
    /// - Wraps the internal implementation (<see cref="FindPathInternal"/>) while exposing a simpler interface.
    /// - Returns a list of grid positions representing the optimal route, or <c>null</c> if no valid path exists.
    ///
    /// Why this exists in RogueShooter:
    /// - Gameplay systems (player input, AI, ability targeting) request paths through this single method.
    /// - The move budget allows computing reachable tiles for tactical range previews (e.g. 6 steps max).
    ///
    /// Implementation notes:
    /// - <paramref name="moveBudgetSteps"/> can be set to <c>int.MaxValue</c> for unrestricted pathfinding.
    /// - Outputs <paramref name="pathLength"/> as total F-cost (movement cost + heuristic) of the found path.
    /// </summary>
    public List<GridPosition> FindPath(
        GridPosition startGridPosition,
        GridPosition endGridPosition,
        out int pathLeght,
        int moveBudgetSteps)
    {
        return FindPathInternal(startGridPosition, endGridPosition, out pathLeght, moveBudgetSteps);
    }

    /// <summary>
    /// Core A* pathfinding algorithm implementation with movement budget and edge-aware navigation.
    ///
    /// What it does:
    /// - Expands nodes using standard A* logic (G = actual cost, H = heuristic, F = G + H).
    /// - Honors per-edge blockers from <see cref="EdgeBaker"/> via <c>CanStep()</c>.
    /// - Supports a movement budget (in “steps”) to limit search range for tactical actions.
    /// - Uses a lightweight custom <see cref="PriorityQueue{T}"/> for open list management.
    ///
    /// Why this exists in RogueShooter:
    /// - Provides deterministic and efficient tactical pathfinding across destructible, multi-floor maps.
    /// - Integrates movement range rules directly into path expansion, avoiding separate “reachable area” passes.
    /// - Enables AI and player systems to share the same consistent grid and cost rules.
    ///
    /// Algorithm overview:
    /// 1. Convert <paramref name="moveBudgetSteps"/> into internal cost units (straight = 10, diagonal = 20).
    /// 2. Early reject if even the heuristic distance exceeds the available budget.
    /// 3. Initialize open and closed sets and enqueue the start node.
    /// 4. While the open queue is not empty:
    ///    - Dequeue the node with the lowest F-cost.
    ///    - If its G-cost exceeds the movement budget → skip.
    ///    - If this is the end node → reconstruct the path and return.
    ///    - Otherwise, expand all valid neighbors that are walkable and not blocked by edges.
    /// 5. Return <c>null</c> if no path exists within the allowed movement cost.
    ///
    /// Performance notes:
    /// - Avoids heap allocations via <see cref="EnsureInit"/> using generation IDs.
    /// - Supports optional runtime diagnostics through <see cref="PathfindingDiagnostics"/> (#if PERFORMANCE_DIAG).
    /// - Handles diagonal movement correctly with octile distances and no corner clipping.
    /// </summary>
    private List<GridPosition> FindPathInternal(
        GridPosition startGridPosition,
        GridPosition endGridPosition,
        out int pathLeght,
        int moveBudgetSteps)
    {
```

```
#if PERFORMANCE_DIAG

        var diag = PathfindingDiagnostics.Instance;
        bool diagOn = diag != null && diag.enabledRuntime;

        System.Diagnostics.Stopwatch sw = null;
        if (diagOn) { sw = new System.Diagnostics.Stopwatch(); sw.Start(); }

        int expanded = 0; // kuinka monta solmua laajennettiin (pop + käsitelty)

#endif
        // 1) Convert step-based budget to internal movement cost units
        int moveBudgetCost = (moveBudgetSteps == int.MaxValue)
            ? int.MaxValue
            : moveBudgetSteps * MOVE_STRAIGHT_COST;

        // Early pruning: skip search if even the heuristic distance exceeds the move budget
        int minPossibleCost = CalculateDistance(startGridPosition, endGridPosition);
        if (minPossibleCost > moveBudgetCost)
        {
            pathLeght = 0;

#if PERFORMANCE_DIAG

            if (diagOn) { sw.Stop(); diag.AddSample(sw.Elapsed.TotalMilliseconds, false, 0, expanded); }

#endif

            return null;
        }

        currentGenerationID++;

        var openQueue = new PriorityQueue<PathNode>();
        HashSet<PathNode> openSet = new HashSet<PathNode>();
        HashSet<PathNode> closedSet = new HashSet<PathNode>();

        PathNode startNode = GetGridSystem(startGridPosition.floor).GetGridObject(startGridPosition);
        PathNode endNode = GetGridSystem(endGridPosition.floor).GetGridObject(endGridPosition);

        // Initialize start node
        EnsureInit(startNode);
        startNode.SetGCost(0);
        startNode.SetHCost(CalculateDistance(startGridPosition, endGridPosition));
        startNode.CalculateFCost();

        openQueue.Enqueue(startNode, startNode.GetFCost());
        openSet.Add(startNode);

        // 2) Main A* loop
        while (openQueue.Count > 0)
        {
            // Dequeue the node with the lowest F-cost; skip outdated entries
            PathNode currentNode = openQueue.Dequeue();
            if (closedSet.Contains(currentNode)) continue;

            EnsureInit(currentNode);

#if PERFORMANCE_DIAG
            expanded++;
#endif
            // Stop expanding if the current path already exceeds move budget
            if (currentNode.GetGCost() > moveBudgetCost)
                continue;

            // Goal reached → build final path
            if (currentNode == endNode)
            {
                pathLeght = endNode.GetFCost();
                var path = CalculatePath(endNode);

#if PERFORMANCE_DIAG

                if (diagOn)
                {
                    sw.Stop();
                    diag.AddSample(sw.Elapsed.TotalMilliseconds, success: true, pathLen: path.Count, expanded: expanded);
                }
#endif
                return path;
            }

            openSet.Remove(currentNode);
            closedSet.Add(currentNode);

            // 3) Expand all valid neighbor nodes
            foreach (PathNode neighbourNode in GetNeighbourList(currentNode))
            {
                if (closedSet.Contains(neighbourNode)) continue;

                if (!neighbourNode.GetIsWalkable())
                {
                    closedSet.Add(neighbourNode);
                    continue;
                }

                EnsureInit(neighbourNode);

                int stepCost = CalculateDistance(currentNode.GetGridPosition(), neighbourNode.GetGridPosition());
                int tentativeG = currentNode.GetGCost() + stepCost;

                // Skip paths that already exceed movement budget
                if (tentativeG > moveBudgetCost)
                    continue;
                // If this route to the neighbor is cheaper, record it
                if (tentativeG < neighbourNode.GetGCost())
                {
                    neighbourNode.SetCameFromPathNode(currentNode);
                    neighbourNode.SetGCost(tentativeG);
                    neighbourNode.SetHCost(CalculateDistance(neighbourNode.GetGridPosition(), endGridPosition));
                    neighbourNode.CalculateFCost();

                    if (!openSet.Contains(neighbourNode))
                    {

                        openQueue.Enqueue(neighbourNode, neighbourNode.GetFCost());
                        openSet.Add(neighbourNode);
                    }
                    else
                    {
                        // No decrease-key in PriorityQueue → push duplicate, old entry ignored when dequeued
                        openQueue.Enqueue(neighbourNode, neighbourNode.GetFCost());
                    }
                }
            }
        }

        // 4) No valid path within move budget
        pathLeght = 0;

#if PERFORMANCE_DIAG

        if (diagOn)
        {
            sw.Stop();
            diag.AddSample(sw.Elapsed.TotalMilliseconds, success: false, pathLen: 0, expanded: expanded);
        }

#endif
        return null;
    }

    /// <summary>
    /// Octile-distance cost between two grid positions for 8-directional movement.
    ///
    /// What it does:
    /// - Computes the admissible A* heuristic and unit step costs using:
    ///   diagonal = min(|dx|, |dz|), straight = | |dx| - |dz| |.
    /// - Returns MOVE_DIAGONAL_COST * diagonal + MOVE_STRAIGHT_COST * straight.
    ///
    /// Why this exists in RogueShooter:
    /// - Matches our movement rules exactly (orthogonal and diagonal with different costs),
    ///   keeping A* both admissible and consistent (no overestimation).
    ///
    /// Implementation notes:
```

```
    /// - MOVE_STRAIGHT_COST = 10, MOVE_DIAGONAL_COST = 20 to align with budget-in-steps logic.
    /// </summary>
    public int CalculateDistance(GridPosition a, GridPosition b)
    {
        GridPosition d = a - b;
        int xDistance = Mathf.Abs(d.x);
        int zDistance = Mathf.Abs(d.z);
        int diagonal = Mathf.Min(xDistance, zDistance);
        int straight = Mathf.Abs(xDistance - zDistance);
        return MOVE_DIAGONAL_COST * diagonal + MOVE_STRAIGHT_COST * straight;
    }

    /// <summary>
    /// Retrieves the grid system instance for a given floor index.
    ///
    /// What it does:
    /// - Returns the <see cref="GridSystem{PathNode}"/> corresponding to the specified floor.
    ///
    /// Why this exists in RogueShooter:
    /// - Supports multi-floor pathfinding where each floor maintains its own grid structure.
    /// - Allows systems to query and operate on nodes per-floor without global lookups.
    ///
    /// Implementation notes:
    /// - Assumes grids were created during <see cref="Setup"/> and stored in <c>gridSystemList</c>.
    /// </summary>
    private GridSystem<PathNode> GetGridSystem(int floor) => gridSystemList[floor];

    /// <summary>
    /// Retrieves a single pathfinding node at the given (x, z, floor) position.
    ///
    /// What it does:
    /// - Resolves to the correct grid system (via <see cref="GetGridSystem"/>) and returns its node.
    ///
    /// Why this exists in RogueShooter:
    /// - Simplifies code that frequently needs to access individual nodes by absolute coordinates.
    /// - Used heavily in A*, edge baking, and AI systems for node-level data manipulation.
    ///
    /// Implementation notes:
    /// - Returns <c>null</c> if the grid system or node does not exist (should not normally happen after Setup()).
    /// </summary>
    public PathNode GetNode(int x, int z, int floor)
        => GetGridSystem(floor).GetGridObject(new GridPosition(x, z, floor));

    /// <summary>
    /// Converts a unit orthogonal delta (dx, dz) into an EdgeMask direction.
    ///
    /// What it does:
    /// - Maps (0,+1)→N, (+1,0)→E, (0,-1)→S, (-1,0)→W.
    /// - Returns <see cref="EdgeMask.None"/> for non-orthogonal deltas.
    ///
    /// Why this exists in RogueShooter:
    /// - Used by <see cref="CanStep"/> to check per-edge walls symmetrically for orthogonal moves.
    /// - Keeps edge checks readable and centralized.
    ///
    /// Implementation notes:
    /// - Diagonal deltas are intentionally not mapped (handled separately in <see cref="CanStep"/>).
    /// </summary>
    private EdgeMask DirFromDelta(int dx, int dz)
    {
        if (dx == 0 && dz == +1) return EdgeMask.N;
        if (dx == +1 && dz == 0) return EdgeMask.E;
        if (dx == 0 && dz == -1) return EdgeMask.S;
        if (dx == -1 && dz == 0) return EdgeMask.W;
        return EdgeMask.None;
    }

    /// <summary>
    /// Returns the opposite edge direction (N↔S, E↔W).
    ///
    /// What it does:
    /// - Maps a cardinal edge to its opposite; otherwise returns <see cref="EdgeMask.None"/>.
    ///
    /// Why this exists in RogueShooter:
    /// - Ensures symmetric edge checks (A’s east equals B’s west) in movement validation.
    /// - Avoids “one-way walls” by enforcing consistency across neighboring nodes.
    /// </summary>
    private EdgeMask Opposite(EdgeMask d) => d switch
    {
        EdgeMask.N => EdgeMask.S,
        EdgeMask.E => EdgeMask.W,
        EdgeMask.S => EdgeMask.N,
        EdgeMask.W => EdgeMask.E,
        _ => EdgeMask.None
    };

    /// <summary>
    /// Determines whether movement from cell A to cell B is allowed,
    /// honoring edge walls and preventing diagonal corner-cutting.
    ///
    /// What it does:
    /// - Validates that the delta is a single orthogonal or diagonal step.
    /// - For orthogonal moves: blocks movement if either side of the shared edge has a wall flag.
    /// - For diagonal moves: requires at least one orthogonal “L-shaped” two-step route to be clear
    ///   (A→X→B or A→Z→B), preventing cutting through blocked corners.
    ///
    /// Why this exists in RogueShooter:
    /// - Enforces tactical rules consistent with baked edge data (from EdgeBaker).
    /// - Prevents unrealistic diagonal slips past doorframes/rails and yields robust cover behavior.
    ///
    /// Implementation notes:
    /// - Uses <see cref="DirFromDelta"/> and <see cref="Opposite(EdgeMask)"/> to test symmetric edge walls.
    /// - For diagonals, both intermediate orthogonal neighbors must be valid and walkable before testing paths.
    /// </summary>
    private bool CanStep(GridPosition a, GridPosition b)
    {
        int dx = b.x - a.x;
        int dz = b.z - a.z;

        bool diagonal = Mathf.Abs(dx) == 1 && Mathf.Abs(dz) == 1;
        bool ortho = (dx == 0) ^ (dz == 0);
        if (!diagonal && !ortho) return false; // Disallow jumps longer than 1 cell


        var nodeA = GetNode(a.x, a.z, a.floor);
        var nodeB = GetNode(b.x, b.z, b.floor);

        // ORTHOGONAL MOVE: both sides of the shared edge must be open
        if (ortho)
        {
            var dir = DirFromDelta(dx, dz);
            if (dir == EdgeMask.None) return false;
            if (nodeA.HasWall(dir)) return false;               // wall on A’s side
            if (nodeB.HasWall(Opposite(dir))) return false;     // wall on B’s side
            return true;
        }

        // DIAGONAL MOVE: require at least one clear L-route (no corner clipping)
        var aToX = new GridPosition(a.x + dx, a.z, a.floor);
        var aToZ = new GridPosition(a.x, a.z + dz, a.floor);

        // Both intermediates must be inside bounds and walkable to be considered
        if (!IsValidGridPosition(aToX) || !IsValidGridPosition(aToZ)) return false;
        if (!IsWalkable(aToX) || !IsWalkable(aToZ)) return false;

        // Route 1: A -> X -> B (two orthogonal steps)
        bool pathViaX = CanStep(a, aToX) && CanStep(aToX, b);

        // Route 2: A -> Z -> B (two orthogonal steps)
        bool pathViaZ = CanStep(a, aToZ) && CanStep(aToZ, b);

        return pathViaX || pathViaZ;

    }

    private bool IsValidGridPosition(GridPosition gridPosition)
    {
        return LevelGrid.Instance.GetGridSystem(gridPosition.floor).IsValidGridPosition(gridPosition);
    }

    private bool IsWalkable(GridPosition gridPosition)
    {
        PathNode node = GetNode(gridPosition.x, gridPosition.z, gridPosition.floor);
        return node != null && node.GetIsWalkable();
    }

    /// <summary>
    /// Collects all valid neighbor nodes (up to 8) for A* expansion from the given node.
    ///
    /// What it does:
    /// - Iterates orthogonal and diagonal neighbors within the current floor bounds.
    /// - Filters out non-walkable cells early.
    /// - Uses <see cref="CanStep"/> to enforce edge walls and anti-corner-cutting rules.
    /// - Additionally appends any explicit link targets (e.g., stairs/elevators) connected to this cell.
    ///
    /// Why this exists in RogueShooter:
    /// - Centralizes movement rules so both AI and player pathfinding share identical constraints.
    /// - Supports multi-floor traversal via designer-authored links without special-casing A*.
    ///
    /// Implementation notes:
    /// - Neighbor order is stable to keep behavior deterministic across runs.
    /// - Links bypass edge checks by design (they represent explicit allowed transitions).
    /// </summary>
    private List<PathNode> GetNeighbourList(PathNode currentNode)
    {
        List<PathNode> result = new List<PathNode>(8);

        GridPosition gp = currentNode.GetGridPosition();

        // Candidate offsets (W, SW, NW, E, SE, NE, S, N)
        static IEnumerable<(int dx, int dz)> Offsets()
        {
            yield return (-1, 0); // W
            yield return (-1, -1); // SW
            yield return (-1, +1); // NW

            yield return (+1, 0); // E
            yield return (+1, -1); // SE
            yield return (+1, +1); // NE

            yield return (0, -1); // S
            yield return (0, +1); // N
        }

        // 1) Same-floor neighbors with edge rules
        foreach (var (dx, dz) in Offsets())
        {
            int nx = gp.x + dx;
            int nz = gp.z + dz;

            // Bounds check
            if (nx < 0 || nz < 0 || nx >= width || nz >= height) continue;

            var ngp = new GridPosition(nx, nz, gp.floor);

            // Early reject: must be walkable
            if (!IsWalkable(ngp)) continue;

            // Respect edge blockers and corner rules
            if (!CanStep(gp, ngp)) continue;

            result.Add(GetNode(nx, nz, gp.floor));
        }

        // 2) Explicit links (stairs/lifts/portals) — allowed transitions across floors
        foreach (GridPosition linkGp in GetPathfindingLinkConnectedGridPositionList(gp))
        {
            // Varmista ettei mennä ulos
            if (!IsValidGridPosition(linkGp)) continue;
            if (!IsWalkable(linkGp)) continue;

            // Links intentionally bypass edge checks; they model designer-approved moves
            result.Add(GetNode(linkGp.x, linkGp.z, linkGp.floor));
        }

        return result;
    }

    /// <summary>
    /// Returns all grid positions directly connected to the given position via explicit pathfinding links.
    ///
    /// What it does:
    /// - Searches the prebuilt <see cref="pathfindingLinkList"/> for connections where the given cell
    ///   is either endpoint (A or B).
    /// - Collects and returns the corresponding linked destinations.
    ///
    /// Why this exists in RogueShooter:
    /// - Enables multi-floor traversal and special transitions (stairs, elevators, hatches, ladders, etc.)
    ///   that bypass standard neighbor logic.
    /// - Keeps such transitions data-driven: designers place <see cref="PathfindingLinkMonoBehaviour"/> objects
    ///   in the scene instead of hardcoding connections.
    ///
    /// Implementation notes:
    /// - Links are treated as bidirectional: A↔B.
    /// - The returned positions are later validated for walkability before use.
    /// </summary>
    private List<GridPosition> GetPathfindingLinkConnectedGridPositionList(GridPosition gridPosition)
    {
        List<GridPosition> result = new List<GridPosition>();
        if (pathfindingLinkList == null || pathfindingLinkList.Count == 0) return result;

        foreach (PathfindingLink link in pathfindingLinkList)
        {
            if (link.gridPositionA == gridPosition) result.Add(link.gridPositionB);
            if (link.gridPositionB == gridPosition) result.Add(link.gridPositionA);
        }
        return result;
    }

    /// <summary>
    /// Reconstructs a complete path from the end node by backtracking through parent pointers.
    ///
    /// What it does:
    /// - Traces the <c>CameFrom</c> chain from the goal node back to the start.
    /// - Reverses the collected list and converts it into grid positions for gameplay use.
    ///
    /// Why this exists in RogueShooter:
    /// - Converts A*’s internal node traversal history into a usable list of <see cref="GridPosition"/> steps.
    /// - Provides a deterministic, minimal path sequence for units to follow.
    ///
    /// Implementation notes:
    /// - Result always includes both the start and end positions.
    /// - Returned list is ordered from start → goal.
    /// </summary>
    private List<GridPosition> CalculatePath(PathNode endNode)
    {
        List<PathNode> pathNodes = new List<PathNode> { endNode };
        PathNode current = endNode;

        while (current.GetCameFromPathNode() != null)
        {
            pathNodes.Add(current.GetCameFromPathNode());
            current = current.GetCameFromPathNode();
        }

        pathNodes.Reverse();

        List<GridPosition> gridPositions = new List<GridPosition>(pathNodes.Count);
        foreach (PathNode n in pathNodes) gridPositions.Add(n.GetGridPosition());

        return gridPositions;
    }

    /// <summary>
    /// Returns whether the given grid position is currently walkable.
    ///
    /// Why this exists in RogueShooter:
    /// - Unified query for gameplay/AI to check if a tile can be occupied.
    /// - Mirrors the internal node flag computed during Setup() (raycasts + edge bake).
    /// </summary>
    public bool IsWalkableGridPosition(GridPosition gridPosition)
        => GetGridSystem(gridPosition.floor).GetGridObject(gridPosition).GetIsWalkable();

    /// <summary>
    /// Sets the walkability of a grid position at runtime.
    ///
    /// Why this exists in RogueShooter:
    /// - Dynamic gameplay (e.g., collapses, placed barricades, hazards) can toggle occupancy rules.
    /// - Lets designers/systems override the initial raycast result if needed.
    ///
    /// Implementation notes:
    /// - Consider calling <see cref="EdgeBaker.RebakeEdgesAround"/> if geometry changes near this tile.
    /// </summary>
    public void SetIsWalkableGridPosition(GridPosition gridPosition, bool isWalkable)
        => GetGridSystem(gridPosition.floor).GetGridObject(gridPosition).SetIsWalkable(isWalkable);

    /// <summary>
    /// Lazily resets per-search A* fields on a node using a generation ID guard.
    ///
    /// What it does:
    /// - If the node was last touched in a previous search (generation mismatch),
    ///   resets G/H/F, clears the “came from” pointer, and marks the node with the current generation.
    ///
    /// Why this exists in RogueShooter:
    /// - Avoids per-search heap allocations and dictionary clears by reusing nodes safely.
    /// - Ensures stale scores from earlier searches never leak into the current query.
    ///
    /// Implementation notes:
    /// - Must be called on any node before reading/updating A* fields during a search.
    /// </summary>
    void EnsureInit(PathNode node)
    {
        if (node.LastGenerationID != currentGenerationID)
        {
            node.SetGCost(int.MaxValue);
            node.SetHCost(0);
            node.CalculateFCost();
            node.ResetCameFromPathNode();
            node.MarkGeneration(currentGenerationID);
        }
    }

    /// <summary>
    /// Converts a movement budget in steps to internal cost units.
    ///
    /// Why this exists in RogueShooter:
    /// - Keeps UI/AI logic readable (work in “steps”) while A* uses cost units (10 per orthogonal step).
    /// </summary>
    public static int CostFromSteps(int steps) => steps * MOVE_STRAIGHT_COST;

    /// <summary>
    /// Gets all explicit pathfinding links collected from the scene (stairs, elevators, robes).
    ///
    /// Why this exists in RogueShooter:
    /// - External systems (UI, debugging, AI) may need to inspect or visualize cross-cell/floor connections.
    /// </summary>
    public List<PathfindingLink> GetPathfindingLinks()
    {
        return pathfindingLinkList ?? new List<PathfindingLink>();
    }

    public int GetWidth()
    {
        return width;
    }
    public int GetHeight()
    {
        return height;
    }
}
```

Assets/scripts/Units/UnitPathFinding/PathfindingLink.cs

```
using UnityEngine;

public class PathfindingLink
{
    public GridPosition gridPositionA;
    public GridPosition gridPositionB;
}
```

Assets/scripts/Units/UnitPathFinding/PathfindingLinkMonoBehaviour.cs

```
using UnityEngine;

public class PathfindingLinkMonoBehaviour : MonoBehaviour
{
    public Vector3 linkPositionA;
    public Vector3 linkPositionB;

    void OnDrawGizmos()
    {
        Gizmos.color = Color.yellow;
        Vector3 aW = transform.TransformPoint(linkPositionA);
        Vector3 bW = transform.TransformPoint(linkPositionB);
        Gizmos.DrawSphere(aW, 0.15f);
        Gizmos.DrawSphere(bW, 0.15f);
        Gizmos.DrawLine(aW, bW);
    }

    public PathfindingLink GetPathfindingLink()
    {
        return new PathfindingLink
        {
            gridPositionA = LevelGrid.Instance.GetGridPosition(linkPositionA),
            gridPositionB = LevelGrid.Instance.GetGridPosition(linkPositionB),
        };
    }
}
```

Assets/scripts/Units/UnitPathFinding/PathFindingUpdate.cs

```
using System;
using UnityEngine;

/// <summary>
/// Updates the pathfinding grid when destructible objects are destroyed.
/// </summary>
public class PathFindingUpdate : MonoBehaviour
{
    private void Start()
    {
        DestructibleObject.OnAnyDestroyed += DestructibleObject_OnAnyDestroyed;
    }

    private void DestructibleObject_OnAnyDestroyed(object sender, EventArgs e)
    {
        DestructibleObject destructibleObject = sender as DestructibleObject;
        PathFinding.Instance.SetIsWalkableGridPosition(destructibleObject.GetGridPosition(), true);
    }
}
```

Assets/scripts/Units/UnitPathFinding/PathNode.cs

```
using Unity.VisualScripting;
using UnityEngine;

[System.Flags]
public enum EdgeMask { None=0, N=1, E=2, S=4, W=8 }
public class PathNode
{
    private GridPosition gridPosition;
    private int gCost;
    private int hCost;
    private int fCost;
    private PathNode cameFromPathNode;

    private bool isWalkable = true;
    private EdgeMask walls; // ← ruudun reunaesteet

    public void AddWall(EdgeMask dir) => walls |= dir;
    public void RemoveWall(EdgeMask dir) => walls &= ~dir;
    public bool HasWall(EdgeMask dir) => (walls & dir) != 0;
    public void ClearWalls() => walls = EdgeMask.None;

    public PathNode(GridPosition gridPosition)
    {
        this.gridPosition = gridPosition;
    }

    public int LastGenerationID { get; private set; } = -1;
    public void MarkGeneration(int generationID) => LastGenerationID = generationID;

    public override string ToString()
    {
        return gridPosition.ToString();
    }

    public int GetGCost()
    {
        return gCost;
    }

    public int GetHCost()
    {
        return hCost;
    }

    public int GetFCost()
    {
        return fCost;
    }

    public void SetGCost(int gCost)
    {
        this.gCost = gCost;
    }

    public void SetHCost(int hCost)
    {
        this.hCost = hCost;
    }

    public void CalculateFCost()
    {
        fCost = gCost + hCost;
    }

    public void ResetCameFromPathNode()
    {
        cameFromPathNode = null;
    }

    public void SetCameFromPathNode(PathNode pathNode)
    {
        cameFromPathNode = pathNode;
    }

    public PathNode GetCameFromPathNode()
    {
        return cameFromPathNode;
    }

    public GridPosition GetGridPosition()
    {
        return gridPosition;
    }

    public bool GetIsWalkable()
    {
        return isWalkable;
    }

    public void SetIsWalkable(bool isWalkable)
    {
        this.isWalkable = isWalkable;
    }

    public bool IsWalkable()
    {
        return isWalkable;
    }

}
```

Assets/scripts/Units/UnitRagdoll/RagdollPoseBinder.cs

```
using System.Collections;
using Mirror;
using UnityEngine;

/// <summary>
/// Online: Client need this to get destroyed unit rootbone to create ragdoll form it.
/// </summary>
public class RagdollPoseBinder : NetworkBehaviour
{
    [SyncVar] public uint sourceUnitNetId;
    [SyncVar] public Vector3 lastHitPos;
    [SyncVar] public int overkill;

    [ClientCallback]
    private void Start()
    {
        StartCoroutine(ApplyPoseWhenReady());
    }

    private IEnumerator ApplyPoseWhenReady()
    {
        var (root, why) = TryFindOriginalRootBone(sourceUnitNetId);
        if (root != null)
        {
            if (TryGetComponent<UnitRagdoll>(out var unitRagdoll))
            {
                unitRagdoll.SetOverkill(overkill);
                unitRagdoll.SetLastHitPosition(lastHitPos);
                unitRagdoll.Setup(root);
            }
            yield break;
        }

        Debug.Log($"[Ragdoll] waiting root for netId {sourceUnitNetId} ({why})");

        yield return new WaitForEndOfFrame();
        Debug.LogWarning($"[RagdollPoseBinder] Source root not found for netId {sourceUnitNetId}");
    }

    private static (Transform root, string why) TryFindOriginalRootBone(uint netId)
    {
        if (netId == 0) return (null, "netId==0");
        if (!Mirror.NetworkClient.spawned.TryGetValue(netId, out var id) || id == null)
            return (null, "identity not in NetworkClient.spawned");

        // Löydä UnitRagdollSpawn myös hierarkiasta
        var spawner = id.GetComponent<UnitRagdollSpawn>()
                ?? id.GetComponentInChildren<UnitRagdollSpawn>(true)
                ?? id.GetComponentInParent<UnitRagdollSpawn>();
        if (spawner == null) return (null, "UnitRagdollSpawn missing under identity");
        if (spawner.OriginalRagdollRootBone == null) return (null, "OriginalRagdollRootBone null");
        return (spawner.OriginalRagdollRootBone, null);
    }

}
```

Assets/scripts/Units/UnitRagdoll/UnitRagdoll.cs

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

public class UnitRagdoll : MonoBehaviour
{

    [SerializeField] private Transform ragdollRootBone;

    private Vector3 lastHitPosition;

    private int overkill;

    public Transform Root => ragdollRootBone;

    public void Setup(Transform orginalRootBone)
    {
        MatchAllChildTransforms(orginalRootBone, ragdollRootBone);
      //  Vector3 randomDir = new Vector3(Random.Range(-1f, +1f), 0, Random.Range(-1, +1));
        ApplyPushForceToRagdoll(ragdollRootBone, 500f + overkill, lastHitPosition, 50f);
    }

    /// <summary>
    /// Sets all ragdoll bones to match dying unit bones rotation and position
    /// </summary>
    private static void MatchAllChildTransforms(Transform sourceRoot, Transform targetRoot)
    {
        var stack = new Stack<(Transform sourceBone, Transform targetBone)>();
        stack.Push((sourceRoot, targetRoot));

        while (stack.Count > 0)
        {
            var (currentSourceBone, currentTargetBone) = stack.Pop();

            currentTargetBone.SetPositionAndRotation(currentSourceBone.position, currentSourceBone.rotation);

            if (currentSourceBone.childCount == currentTargetBone.childCount)
            {

                for (int i = 0; i < currentSourceBone.childCount; i++)
                {
                    stack.Push((currentSourceBone.GetChild(i), currentTargetBone.GetChild(i)));
                }
            }
        }
    }

    private void ApplyPushForceToRagdoll(Transform root, float pushForce, Vector3 pushPosition, float PushRange)
    {
        foreach (Transform child in root)
        {
            if (child.TryGetComponent<Rigidbody>(out Rigidbody childRigidbody))
            {
                childRigidbody.AddExplosionForce(pushForce, pushPosition, PushRange);
            }

            ApplyPushForceToRagdoll(child, pushForce, pushPosition, PushRange);
        }
    }

    public void SetLastHitPosition(Vector3 hitPosition)
    {
        lastHitPosition = hitPosition;
    }

    public void SetOverkill(int overkill)
    {
        this.overkill = overkill;
    }
}
```

Assets/scripts/Units/UnitRagdoll/UnitRagdollSpawn.cs

```
using System;
using UnityEngine;


[RequireComponent(typeof(HealthSystem))]
public class UnitRagdollSpawn : MonoBehaviour
{
    [SerializeField] private Transform ragdollPrefab;
    [SerializeField] private Transform orginalRagdollRootBone;
    public Transform OriginalRagdollRootBone => orginalRagdollRootBone;

    private HealthSystem healthSystem;

    // To prevent multiple spawns
    private bool spawned;

    private void Awake()
    {
        healthSystem = GetComponent<HealthSystem>();
        healthSystem.OnDead += HealthSystem_OnDied;
    }

    private void HealthSystem_OnDied(object sender, EventArgs e)
    {
        if (spawned) return;
        spawned = true;
        Vector3 lastHitPosition = healthSystem.LastHitPosition;
        int overkill = healthSystem.Overkill;
        var ni = GetComponentInParent<Mirror.NetworkIdentity>();
        uint id = ni ? ni.netId : 0;

        NetworkSync.SpawnRagdoll(
            ragdollPrefab.gameObject,
            transform.position,
            transform.rotation,
            id,
            orginalRagdollRootBone,
            lastHitPosition,
            overkill);

        healthSystem.OnDead -= HealthSystem_OnDied;
    }
}
```

Assets/scripts/Units/UnitsControlUI/TurnSystemUI.cs

```
using System;
using UnityEngine;
using UnityEngine.UI;
using TMPro;
using Utp;

///<sumary>
/// TurnSystemUI manages the turn system user interface.
/// It handles both singleplayer and multiplayer modes.
/// In multiplayer, it interacts with PlayerController to manage turn ending.
/// It also updates UI elements based on the current turn state.
///</sumary>
public class TurnSystemUI : MonoBehaviour
{
    [SerializeField] private Button endTurnButton;
    [SerializeField] private TextMeshProUGUI turnNumberText;            // (valinnainen, käytä SP:ssä)
    [SerializeField] private GameObject enemyTurnVisualGameObject;      // (valinnainen, käytä SP:ssä)
    [SerializeField] private TextMeshProUGUI playerReadyText;          // (Online)

    bool isCoop;
    private PlayerController localPlayerController;

    void Start()
    {
        isCoop = GameModeManager.SelectedMode == GameMode.CoOp;

        // kiinnitä handler tasan kerran
        if (endTurnButton != null)
        {
            endTurnButton.onClick.RemoveAllListeners();
            endTurnButton.onClick.AddListener(OnEndTurnClicked);
        }

        if (isCoop)
        {
            // Co-opissa nappi on DISABLED kunnes serveri kertoo että saa toimia
            TurnSystem.Instance.OnTurnChanged += TurnSystem_OnTurnChanged;
            SetCanAct(false);
        }
        else
        {
            // Singleplayerissa kuuntele vuoron vaihtumista
            if (TurnSystem.Instance != null)
            {
                TurnSystem.Instance.OnTurnChanged += TurnSystem_OnTurnChanged;
                UpdateForSingleplayer();
            }
        }

        if (playerReadyText) playerReadyText.gameObject.SetActive(false);
    }

    void OnDisable()
    {
        TurnSystem.Instance.OnTurnChanged -= TurnSystem_OnTurnChanged;
    }

    // ====== julkinen kutsu PlayerController.TargetNotifyCanAct:ista ======
    public void SetCanAct(bool canAct)
    {
        if (endTurnButton == null) return;

        endTurnButton.onClick.RemoveListener(OnEndTurnClicked);
        if (canAct) endTurnButton.onClick.AddListener(OnEndTurnClicked);

        endTurnButton.gameObject.SetActive(canAct);   // jos haluat pitää aina näkyvissä, vaihda SetActive(true)
        endTurnButton.interactable = canAct;
    }

    // ====== nappi ======
    private void OnEndTurnClicked()
    {
        // Päättele co-op -tila tilannekohtaisesti (ei SelectedMode)
        bool isOnline =
            NetTurnManager.Instance != null &&
            (GameNetworkManager.Instance.GetNetWorkServerActive() || GameNetworkManager.Instance.GetNetWorkClientConnected());
        if (!isOnline)
        {
            if (TurnSystem.Instance != null)
            {
                TurnSystem.Instance.NextTurn();
            }
            else
            {
                Debug.LogWarning("[UI] TurnSystem.Instance is null");
            }
            return;
        }

        CacheLocalPlayerController();
        if (localPlayerController == null)
        {
            Debug.LogWarning("[UI] Local PlayerController not found");
            return;
        }
        // Istantly lock input
        if (UnitActionSystem.Instance != null)
        {
            UnitActionSystem.Instance.LockInput();
        }
        // Prevent double clicks
        SetCanAct(false);
        // Lähetä serverille
        localPlayerController.ClickEndTurn();

        //Päivitä player ready hud
    }

    private void CacheLocalPlayerController()
    {
        if (localPlayerController != null) return;

        // 1) Varmista helpoimman kautta
        if (PlayerController.Local != null)
        {
            localPlayerController = PlayerController.Local;
            return;
        }

        // 2) Fallback: Mirrorin client-yhteyden identity
        var conn = GameNetworkManager.Instance != null
            ? GameNetworkManager.Instance.NetWorkClientConnection()
            : null;
        if (conn != null && conn.identity != null)
        {
            localPlayerController = conn.identity.GetComponent<PlayerController>();
            if (localPlayerController != null) return;
        }

        // 3) Viimeinen oljenkorsi: etsi skenestä local-pelaaja
        var pcs = FindObjectsByType<PlayerController>(FindObjectsSortMode.InstanceID);
        foreach (var pc in pcs)
        {
            if (pc.isLocalPlayer) { localPlayerController = pc; break; }
        }
    }


    // ====== singleplayer UI (valinnainen) ======
    private void TurnSystem_OnTurnChanged(object s, EventArgs e) => UpdateForSingleplayer();

    private void UpdateForSingleplayer()
    {

        if (turnNumberText != null)
            turnNumberText.text = "Turn: " + TurnSystem.Instance.GetTurnNumber();

        if (enemyTurnVisualGameObject != null)
            enemyTurnVisualGameObject.SetActive(!TurnSystem.Instance.IsPlayerTurn());

        if (endTurnButton != null)
            endTurnButton.gameObject.SetActive(TurnSystem.Instance.IsPlayerTurn());
    }
    // Kutsutaan verkosta
    public void SetTeammateReady(bool visible, string whoLabel = null)
    {
        if (!playerReadyText) return;
        if (visible)
        {
            playerReadyText.text = $"{whoLabel} READY";
            playerReadyText.gameObject.SetActive(true);
        }
        else
        {
            playerReadyText.gameObject.SetActive(false);
        }
    }
}
```

Assets/scripts/Units/UnitsControlUI/UnitActionBusyUI.cs

```
using UnityEngine;

/// <summary>
///     This class is responsible for displaying the busy UI when the unit action system is busy
/// </summary>
public class UnitActionBusyUI : MonoBehaviour
{
    private void Start()
    {
        UnitActionSystem.Instance.OnBusyChanged += UnitActionSystem_OnBusyChanged;
        Hide();
    }

    void OnEnable()
    {
        UnitActionSystem.Instance.OnBusyChanged += UnitActionSystem_OnBusyChanged;
    }

    void OnDisable()
    {
        UnitActionSystem.Instance.OnBusyChanged -= UnitActionSystem_OnBusyChanged;
    }

    private void Show()
    {
        gameObject.SetActive(true);
    }
    private void Hide()
    {
        gameObject.SetActive(false);
    }
    /// <summary>
    ///     This method is called when the unit action system is busy or not busy
    /// </summary>
    private void UnitActionSystem_OnBusyChanged(object sender, bool isBusy)
    {
        if (isBusy)
        {
            Show();
        }
        else
        {
            Hide();
        }
    }
}
```

Assets/scripts/Units/UnitsControlUI/UnitActionButtonUI.cs

```
using UnityEngine;
using UnityEngine.UI;
using TMPro;

/// <summary>
///     This class is responsible for displaying the action button TXT in the UI
/// </summary>

public class UnitActionButtonUI : MonoBehaviour
{
    [SerializeField] private TextMeshProUGUI textMeshPro;
    [SerializeField] private Button actionButton;
    [SerializeField] private GameObject actionButtonSelectedVisual;

    private BaseAction baseAction;

    public void SetBaseAction(BaseAction baseAction)
    {
        this.baseAction = baseAction;
        textMeshPro.text = baseAction.GetActionName().ToUpper();

        actionButton.onClick.AddListener(() =>
        {
            UnitActionSystem.Instance.SetSelectedAction(baseAction);
        } );

    }

    public void UpdateSelectedVisual()
    {
        BaseAction selectedbaseAction = UnitActionSystem.Instance.GetSelectedAction();
        actionButtonSelectedVisual.SetActive(selectedbaseAction == baseAction);
    }

}
```

Assets/scripts/Units/UnitsControlUI/UnitActionSystemUI.cs

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

/// <summary>
///     This class is responsible for displaying the action buttons for the selected unit in the UI.
///     It creates and destroys action buttons based on the selected unit's actions.
/// </summary>

public class UnitActionSystemUI : MonoBehaviour
{

    [SerializeField] private Transform actionButtonPrefab;
    [SerializeField] private Transform actionButtonContainerTransform;
    [SerializeField] private TextMeshProUGUI actionPointsText;

    private List<UnitActionButtonUI> actionButtonUIList;

    private void Awake()
    {
        actionButtonUIList = new List<UnitActionButtonUI>();
    }

    private void Start()
    {
        if (UnitActionSystem.Instance != null)
        {
            UnitActionSystem.Instance.OnSelectedUnitChanged += UnitActionSystem_OnSelectedUnitChanged;
            UnitActionSystem.Instance.OnSelectedActionChanged += UnitActionSystem_OnSelectedActionChanged;
            UnitActionSystem.Instance.OnActionStarted += UnitActionSystem_OnActionStarted;

        } else
        {
            Debug.Log("UnitActionSystem instance found.");
        }
        if (TurnSystem.Instance != null)
        {
            TurnSystem.Instance.OnTurnChanged += TurnSystem_OnTurnChanged;
        } else
        {
            Debug.Log("TurnSystem instance not found.");
        }

        Unit.OnAnyActionPointsChanged += Unit_OnAnyActionPointsChanged;
    }

    /*
    void OnEnable()
    {
        if (UnitActionSystem.Instance != null)
        {
            UnitActionSystem.Instance.OnSelectedUnitChanged += UnitActionSystem_OnSelectedUnitChanged;
            UnitActionSystem.Instance.OnSelectedActionChanged += UnitActionSystem_OnSelectedActionChanged;
            UnitActionSystem.Instance.OnActionStarted += UnitActionSystem_OnActionStarted;

        } else
        {
            Debug.Log("UnitActionSystem instance found.");
        }
        if (TurnSystem.Instance != null)
        {
            TurnSystem.Instance.OnTurnChanged += TurnSystem_OnTurnChanged;
        } else
        {
            Debug.Log("TurnSystem instance not found.");
        }

        Unit.OnAnyActionPointsChanged += Unit_OnAnyActionPointsChanged;
    }
    */
    void OnDisable()
    {
        UnitActionSystem.Instance.OnSelectedUnitChanged -= UnitActionSystem_OnSelectedUnitChanged;
        UnitActionSystem.Instance.OnSelectedActionChanged -= UnitActionSystem_OnSelectedActionChanged;
        UnitActionSystem.Instance.OnActionStarted -= UnitActionSystem_OnActionStarted;
        TurnSystem.Instance.OnTurnChanged -= TurnSystem_OnTurnChanged;
        Unit.OnAnyActionPointsChanged -= Unit_OnAnyActionPointsChanged;
    }

    private void CreateUnitActionButtons()
    {

        Unit selectedUnit = UnitActionSystem.Instance.GetSelectedUnit();
        if (selectedUnit == null)
        {
            Debug.Log("No selected unit found.");
            return;
        }
        actionButtonUIList.Clear();

        foreach (BaseAction baseAction in selectedUnit.GetBaseActionsArray())
        {
            Transform actionButtonTransform = Instantiate(actionButtonPrefab, actionButtonContainerTransform);
            UnitActionButtonUI actionButtonUI = actionButtonTransform.GetComponent<UnitActionButtonUI>();
            actionButtonUI.SetBaseAction(baseAction);
            actionButtonUIList.Add(actionButtonUI);

        }
    }

    private void DestroyActionButtons()
    {
        foreach (Transform child in actionButtonContainerTransform)
        {
            Destroy(child.gameObject);
        }
    }

    private void UnitActionSystem_OnSelectedUnitChanged(object sender, EventArgs e)
    {
        DestroyActionButtons();
        CreateUnitActionButtons();
        UpdateSelectedVisual();
        UpdateActionPointsVisual();
    }

    private void UnitActionSystem_OnSelectedActionChanged(object sender, EventArgs e)
    {
        UpdateSelectedVisual();
    }

    private void UnitActionSystem_OnActionStarted(object sender, EventArgs e)
    {
        UpdateActionPointsVisual();
    }

    private void UpdateSelectedVisual()
    {
        foreach (UnitActionButtonUI actionButtonUI in actionButtonUIList)
        {
            actionButtonUI.UpdateSelectedVisual();
        }
    }

    private void UpdateActionPointsVisual()
    {
        // Jos tekstiä ei ole kytketty Inspectorissa, poistu siististi
        if (actionPointsText == null) return;

        // Jos järjestelmä ei ole vielä valmis, näytä viiva
        if (UnitActionSystem.Instance == null)
        {
            actionPointsText.text = "Action Points: -";
            return;
        }
        Unit selectedUnit = UnitActionSystem.Instance.GetSelectedUnit();
        if (selectedUnit == null)
        {
            actionPointsText.text = "Action Points: -";
            return;
        }
        actionPointsText.text = "Action Points: " + selectedUnit.GetActionPoints();
    }
    /// <summary>
    ///     This method is called when the turn changes. It resets the action points UI to the maximum value.
    /// </summary>
    private void TurnSystem_OnTurnChanged(object sender, EventArgs e)
    {
        UpdateActionPointsVisual();
    }

    /// <summary>
    ///     This method is called when the action points of any unit change. It updates the action points UI.
    /// </summary>
    private void Unit_OnAnyActionPointsChanged(object sender, EventArgs e)
    {
        UpdateActionPointsVisual();
    }

}
```

Assets/scripts/Units/UnitSelectedVisual.cs

```
using System;
using UnityEngine;

/// <summary>
/// This class is responsible for displaying a visual indicator when a unit is selected in the game.
/// It uses a MeshRenderer component to show or hide the visual representation of the selected unit.
/// </summary>
public class UnitSelectedVisual : MonoBehaviour
{
    [SerializeField] private Unit unit;
    [SerializeField] private MeshRenderer meshRenderer;

    private void Awake()
    {
        if (!meshRenderer) meshRenderer = GetComponentInChildren<MeshRenderer>(true);
        if (meshRenderer) meshRenderer.enabled = false;
    }

    private void Start()
    {
        /*
        if (UnitActionSystem.Instance != null)
        {
            UnitActionSystem.Instance.OnSelectedUnitChanged += UnitActionSystem_OnSelectedUnitChanged;
            UpdateVisual();
        }
        */
    }

    void OnEnable()
    {
        if (UnitActionSystem.Instance != null)
        {
            UnitActionSystem.Instance.OnSelectedUnitChanged += UnitActionSystem_OnSelectedUnitChanged;
            UpdateVisual();
        }
    }

    void OnDisable()
    {
        if (UnitActionSystem.Instance != null)
        {
            UnitActionSystem.Instance.OnSelectedUnitChanged -= UnitActionSystem_OnSelectedUnitChanged;
            UpdateVisual();
        }
    }

    /*
    private void OnDestroy()
    {
        if (UnitActionSystem.Instance != null)
            UnitActionSystem.Instance.OnSelectedUnitChanged -= UnitActionSystem_OnSelectedUnitChanged;
    }
    */
    private void UnitActionSystem_OnSelectedUnitChanged(object sender, EventArgs empty)
    {
        UpdateVisual();
    }

    private void UpdateVisual()
    {
        if (!this || meshRenderer == null || UnitActionSystem.Instance == null) return;
        var selected = UnitActionSystem.Instance.GetSelectedUnit();
        meshRenderer.enabled = unit != null && selected == unit;
    }
}
```

Assets/scripts/Units/UnitStatsUI/UnitUIBroadcaster.cs

```
using Mirror;

public class UnitUIBroadcaster : NetworkBehaviour
{
    public static UnitUIBroadcaster Instance { get; private set; }
    void Awake() { if (Instance == null) Instance = this; }

    // Tätä saa kutsua vain serveri (hostin serveripuoli)
    [Server]
    public void BroadcastUnitWorldUIVisibility(bool allready)
    {
        if (!NetworkServer.active) return;

        // käy kaikki serverillä tunnetut unitit läpi
        foreach (var kvp in NetworkServer.spawned)
        {
            var unit = kvp.Value.GetComponent<Unit>();
            if (!unit) continue;

            // serveri voi laskea logiikan: pitääkö tämän unitin AP näkyä
            bool visible = ShouldBeVisible(unit, allready);

            // lähetä client-puolelle että tämän unitin UI asetetaan
            RpcSetUnitUIVisibility(unit.netId, visible);
        }
    }

    // Tätä kutsuu serveri, suoritetaan kaikilla clienteillä
    [ClientRpc]
    private void RpcSetUnitUIVisibility(uint unitId, bool visible)
    {
        if (NetworkClient.spawned.TryGetValue(unitId, out var ni) && ni != null)
        {
            var ui = ni.GetComponentInChildren<UnitWorldUI>();
            if (ui != null) ui.SetVisible(visible);
        }
    }

    // serverilogiikka omistajan perusteella
    [Server]
    private bool ShouldBeVisible(Unit unit, bool allready)
    {
        // Kaikki pelaajat ovat valmiina joten näytetään vain vihollisen AP pisteeet.
        if (allready)
        {
            return unit.IsEnemy();
        }

        // Co-Op
        bool playersPhase = TurnSystem.Instance.IsPlayerTurn();
        bool ownerEnded = false;
        if (unit.OwnerId != 0 &&
            NetworkServer.spawned.TryGetValue(unit.OwnerId, out var ownerIdentity) &&
            ownerIdentity != null)
        {
            var pc = ownerIdentity.GetComponent<PlayerController>();
            if (pc != null) ownerEnded = pc.hasEndedThisTurn;
        }

        // 2) Päätä näkyvyys
        if (playersPhase)
        {
            // Pelaajavaihe: näytä kaikki ei-viholliset, joiden omistaja EI ole lopettanut
            return !unit.IsEnemy() && !ownerEnded;
        }
        else
        {
            // Vihollisvaihe: näytä vain viholliset
            return unit.IsEnemy();
        }
    }
}
```

Assets/scripts/Units/UnitStatsUI/UnitWorldUI.cs

```
using UnityEngine;
using TMPro;
using System;
using UnityEngine.UI;
using Mirror;
using System.Collections.Generic;

/// <summary>
/// Displays world-space UI for a single unit, including action points and health bar.
/// Reacts to turn events and ownership rules to show or hide UI visibility
/// </summary>
public class UnitWorldUI : MonoBehaviour
{
    [SerializeField] private TextMeshProUGUI actionPointsText;
    [SerializeField] private Unit unit;
    [SerializeField] private Image healthBarImage;
    [SerializeField] private HealthSystem healthSystem;

    /// <summary>
    /// Reference to the unit this UI belongs to.
    /// Which object's visibility do we want to change?
    /// </summary>
    [Header("Visibility")]
    [SerializeField] private GameObject actionPointsRoot;

    /// <summary>
    /// Cached network identity for ownership.
    /// </summary>
    private NetworkIdentity unitIdentity;


    // --- NEW: tiny static registry for ready owners (co-op only) ---
   // private static readonly HashSet<uint> s_readyOwners = new();
  //  public static bool HasOwnerEnded(uint ownerId) => s_readyOwners.Contains(ownerId);

    private void Awake()
    {
        unitIdentity = unit ? unit.GetComponent<NetworkIdentity>() : GetComponentInParent<NetworkIdentity>();
    }

    private void Start()
    {
        // unitIdentity = unit ? unit.GetComponent<NetworkIdentity>() : GetComponentInParent<NetworkIdentity>();

        Unit.OnAnyActionPointsChanged += Unit_OnAnyActionPointsChanged;
        healthSystem.OnDamaged += HealthSystem_OnDamaged;
        UpdateActionPointsText();
        UpdateHealthBarUI();
        // Co-opissa. Ei paikallista seurantaa.Ainoastaan alku asettelu
        if (GameModeManager.SelectedMode == GameMode.CoOp)
        {
            if (unit.IsEnemy())
            {
                actionPointsRoot.SetActive(false);
            }

            return;
        }


        PlayerLocalTurnGate.LocalPlayerTurnChanged += PlayerLocalTurnGate_LocalPlayerTurnChanged;
        PlayerLocalTurnGate_LocalPlayerTurnChanged(PlayerLocalTurnGate.LocalPlayerTurn);

    }

    /*
    private void OnEnable()
    {
        Unit.OnAnyActionPointsChanged += Unit_OnAnyActionPointsChanged;
        healthSystem.OnDamaged += HealthSystem_OnDamaged;
        PlayerLocalTurnGate.LocalPlayerTurnChanged += PlayerLocalTurnGate_LocalPlayerTurnChanged;
    }
    */

    private void OnDisable()
    {
        Unit.OnAnyActionPointsChanged -= Unit_OnAnyActionPointsChanged;
        healthSystem.OnDamaged -= HealthSystem_OnDamaged;
        PlayerLocalTurnGate.LocalPlayerTurnChanged -= PlayerLocalTurnGate_LocalPlayerTurnChanged;
    }

    private void OnDestroy()
    {
        Unit.OnAnyActionPointsChanged -= Unit_OnAnyActionPointsChanged;
        healthSystem.OnDamaged -= HealthSystem_OnDamaged;
        PlayerLocalTurnGate.LocalPlayerTurnChanged -= PlayerLocalTurnGate_LocalPlayerTurnChanged;
    }

    private void UpdateActionPointsText()
    {
        actionPointsText.text = unit.GetActionPoints().ToString();
    }

    private void Unit_OnAnyActionPointsChanged(object sender, EventArgs e)
    {
        UpdateActionPointsText();
    }

    private void UpdateHealthBarUI()
    {
        healthBarImage.fillAmount = healthSystem.GetHealthNormalized();
    }

    /// <summary>
    /// Event handler: refreshes the health bar UI when this unit takes damage.
    /// </summary>
    private void HealthSystem_OnDamaged(object sender, EventArgs e)
    {
        UpdateHealthBarUI();
    }

    /// <summary>
    /// SinglePlayer/Versus: paikallinen turn-gate. Co-opissa ei käytetä.
    /// </summary>
    private void PlayerLocalTurnGate_LocalPlayerTurnChanged(bool canAct)
    {
        if (GameModeManager.SelectedMode == GameMode.CoOp) return; // Co-op: näkyvyys tulee RPC:stä
        if (!this || !gameObject) return;

        bool showAp;
        if (GameModeManager.SelectedMode == GameMode.SinglePlayer)
        {
            showAp = canAct ? !unit.IsEnemy() : unit.IsEnemy();
        }
        else // Versus
        {
            bool unitIsMine = unitIdentity && unitIdentity.isOwned;
            showAp = (canAct && unitIsMine) || (!canAct && !unitIsMine);
        }

        actionPointsRoot.SetActive(showAp);
    }

    public void SetVisible(bool visible)
    {
        actionPointsRoot.SetActive(visible);
    }
}
```

Assets/scripts/Weapons/BulletProjectile.cs

```
using Mirror;
using UnityEngine;

public class BulletProjectile : NetworkBehaviour
{
    [SerializeField] private TrailRenderer trailRenderer;
    [SerializeField] private Transform bulletHitVfxPrefab;

    [SyncVar] private Vector3 targetPosition;


    public void Setup(Vector3 targetPosition)
    {
        this.targetPosition = targetPosition;
    }

    public override void OnStartClient()
    {
        base.OnStartClient();

        if (trailRenderer && !trailRenderer.emitting) trailRenderer.emitting = true;
    }

    private void Update()
    {
        Vector3 moveDirection = (targetPosition - transform.position).normalized;

        float distanceBeforeMoving = Vector3.Distance(transform.position, targetPosition);

        float moveSpeed = 200f; // Adjust the speed as needed
        transform.position += moveSpeed * Time.deltaTime * moveDirection;

        float distanceAfterMoving = Vector3.Distance(transform.position, targetPosition);

            // Check if we've reached or passed the target position
        if (distanceBeforeMoving < distanceAfterMoving)
        {
            transform.position = targetPosition;

            if (trailRenderer) trailRenderer.transform.parent = null;

            if (bulletHitVfxPrefab)
                Instantiate(bulletHitVfxPrefab, targetPosition, Quaternion.identity);

            // Network-aware destruction
            if (isServer) NetworkServer.Destroy(gameObject);
            else Destroy(gameObject);
        }

    }
}
```

Assets/scripts/Weapons/GranadeProjectile.cs

```
using System;
using UnityEngine;
using Mirror;
using System.Collections;

public class GrenadeProjectile : NetworkBehaviour
{
    public static event EventHandler OnAnyGranadeExploded;

    [SerializeField] private Transform granadeExplodeVFXPrefab;
    [SerializeField] private float damageRadius = 4f;
    [SerializeField] private int damage = 30;
    [SerializeField] private float moveSpeed = 15f;
    [SerializeField] private AnimationCurve arcYAnimationCurve;

    [SyncVar(hook = nameof(OnTargetChanged))] private Vector3 targetPosition;

    private float totalDistance;
    private Vector3 positionXZ;
    private const float MIN_DIST = 0.01f;

    private bool isExploded = false;

    private bool _ready;

    public override void OnStartClient()
    {
        base.OnStartClient();
    }

    public void Setup(Vector3 targetWorld)
    {
        var groundTarget = SnapToGround(targetWorld);
        // Aseta SyncVar, hook kutsutaan kaikilla (server + clientit)
        targetPosition = groundTarget;
        RecomputeDerived(); // varmistetaan serverillä heti
        _ready = true;
    }

    private Vector3 SnapToGround(Vector3 worldXZ)
    {
        return new Vector3(worldXZ.x, 0f, worldXZ.z);
    }

    void OnTargetChanged(Vector3 _old, Vector3 _new)
    {
        // Kun SyncVar saapuu clientille, laske johdetut kentät sielläkin
        RecomputeDerived();
        _ready = true;
    }
    private void RecomputeDerived()
    {
        positionXZ = transform.position;
        positionXZ.y = 0f;

        totalDistance = Vector3.Distance(positionXZ, targetPosition);
        if (totalDistance < MIN_DIST) totalDistance = MIN_DIST; // suoja nollaa vastaan
    }

    private void Update()
    {
        if (!_ready || isExploded) return;

        Vector3 moveDir = targetPosition - positionXZ;
        if (moveDir.sqrMagnitude < 1e-6f) moveDir = Vector3.forward; // varadir, ettei normalized → NaN
        moveDir.Normalize();

        positionXZ += moveSpeed * Time.deltaTime * moveDir;

        float distance = Vector3.Distance(positionXZ, targetPosition);
        if (totalDistance < 1e-6f) totalDistance = 0.01f;
        float distanceNormalized = 1f - (distance / totalDistance);
        distanceNormalized = Mathf.Clamp01(distanceNormalized);

        float maxHeight = totalDistance / 4f;
        float positionY = arcYAnimationCurve != null
            ? arcYAnimationCurve.Evaluate(distanceNormalized) * maxHeight
            : 0f;

        if (float.IsNaN(positionY)) positionY = 0f;                   // viimeinen pelastus
        transform.position = new Vector3(positionXZ.x, positionY, positionXZ.z);

        float reachedTargetDistance = .2f;


        if ((Vector3.Distance(positionXZ, targetPosition) < reachedTargetDistance) && !isExploded)
        {
            isExploded = true;
            if (NetworkServer.active || !NetworkClient.isConnected) // Server or offline
            {
                Collider[] colliderArray = Physics.OverlapSphere(targetPosition, damageRadius);

                foreach (Collider collider in colliderArray)
                {
                    if (collider.TryGetComponent<Unit>(out Unit targetUnit))
                    {
                        NetworkSync.ApplyDamageToUnit(targetUnit, damage, targetPosition);
                    }
                    if (collider.TryGetComponent<DestructibleObject>(out DestructibleObject targetObject))
                    {
                         NetworkSync.ApplyDamageToObject(targetObject, damage, targetPosition);
                    }
                }
            }

            // Screen Shake
            OnAnyGranadeExploded?.Invoke(this, EventArgs.Empty);
            // Explode VFX
            Instantiate(granadeExplodeVFXPrefab, targetPosition + Vector3.up * 1f, Quaternion.identity);

            if (!NetworkServer.active)
            {
                Destroy(gameObject);
                return;
            }

            // Online: Hide Granade before destroy it, so that client have time to create own explode VFX from orginal Granade pose.
            SetSoftHiddenLocal(true);
            RpcSetSoftHidden(true);
            StartCoroutine(DestroyAfter(0.30f));
        }
    }

    private IEnumerator DestroyAfter(float seconds)
    {
        yield return new WaitForSeconds(seconds);
        NetworkServer.Destroy(gameObject);
    }

    [ClientRpc]
    private void RpcSetSoftHidden(bool hidden)
    {
        SetSoftHiddenLocal(hidden);
    }

    private void SetSoftHiddenLocal(bool hidden)
    {
        foreach (var r in GetComponentsInChildren<Renderer>())
        {
            r.enabled = !hidden;
        }
    }
}
```